Electronic Notes in Theoretical Computer Science 15 (1998)
URL: http://www.elsevier.nl/locate/entcs/volume15.html 21 pages

Metalevel Computation in Maude [1]

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

[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*

Abstract

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way, and makes possible many advanced metaprogramming applications, including user-definable strategy languages, language extensions by new module composition operations, development of theorem proving tools, and reifications of other languages and logics within rewriting logic. A naive implementation of reflection can be computationally very expensive. We explain the semantic principles and implementation techniques through which efficient ways of performing reflective computations are achieved in Maude through its predefined `META-LEVEL` module.

1 Introduction

Maude's language design and implementation make systematic use of the fact that rewriting logic is reflective [13,14,8]. This makes the metatheory of rewriting logic accessible to the user in a clear and principled way, and makes possible many advanced metaprogramming applications, including user-definable strategy languages, language extensions by new module composition operations, development of theorem proving tools, and reifications of other languages and logics within rewriting logic.

A naive implementation of reflection can be very expensive both in time and memory use. Therefore, a good implementation must provide efficient ways of performing reflective computations. This paper explains how this is

[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.

achieved in Maude through its predefined `META-LEVEL` module. We first discuss the semantics of metalevel computations, and how their efficiency can be dramatically increased by conservatively extending the universal theory $\mathcal{U}$ to a metalevel theory $\mathcal{M}$ with *descent* functions and rules that allow lowering deductions at higher levels of reflection to much more efficient deductions at lower levels. Then, we explain how terms and modules are meta-represented in `META-LEVEL`, and how these semantic principles are supported in important special cases by the `META-LEVEL` module in a built-in way. We also summarize a good number of important metalevel applications and experiments that have already been carried out. However, due to space limitations, and given that many of those applications have already been fully documented in several publications, in this paper we only include simple examples illustrating particular features. The implementation of `META-LEVEL` in Maude's interpreter and a number of useful optimizations are also discussed.

The present version of `META-LEVEL` adds many improvements and new capabilities to the first version demonstrated in the First Rewriting Logic Workshop [12], and also to the more recent beta version [11]; however, we still expect to make some additional improvements and optimizations before the upcoming first release of Maude.

2 Reflection and Metalevel Computation

Rewriting logic is reflective [13,8], that is, there is a finitely presented rewrite theory $\mathcal{U}$ that is *universal* in the sense that we can represent any finitely presented rewrite theory $\mathcal{R}$ (including $\mathcal{U}$ itself) and any terms t, t' in $\mathcal{R}$ as terms $\overline{\mathcal{R}}$ and $\overline{t}, \overline{t'}$ in $\mathcal{U}$, and we then have the following equivalence

$$(\dagger)\ \ \mathcal{R} \vdash t \longrightarrow t' \ \ \Leftrightarrow\ \ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle .$$

Since $\mathcal{U}$ is representable in itself, we can achieve a "reflective tower" with an arbitrary number of levels of reflection, since we have

$$\mathcal{R} \vdash t \to t' \ \Leftrightarrow\ \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \to \langle \overline{\mathcal{R}}, \overline{t'} \rangle \ \Leftrightarrow\ \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t} \rangle} \rangle \to \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \overline{t'} \rangle} \rangle \ldots$$

In this chain of equivalences we say that the first rewriting computation takes place at level 0, the second at level 1, and so on. In a naive implementation, each step up the reflective tower comes at considerable computational cost, because simulating a single step of rewriting at one level involves many rewriting steps one level up. It is therefore important to have systematic ways of lowering the levels of reflective computations as much as possible—so that a rewriting subcomputation happens at a higher level in the tower only when this is strictly necessary.

