Electronic Notes in Theoretical Computer Science 15 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume15.html 13 pages

Maude as a Metalanguage [1]

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

[a] *Department of Philosophy, University of Navarre, Spain*

[b] *SRI International, Menlo Park, CA 94025, USA*

[c] *Facultad de Ciencias Matemáticas, Universidad Complutense, Madrid, Spain*

[d] *CICA (Centro de Informática Científica de Andalucía), Seville, Spain*

Abstract

One of the key goals of rewriting logic from its beginning has been to provide a *semantic* and *logical framework* in which many models of computation and languages can be naturally represented. There is by now very extensive evidence supporting the claim that rewriting logic is indeed a very flexible and simple logical and semantic framework. From a language design point of view the obvious question to ask is: how can a rewriting logic language best support logical and semantic framework applications, so that it becomes a *metalanguage* in which a very wide variety of logics and languages can be both *semantically defined*, and *implemented*? Our answer is: by being *reflective*. This paper discusses our latest language design and implementation work on Maude as a reflective metalanguage in which entire environments—including syntax definition, parsing, pretty printing, execution, and input/output—can be defined for a language or logic $\mathcal{L}$ of choice.

1 Introduction

This paper is an overview of our latest language design and implementation work on Maude. The goals of language design and implementation are at the service of the uses that we envision for Maude, and in turn, these should help furthering the overall goals of the rewriting logic research program.

We are particularly interested in supporting uses of rewriting logic as a logical and semantic framework, in which many different logics, models of computation, and languages can be represented, can be given a precise semantics, and can be executed. Since rewriting logic is used as what we might

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

call a *metalogic*, it seems very natural to conceive a rewriting logic language as a *metalanguage*, in which we can create executable *environments* for many other languages and logics. Such metalanguage capabilities are among our highest priorities in designing Maude.

Rewriting logic is itself reflective [12,7]. By efficiently supporting this reflection in Maude through its predefined module `META-LEVEL`, we in fact obtain a very useful *gateway* to the metatheory of rewriting logic, enabling Maude to be used as a metalanguage. The essential reason for the appropriateness of this solution is that the desired representation maps and environments manipulate metalevel entities such as program or specification modules in the object language of interest and in Maude. Reflection then gives us the desired disciplined access to such a metalevel within the logic, and underlies very generic techniques for syntax definition, parsing, pretty printing and input/output that allow us to build language environments in Maude. We explain all this in more detail—and how it is supported by the Maude language and implementation—in the rest of the paper.

It is useful to distinguish a CoreMaude sublanguage of *flat* functional and system modules—corresponding, respectively, to equational and rewriting logic specifications that do not have any submodules. This core language is efficiently supported by Maude's rewriting engine, as it is discussed—along with other aspects of the Maude system—in Section 2. The reflective capabilities are then supported by the module `META-LEVEL`. Section 3 explains how this module provides the cornerstone for making Maude highly extensible and flexible. For extending Maude, and, more generally, for creating an environment for any other language $\mathcal{L}$ using Maude, we need generic syntax definition, meta-parsing, and meta-pretty printing capabilities that can deal with expressions in any language, including languages like Maude whose modules have user-definable syntax. And we need a general facility for input/output that can be customized for each language of interest. Section 4 explains how all this can be done in Maude thanks to its reflective design. We conclude the paper with some remarks about related work and future plans.

2 The CoreMaude Interpreter

The Maude system is built around the CoreMaude interpreter, which is implemented in C++ and consists of two parts, each of which is made up of a number of components.

2.1 The Rewrite Engine

The rewrite engine is an improved version of that described in [11]. The overall design 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 a particular theory, such as equations, rules, sorts, and connected components of sorts. 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 source level tracing and the possibility of adding user interrupts and debugging facilities. 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.

2.2 The Mixfix Frontend

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 mixfix parts of Maude over the user's signature), a context free parser, the built-in functions for the metalevel, together with a considerable amount of "glue" code which holds everything together. Many of the C++ classes in this part of the system are derived from those in the rewrite engine.

2.3 The SCP Parsing Algorithm

The current version of Maude implements a parser based on the SCP parsing algorithm [31]. Basically, SCP is a bidirectional, bottom-up and event-driven parser for unrestricted context-free grammars. From a mathematical point of view, we have proved that SCP is sound and complete for the set of CFGs. From a computational perspective, SCP avoids overparsing [30], a common problem of CF parsers like Earley, chart or GLR. The use of multi-virtual trees [29] at the level of representation and the relations of coverage, partial derivability and adjacency as top-down predictions over the basic bottom-up strategy, obtain a high level of efficiency without diminishing the generality of the algorithm.

2.4 Other Aspects of the System

The current Maude system exhibits many other important properties. It supports a metalevel with efficient rewriting and representation-shifting operations. Heavy use is made of the rewrite engine's extensible design in the implementation of many aspects of the metalevel. The current mixfix frontend supports flat functional and system modules, i.e., there is little or no support for importation, renaming, parameterization or object-oriented modules. Typical performance with the current version is 800K-840K free-theory rewrites per second and 27K-111K AC-theory rewrites per second on standard hardware (300MHz P6). 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.

3 The Module `META-LEVEL`

Informally, a reflective logic is a logic in which important aspects of its metatheory can be represented at the object level in a consistent way, so that the object-level representation correctly simulates the relevant metatheoretic aspects. In other words, a logic which can be faithfully interpreted in itself. Rewriting logic is a reflective logic [12,7] in the precise sense that there is a finitely presented *universal* rewrite theory $\mathcal{U}$ that can simulate all other finitely presented theories, including itself. Maude's language design and implementation make systematic use of this fact to give the user a *gateway* to the metatheory of rewriting logic in a clear and principled way. This gateway is the predefined functional module `META-LEVEL`, that provides the user with key functionality in the universal theory for rewriting logic. In other words, the module `META-LEVEL` can be seen as a (partial) specification of the metatheory of rewriting logic. In particular, in the module `META-LEVEL`:

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

The operations `meta-reduce`, `meta-apply`, and `meta-rewrite`, the parsing and pretty printing operations, and the sort operations, are all functions equationally definable in the module `META-LEVEL`. However, for efficiency and convenience, they are built-in functions in Maude. The Maude implementation produces the exact same behavior as if `META-LEVEL` had been defined as a normal functional Maude module, that could itself be represented as a term of sort `Module` in `META-LEVEL`. In this way, a "reflective tower" with an arbitrary number of levels of reflection is supported. A detailed explanation of the signature and operations of the current version of `META-LEVEL` is given in [8].

Notice that, since the Maude system is essentially a particular implementation of the metatheory of rewriting logic, the module `META-LEVEL` can also be used as a gateway to the Maude system itself, so that the user can effectively re-design Maude (in Maude) to fit his/her particular computational needs. Following this line of thought:

- Clavel and Meseguer [13,7] have shown how user-definable *internal strategy languages*, that typically extend the module `META-LEVEL`, can be used to change the (default) operational semantics of Maude for system modules;
- Durán and Meseguer [15] have explained how Maude's parameterized programming style in the Clear and OBJ tradition [5,17] can be internalized in Maude in an extension of `META-LEVEL` and enriched with new modes of parameterization, new methods of program composition, and new ways of defining views of other program modules; and
- Maude's (default) syntax for programs can be re-defined in an extension of `META-LEVEL`, so that programs written in a new user-defined syntax will be understood by Maude's parser (see Sections 4.1 and 4.2).

This potentiality of the module `META-LEVEL` as a building block that users can employ to build customized versions of Maude has inspired us to make it the cornerstone of the Maude system design. The part of the Maude system implemented in C++ (the CoreMaude interpreter presented in Section 2) basically consists of the rewrite engine and a parser for *flat* modules. The rest of the system is fully specified in Maude as an extension of the module `META-LEVEL`. We plan to further optimize the module `META-LEVEL` in order to make metalevel computations even more efficient, and to extend it with new built-in functions to increase its usability and range of applications.

4 Maude as a Metalanguage

One of the key goals of rewriting logic from its beginning has been to provide a *semantic framework* in which many models of computation—particularly concurrent and distributed ones—and languages can be naturally represented. Because of the intrinsic duality between logic and computation that rewriting logic supports, the very same reasons making rewriting logic a suitable semantic framework, make it also an attractive *logical framework* [20] to represent many different logics. Thanks to the sustained efforts of many researchers, particularly in the ELAN, Pisa, Stanford, and Maude teams, there is by now very extensive evidence supporting the claim that rewriting logic is indeed a very flexible and simple semantic framework [21,23,24,4], and logical framework [20,18,34,19,3,32,6,14,7,10]. Moreover, object-oriented design languages, architectural description languages, and languages for distributed components also have a natural semantics in rewriting logic [35,25,33,27,28].