To achieve a systematic descent into equivalent rewriting computations at lower levels, the key idea is to exploit the equivalence ($\dagger$). A detailed proof of this equivalence has been given for the case of unconditional and unsorted theories [8]. The extension to the case of interest for Maude—namely to conditional rewrite theories with membership equational logic [31,4] as the underlying equational logic—although nontrivial, is essentially unproblematic. We therefore assume a universal theory $\mathcal{U}$ for this more general class of finitely presented rewrite theories. In particular, the signature $\Sigma_{\mathcal{U}}$ of $\mathcal{U}$ has sorts *Term*, *Module*, and *Kind*, whose respective elements $\overline{t} : \mathit{Term}$, $\overline{\mathcal{R}} : \mathit{Module}$, and $\overline{K} : \mathit{Kind}$ represent terms, rewrite theories, and kinds[2] in a signature, respectively. We assume that there is also an equationally defined Boolean predicate $\mathit{parse} : \mathit{Module} \times \mathit{Kind} \times \mathit{Term} \longrightarrow \mathit{Bool}$ so that $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{true}$ if t is an $\mathcal{R}$-term of kind K, and $\mathit{parse}(\overline{\mathcal{R}}, \overline{K}, \overline{t}) = \mathit{false}$ otherwise.

We can exploit the equivalence (†) by introducing the notion of *descent function*, that is, a function that, given metalevel representations for a rewrite theory $\mathcal{R}$ and a term t in it, rewrites such a term in $\mathcal{R}$ according to a given strategy and returns the meta-representation of the resulting term. Such functions can be simply expressed in terms of a general *sequential interpreter function* I for rewriting logic. This is a *partial* function that takes three arguments: a finitely presentable rewrite theory $\mathcal{R}$, a term t, and a deterministic strategy S. In case of termination it returns either the term t' to which t was rewritten according to S, or an error message that is not a term in $\mathcal{R}$. The function is undefined in case the strategy does not terminate. For any finitely presentable rewrite theory $\mathcal{R}$, terms t, t' in it, and admissible deterministic strategy S, any such interpreter function must of course satisfy the correctness requirement

$$(\flat)\quad I(\mathcal{R}, t, S) = t' \quad\Rightarrow\quad \mathcal{R} \vdash t \longrightarrow t'.$$

The point is that, regardless of the particular details of I, we can always equationally axiomatize any such effective interpreter function by means of a Church-Rosser, but in general non-terminating, finitary equational theory $\mathcal{I}$. This can be done in a signature that we can assume contains $\Sigma_{\mathcal{U}}$ as a subsignature. By extending our universal theory $\mathcal{U}$ with the new sorts, operations, and equations of $\mathcal{I}$, we obtain an extended rewrite theory $\mathcal{U} \cup \mathcal{I}$. A *descent function* is then a function $d : \mathit{Module} \times \mathit{Term} \times \mathit{Parameters} \longrightarrow \mathit{Term}$ such that there is a deterministic strategy expression S_d with a single free variable of sort *Parameters* satisfying the equality

$$d(\overline{\mathcal{R}}, \overline{t}, p) = \overline{I(\mathcal{R}, t, S_d(p))}.$$

Such descent functions are of course easily definable equationally as definitional extensions of the theory $\mathcal{U} \cup \mathcal{I}$. Note that, since we have only added some new equations, the only rewrite rules in $\mathcal{U} \cup \mathcal{I}$ are exactly those in $\mathcal{U}$. But, given a descent function d, we can now exploit the equivalence (†) by adding to $\mathcal{U} \cup \mathcal{I}$ a *descent rule*

$$d : \langle M, x\rangle \longrightarrow \langle M, y\rangle$$
$$\mathit{if}\ \mathit{parse}(M, K, x) = \mathit{true} \wedge \mathit{parse}(M, K, y) = \mathit{true} \wedge d(M, x, p) = y.$$

[2] In a membership equational logic signature, terms always have a *kind*; they may or may not have a *sort* of that kind. In the Maude system kinds are represented as *error sorts*, that are added by the system at the top of each connected component of sorts defined by the user.

where M : *Module*, x, y : *Term*, K : *Kind*, and p : *Parameters*. The equivalence (†) can be exploited for efficiency reasons with such a rule, because the sequential interpreter I can be a built-in function such as the Maude interpreter; therefore, instantiating M with $\overline{\mathcal{R}}$, we can use efficient deduction in $\mathcal{R}$ to perform deduction in $\mathcal{U}$. Let $\mathcal{M}$ denote a rewrite theory of the form $\mathcal{M} = \mathcal{U} \cup \mathcal{I} \cup \mathcal{D}$, where $\mathcal{D}$ is the addition of several descent functions and of their associated descent rules. We shall call $\mathcal{M}$ a *metalevel theory*.

The addition of descent rules to $\mathcal{U}$ is of course *conservative*, in the sense of not adding any rewrites that could not be performed, albeit less efficiently, in $\mathcal{U}$ itself, since for any descent rule d we have [3],

$$\begin{aligned} &\mathcal{M} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \xrightarrow{d} \langle \overline{\mathcal{R}}, \overline{t'} \rangle \;\;\Rightarrow\;\; I(\mathcal{R}, t, S_d(p)) = t' \\ &\overset{\flat}{\Rightarrow}\;\; \mathcal{R} \vdash t \longrightarrow t' \;\;\overset{\dagger}{\Leftrightarrow}\;\; \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \overline{t} \rangle \longrightarrow \langle \overline{\mathcal{R}}, \overline{t'} \rangle. \end{aligned}$$

Note that, by applying several descent functions, we can descend several levels in the reflective tower. Assume that $\mathcal{M}$ includes descent functions d and d', and let $\mathcal{R}$ and t be an arbitrary rewrite theory and a term in it; then we have, for example,

$$\begin{aligned} &d(\overline{\mathcal{M}}, \overline{d'(\overline{\mathcal{R}}, \overline{t}, p')}, p) = \overline{I(\mathcal{M}, d'(\overline{\mathcal{R}}, \overline{t}, p'), S_d(p))} = \\ &= \overline{I(\mathcal{M}, \overline{I(\mathcal{R}, t, S_{d'}(p'))}, S_d(p))}. \end{aligned}$$

That is, a meta-metalevel computation can be efficiently carried out at the object level. An example of this kind of combined descent is given in Section 3.3. More generally, we should view descent functions as *basic strategies*, that can be used as fundamental building blocks to define *internal strategy languages* [15,8], in which they can be combined with each other and with more complex strategies at several levels of reflection to perform efficiently sophisticated metalevel computations.

3 Maude's Metalevel

In Maude, key functionality of a metalevel theory $\mathcal{M}$ with several descent functions has been efficiently implemented in a functional module `META-LEVEL`, by using as the interpreter function I Maude's own interpreter. Furthermore, several other useful functions of the universal theory $\mathcal{U}$ are also built-in for efficiency reasons. What follows is a snapshot of our latest version of `META-LEVEL`, which offers a number of significant improvements—such as mix-fix syntax, and allowing ad-hoc overloading in the signatures of modules—as well as new

[3] Of course, to ensure conservativity we also should assume that the new equations in $\mathcal{M}$ do not disturb the equality of terms or the rewriting relation in $\mathcal{U}$. Since the equations are assumed to be Church-Rosser, conservativity of equality can be easily achieved by assuming that the tops of lefthand sides of new equations are new function symbols. Preservation of the rewriting relation can be achieved by forbidding nonvariable overlaps between lefthand sides of new equations and of rules, as done in [8].

functionality over the beta version [11]. We expect some syntax details to change, and some more new functionality to be added before the upcoming release of Maude.