What is common to all these applications is that the models of computation, logics, or languages are represented in rewriting logic by mappings of the form

$$\Phi : \mathcal{L} \longrightarrow \mathit{RWLogic}.$$

¿From a language design point of view the obvious question to ask is: how can a rewriting logic language best support such representation maps, so that it becomes a *metalanguage* in which a very wide variety of programming, specification, and design languages, and of computational and logical systems can be both *semantically defined*, and *implemented* in it?

Our answer is: by being reflective. Specifically, we can define an abstract data type $\mathtt{Module}_{\mathcal{L}}$ representing modules in the logic or language $\mathcal{L}$, and *internalize* a representation map Φ as an equationally defined function

$$\overline{\Phi} : \mathtt{Module}_{\mathcal{L}} \longrightarrow \mathtt{Module},$$

where `Module` is the data type representing finitely presentable rewrite theories in the language—in Maude this data type is part of the `META-LEVEL` module (see Section 3). In fact, thanks to the general metaresult of Bergstra and Tucker [2], any computable representation map Φ can be specified in this way by a finite number of Church-Rosser and terminating equations. Using this metalevel gateway, we can then execute in Maude the rewrite theory $\overline{\Phi}(M)$ associated to a module M in $\mathcal{L}$. This has been done, for example, for linear logic in [7], for structured Maude modules in [15], and could be done for a very wide range of other languages and logics using the same method.

In practice, however, we would like to be able not only to *represent* and *transform* the modules of a language $\mathcal{L}$ as terms of a data type $\mathtt{Module}_{\mathcal{L}}$ within Maude. We would like to use Maude to generate a whole *environment* for $\mathcal{L}$, including a facility for defining and modifying the language's syntax, an input/output facility, a parser, a pretty printer, and an execution environment for it. Furthermore, we would like to be able to do this for languages that—like Maude itself, and many other formal specification languages—have modules with *user-definable* syntax, so that expressions in those modules cannot be parsed with a fixed syntax for the language, but need to be parsed with the particular syntax introduced in the module.

4.1 Syntax Definition

In order to generate in Maude a whole *environment* for a language $\mathcal{L}$, the first thing we need to do is to define the syntax for $\mathcal{L}$-modules. This can be done by extending the module `META-LEVEL` with a data type $\mathtt{Module}_{\mathcal{L}}$ for $\mathcal{L}$-modules, and with other auxiliary data types for commands and other constructs. This can be easily and naturally done using the mixfix frontend, and the built-in data types `Token` (any identifier) and `Bubble` (any string of identifiers). The intuition behind these types is that they correspond to pieces of a module in a language that can only be parsed once the grammar introduced by the signature of the module is available.

The idea is that for a language that allows modules with user-definable syntax—as it is the case of Maude—it is natural to see its syntax as a combined syntax, at two different levels: what we may call the *top level* syntax of the language, and the user-definable syntax introduced in each module. The data types `Token` and `Bubble` allow us to reflect this duality of levels in the syntax definition.

To illustrate these concepts, suppose that we want to define the syntax of Maude in Maude. Consider the following Maude module:

```
fmod NAT3 is
     sort Nat3 .
     op 0 :  -> Nat3 .
     op s_ :  Nat3 -> Nat3 .
     eq [ s s s 0 ] = [ 0 ] .
endfm
```

Notice that the strings of characters inside the boxes are not part of the top level syntax of Maude. In fact, they can only be parsed with the grammar associated to the signature of the module `NAT3`. A correct and useful (in terms of defining afterwards a meaningful parser) definition of the syntax of Maude in Maude must reflect this duality of syntax levels. We show below parts of the module `MAUDE` that defines the syntax for functional modules in Maude:

```
fmod MAUDE is
     sort PreModule PreCommand .
     subsort Decl < DeclList .
     op eq_=_. : Bubble Bubble -> Decl .
     op fmod_is_endfm : Token DeclList -> PreModule .
     op red in_:_. : Token Bubble -> PreCommand .
     ...
```

Notice how we explicitly declare operators that correspond to the top level syntax of Maude, and represent as terms of sort `Bubble` those pieces of the module that can only be parsed with the user-defined syntax; for example, the left and righthand sides of an equation.