We summarize below the key functionality provided by META-LEVEL. We recall that Maude's *functional modules* are equational theories that are assumed to be Church-Rosser and terminating modulo some axioms for which matching algorithms are available in the implementation, and that *system modules* are rewrite theories whose equational part satisfies the same requirements as a functional module, and where the equations and the rules are assumed to be *weakly coherent* [34] modulo the axioms. In META-LEVEL:

- Maude terms are reified as elements of a data type Term of terms;
- Maude modules are 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 descent function meta-reduce;
- the process of applying a rule of a system module to a subject term is reified by a descent function meta-apply;
- the process of rewriting a term in a system module using Maude's default interpreter is reified by a descent 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.

As all other Maude modules, the module META-LEVEL imports the built-in module BOOL as a submodule. In addition, it also imports the predefined modules MACHINE-INT and QID of machine integers and of quoted identifiers, all with the expected syntax. We first introduce the syntax used in META-LEVEL for representing terms; then we explain how modules are represented; and finally we discuss the different built-in functions, namely, the descent functions, and the parsing, pretty printing, and sort functions.

3.1 Representing Terms

Terms are reified as elements of the data type Term of terms, with the following signature

```
subsort Qid < Term .
subsort Term < TermList .
op {_}_ : Qid Qid -> Term .
op _[_] : Qid TermList -> Term .
op _,_ : TermList TermList -> TermList [assoc] .
op _:_ : Term Qid -> Term .
op _::_ : Term Qid -> Term .
op error* : -> Term .
```

The first declaration, making Qid a subsort of Term is used to represent variables by the corresponding quoted identifiers. Thus, the variable `N` is represented by `'N`. The operator `{_}_` is used for representing constants essentially as pairs, with the first argument the constant, in quoted form, and the second argument the sort of the constant, also in quoted form. For example, the constant `0` in the module `NAT` in Section 3.2 is represented as `{'0}'Nat`. The operator `_[_]` corresponds to the recursive construction of terms out of subterms, with the first argument the top operator in quoted form, and the second argument the list of its subterms, where list concatenation is denoted `_,_`. For example, the term `s(s(0)) + s(0)` of sort `Nat` in module `NAT` is meta-represented as

```
'_+_['s['s[{'0}'Nat]], 's[{'0}'Nat]].
```

As already mentioned when discussing the universal theory, since terms in `META-LEVEL` can be meta-represented just as terms in any other module, the representation of terms can be iterated. For example, the meta-meta-representation $\overline{\overline{\texttt{s(0)}}}$ of the term `s(0)` in `NAT` is the term

```
'_[_][{''s}'Qid,'{_}_[{''0}'Qid,{''Nat}'Qid]].
```

For convenience, the syntax of Maude modules is automatically extended with an `if_then_else_fi` operator, and with Boolean-valued predicates for equality and inequality, and for membership of a term in a sort. Since we assume that the membership equational logic theories underlying functional and system modules are confluent, terminating, and sort-decreasing, all these predicates are *decidable* [3]. By a general meta-result of Bergstra and Tucker [1], we could extend our original specification to another in which all the above predicates are explicitly defined equationally; but it is much easier and efficient to provide them in a built-in way. For the most part, the meta-representation of terms involving such built-in operators proceeds as for any other term. For example, the application of the equality predicate `==` in the term `s(s(0)) == s(0)` is represented as the term

```
'_==_['s['s[{'0}'Nat]], 's[{'0}'Nat]].
```

But since we can think of membership predicates $t : s$ not as unary predicates, one for each sort s, but as a binary predicate with the second argument varying over sorts, it is natural to meta-represent them in a uniform way by means of a binary constructor `_:_` with the first argument the representation of the term, and the second the representation of the sort as a quoted identifier. For example, the membership predicate `s(0):Nat` is represented as the term

```
's[{'0}'Nat]:'Nat.
```

Similarly, there is also a binary constructor `_::_` for meta-representing the "lazy" membership predicate that does not evaluate the term in question at all, but uses only the syntactic declarations in the module's order-sorted signature (that is assumed preregular [21]) to decide whether the least sort of the term is smaller or equal to a given sort. The last declaration for the data type of terms is a constant `error*` to be used as an error element.