Then, the functional module `NAT3` above can be parsed as a term of sort `PreModule` in `MAUDE`. The name of this sort reflects the fact that not all terms of `PreModule` actually represent Maude modules. In particular, for a term of sort `PreModule` to represent a Maude module all the `Bubble`s must be correctly parsed in the user-defined syntax. We will come back to this important point in the next section. Finally, notice that we have also defined the syntax for commands like `reduce` in Maude. We will see in Section 4.4 how the user can define the semantics of these commands.

4.2 Parsing and Pretty Printing

A built-in function `meta-parse` is declared in `META-LEVEL` with syntax

```
op meta-parse : Module Bubble -> Term .
```

The function `meta-parse`$(M,\ i_1 \ldots i_n)$ checks whether the string of identifiers $i_1 \ldots i_n$ is a well-formed term in the module represented by M; if this is the case, `meta-parse` returns the representation of $i_1 \ldots i_n$ as a term of sort `Term` in `META-LEVEL`; otherwise, it returns `error*`.

This built-in function can be very useful to specify in Maude an efficient parser for a language $\mathcal{L}$ that allows modules with user-definable syntax. For example, a parser for `Premodule`s in Maude essentially consists of a function `parse-premodule` with syntax

```
op parse-premodule : PreModule -> Module? .
```

Given a `PreModule` M, `parse-premodule` generates, first, a `Module` M' that represents a module in Maude with the exact same signature as M but without equations; then, `parse-premodule` attempts to transform the `PreModule` M into a `Module` M'' by reducing each `Bubble` b in M to a `Term` t, where t is the successful result of `meta-parse`(M', b). In case of failure, an error term in the supersort `Module?` of `Module` is returned.

The inverse function to `meta-parse` is also included as a built-in function in the module `META-LEVEL`, with syntax:

```
op meta-pretty-print : Module Term -> Bubble .
```

It takes as arguments the representation of a module M and the representation of a term t. It returns a string of identifiers produced by pretty printing t in the syntax given by M.

4.3 Module Algebra and Execution

We are interested in using Maude to build environments for languages $\mathcal{L}$ such as formal specification languages and modular programming languages whose modules can have user-definable syntax. In addition, such modules can be highly structured and parameterized. That is, there can be a rich collection of module composition operations endowing $\mathcal{L}$ with a *module algebra*. In such cases we typically have two data types of modules, a data type $\texttt{Module}_{\mathcal{L}}$ of *flat* or unstructured modules, and a more general data type $\texttt{StrModule}_{\mathcal{L}}$ of structured modules.

The point is that both data types and all the module algebra operations for $\mathcal{L}$ can be defined within Maude as an extension of the module `META-LEVEL`. In this way, the environment that we can build for $\mathcal{L}$ using Maude can also support all the module composition operations of $\mathcal{L}$. Among those module operations a common and important one is *flattening*, that is, the process of passing from a structured module to its unstructured flat form. This can be understood as a function

$$\texttt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^\flat} \texttt{Module}_{\mathcal{L}}.$$

Since modularity constructs can change from language to language, it may be simpler to represent $\mathcal{L}$ in rewriting logic by representing only its flat modules, that is, by a function $\overline{\Phi} : \texttt{Module}_{\mathcal{L}} \longrightarrow \texttt{Module}$ which makes the language $\mathcal{L}$ executable on top of Maude. But using the function $(\_)^\flat$ we can also make structured modules in $\mathcal{L}$ executable by means of the function composition

$$\texttt{StrModule}_{\mathcal{L}} \xrightarrow{(\_)^\flat} \texttt{Module}_{\mathcal{L}} \xrightarrow{\overline{\Phi}} \texttt{Module}.$$

The first most obvious language $\mathcal{L}$ to which we can apply these ideas is Maude itself. The CoreMaude sublanguage has flat modules represented by the data type `Module`. But general Maude modules can be structured and parameterized, and can contain very complex *module expressions* that instantiate and rename several, possibly parameterized, modules; and all this can also happen for object-oriented modules. A module algebra for Maude written in Maude is presented in [15]. In particular, such an algebra contains as well a flattening function making structured Maude modules executable in the Maude engine.

4.4 Input/Output

Using object-oriented concepts, we can specify in Maude a general input/output facility provided by a module `LOOP` that extends the module `META-LEVEL` into a generic read-eval-print loop. This facility can then be specialized for each language $\mathcal{L}$. There is a class `I/O` of input/output objects acting on behalf of users, with two attributes: an input and an output buffer storing `Bubbles`. There is also a class `System` with an input buffer of sort `Input`, which is defined as a supersort of `PreModule` and `PreCommand`, an output buffer of sort `Bubble`, and an attribute of sort `Database`. The sort `Database` is left completely unspecified, so that, depending on the language $\mathcal{L}$ of interest it can be specialized accordingly. For example, for Maude it will contain the current database of modules already entered in the system.

```
subsorts PreCommand PreModule < Input .
class I/O | input : Bubble, output : Bubble .
class System | db : Database, input : Input, output : Bubble .
```

The module `LOOP` provides a *persistent* pair of objects: one of class `I/O` called `user`, and one of class `System` called `system` that interact with each other by exchanging data in and out between their buffers.

For each particular language both the sort `Database` for the `system` object and the additional rewrite rules defining the system behavior are specified according to the specific details of the language in question. We illustrate below the case of Maude. Processing of a `PreModule` once it has been entered into the system is done by the rule

```
var DB : Database . var PM: PreModule .
var I : Identifier . var B : Bubble .

rl [premodule] :
   < system : System | db : DB, input : PM >
   => < system : System | db : processPreModule(PM, DB),
                          input : empty > .
```

The function `processPreModule` attempts to parse the `PreModule` using the function `parse-premodule`, and, if it succeeds, it introduces the resulting `Module` into the database.

Then, for user-defined commands as `red in _:_`, we can define rules of the form

```
rl [reduce] :
   < system : System | db : DB, input : red in I : B . >
   => < system : System | input : empty,
                 output : meta-pretty-print(getModule(I, DB),
                            meta-reduce(getModule(I, DB),
                              meta-parse(getModule(I, DB), B))) > .
```

where `getModule` is a function that extracts from the database the flat module whose name is given in its first argument.

5 Concluding Remarks

We have explained our current design and implementation work to support logical and semantic framework applications of Maude as a rewriting logic metalanguage. This work should be seen as a further step in the context of previous efforts to use algebraic languages to give executable semantic definitions of other languages, including OBJ [16], ASF+SDF [1], action semantics [26], and ELAN [18]. What seems to be new is the systematic use of reflection to achieve more powerful metalanguage functionalities, as well as the capacity to deal with languages that can be extended by user-definable syntax. In particular, when these capabilities are applied to Maude itself, they allow new very flexible ways of extending the language and its module operations, and of changing its syntax.

Much more work remains ahead. The first, most immediate goal is the upcoming release of Maude in the next few months, that will include the metalanguage capabilities described here. This will enable many new logical and semantic framework applications to be developed, such as execution environments for logics, theorem provers, architecture description languages, specification languages, and natural language processing systems. After the release of Maude we will probably concentrate on further extensions of the language such as built-in unification modulo different equational theories—that will permit narrowing computations—, built-in objects and foreign interface modules, and Mobile Maude, a distributed and mobile extension. We will also work on further developing Maude's theorem proving environment [9].

Acknowledgement

We thank Peter Mosses for his remarks on a draft of this paper. The work of Maude is part of a larger international collaboration on the rewriting logic research program, including important contributions to the tool-building effort by the ELAN and CafeOBJ groups, and many theoretical developments and applications [24]. We are particularly grateful to those researchers with whom we have worked most closely on Maude and rewriting logic ideas, including our fellow members at the Logic and Specification Group at SRI, as well as Adel Bouhoula, Roberto Bruni, Jean-Pierre Jouannaud, Ugo Montanari, and Carolyn Talcott.

References

[1] J. Bergstra, J. Heering, and P. Klint. *Algebraic Specification.* ACM Press, 1989.

[2] J. Bergstra and J. Tucker. Characterization of computable data types by means of a finite equational specification method. In J. W. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming, Seventh Colloquium*, LNCS 81, pages 76–90. Springer-Verlag, 1980.

[3] P. Borovanský. Implementation of higher-order unification based on calculus of explicit substitutions. In M. Bartosek, J. Staudek, and J. Wiedermann, editors, *Proc. SOFTSEM'95*, LNCS 1012, pages 363–368. Springer-Verlag, 1995.

[4] R. Bruni, J. Meseguer, and U. Montanari. Process and term tile logic. Technical Report SRI-CSL-98-06, SRI International, July 1998.

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

[6] C. Castro. An approach to solving binary CSP using computational systems. In Meseguer [22], pages 245–264.

[7] M. Clavel. *Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language.* PhD thesis, University of Navarre, 1998. To be published by CSLI Publications, Stanford University.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, and J. Meseguer. Metalevel computation in Maude. In this volume.

[9] M. Clavel, F. Durán, S. Eker, and J. Meseguer. Design and implementation of the Cafe prover and Church-Rosser checker tools. Technical report, SRI International, December 1997.

[10] 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.

[11] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In Meseguer [22], pages 65–89.

[12] M. Clavel and J. Meseguer. Axiomatizing reflective logics and languages. In G. Kiczales, editor, *Proc. Reflection'96, San Francisco, California, April 1996*, pages 263–288. Xerox PARC, 1996.

[13] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In Meseguer [22], pages 125–147.

[14] M. Clavel and J. Meseguer. Internal strategies in a reflective logic. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, pages 1–12, 1997.

[15] F. Durán and J. Meseguer. An extensible module algebra for maude. In this volume.

[16] J. Goguen and K. Parsaye-Ghomi. Algebraic denotational semantics using parameterized abstract modules. In J. Diaz and I. Ramos, editors, *Formalizing Programming Concepts*, LNCS 107, pages 292–309. Springer-Verlag, 1981.

[17] 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, March 1992. To appear in J. Goguen and G. R. Malcolm, editors, *Applications of Algebraic Specification Using OBJ*, Academic Press, 1998.

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

[19] H. Kirchner and P.-E. Moreau. Prototyping completion with constraints using computational systems. In J. Hsiang, editor, *Proc. Rewriting Techniques and Applications, Kaiserslautern*, LNCS 914, pages 438–443. Springer-Verlag, 1995.

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

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

[22] J. Meseguer, editor. *First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, Asilomar (California), September 1996. Elsevier. `http://www.elsevier.nl/locate/entcs/volume4.html`.

[23] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'96, Pisa, August 1996*, LNCS 1119, pages 331–372. Springer-Verlag, 1996.

[24] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, NATO Advanced Study Institute, Marktoberdorf, Germany, July 29 – August 6, 1997.* Springer-Verlag, 1998.

[25] J. Meseguer and C. Talcott. Using rewriting logic to interoperate architectural description languages (I and II). Lectures at the Santa Fe and Seattle DARPA-EDCS Workshops, March and July 1997. `http://www-formal.stanford.edu/clt/ArpaNsf/adl-interop.html`.

[26] P. Mosses. *Action Semantics.* Cambridge University Press, 1992.

[27] E. Najm and J.-B. Stefani. Computational models for open distributed systems. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-based Distributed Systems, Vol. 2*, pages 157–176. Chapman & Hall, 1997.

[28] S. Nakajima. Encoding mobility in CafeOBJ: an exercise of describing mobile code-based software architecture. In *Proc. of the CafeOBJ Symposium '98, Numazu, Japan.* CafeOBJ Project, April 1998.

[29] J. F. Quesada. Bidirectional and event-driven parsing with multi-virtual trees. *Proceedings of the II International Conference on Mathematical Linguistics*, Tarragona, Spain, May 1996. To appear in C. Martín-Vide, editor, *Mathematical and Computational Models in Linguistics*, John Benjamins.

[30] J. F. Quesada. Overparsing. *Workshop on Mathematical Linguistics*, Pennsylvania State University, State College, April 1998.

[31] J. F. Quesada. *The SCP parsing algorithm based on syntactic constraints propagation.* PhD thesis, University of Seville, 1997.

[32] C. Ringeissen. Prototyping combination of unification algorithms with the ELAN rule-based programming language. In H. Comon, editor, *Proceedings of the 8th Conference on Rewriting Techniques and Applications*, LNCS 1232. Springer-Verlag, 1997.

[33] C. L. Talcott. An actor rewrite theory. In Meseguer [22], pages 360–383.

[34] M. Vittek. *ELAN: Un cadre logique pour le prototypage de langages de programmation avec contraintes.* PhD thesis, Université Henry Poincaré — Nancy I, 1994.

[35] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In Meseguer [22], pages 321–359.