3.2 Representing Modules

Functional and system modules are meta-represented in a syntax very similar to their original user syntax. The main differences are that: (1) terms in equations, membership axioms, and rules are now meta-represented as we have already explained; (2) in the meta-representation of modules we follow a *fixed order* in introducing the different kinds of declarations for sorts, subsorts, variables, equations, etc., whereas in the user syntax there is considerable flexibility for introducing such different declarations in an interleaved and piecemeal way; and (3) sets of identifiers—used in declarations of sorts—are represented as sets of quoted identifiers built with an associative and commutative operator _;_.

To motivate the general syntax for representing modules, we illustrate it with a simple example—namely, a module `NAT` for natural numbers with zero and successor and with commutative addition and multiplication operators.

```
fmod NAT is
  sorts Zero Nat .
  subsort Zero < Nat .
  op 0 : -> Zero .
  op s : Nat -> Nat .
  op _+_ : Nat Nat -> Nat [comm] .
  op _*_ : Nat Nat -> Nat [comm] .
  vars N M : Nat .
  eq 0 + N = N .
  eq s(N) + M = s(N + M) .
  eq 0 * N = 0 .
  eq s(N) * M = M + (N * M) .
endfm
```

The syntax for the top-level operator representing functional modules, and the corresponding syntax for representing system modules are as follows

```
op fmod_is_______endfm : Qid ImportList SortDecl
      SubsortDeclSet OpDeclSet
      VarDeclSet MembAxSet EquationSet -> Module .

op mod_is________endm : Qid ImportList SortDecl
      SubsortDeclSet OpDeclSet
      VarDeclSet MembAxSet EquationSet RuleSet -> Module .
```

The representation $\overline{\texttt{NAT}}$ of `NAT` in `META-LEVEL` is the term

```
fmod 'NAT is
  nil
  sorts 'Zero ; 'Nat .
  subsort 'Zero < 'Nat .
  op '0 : nil -> 'Zero [none] .
```

```
    op 's : 'Nat -> 'Nat [none] .
    op '_+_ : 'Nat 'Nat -> 'Nat [comm] .
    op '_*_ : 'Nat 'Nat -> 'Nat [comm] .
    var 'N : 'Nat .
    var 'M : 'Nat .
    none
    eq '_+_[{'0}'Nat, 'N] = 'N .
    eq '_+_['s['N], 'M] = 's['_+_['N, 'M]] .
    eq '_+_[{'0}'Nat, 'N] = {'0}'Nat .
    eq '_+_['s['N], 'M] = '_+_['N, '_*_['N, 'M]] .
  endfm
```

of sort `Module`. Since `NAT` has no list of imported submodules[4] and no membership axioms, those fields are filled by the `nil ImportList`, and the `none MembAxSet`. Similarly, since the zero and successor operators have no attributes, they have the `none` set of attributes.

Part of the syntax for functional and system modules is listed below. It extends in a natural way the fragment illustrated in the above example and should, for the most part, be self-explanatory for someone familiar with the Maude syntax, which is mirrored quite closely. Note that we have to represent the set of attributes of an operator. For this, sorts `Attr` and `AttrSet` are used. Such attributes may be equational axioms to rewrite modulo, syntactic attributes for parsing purposes, and so on. In particular, some operators of predefined modules such as `MACHINE-INT`, `QID`, and `META-LEVEL` are computed in a built-in way by associating them to special C++ functions in the underlying implementation. Such built-in operators are declared with the `special` attribute, that takes a `HookList` of bindings to actual code as argument.

```
  sorts Module Import ImportList QidList MachineIntList
    QidSet Sort SortDecl SubsortDecl SubsortDeclSet Attr AttrSet
    OpDecl OpDeclSet VarDecl VarDeclSet Term TermList Equation
    EquationSet Rule RuleSet MembAx MembAxSet Hook HookList .
  subsort Import < ImportList .
  subsort Qid < QidList .
  subsort Qid < QidSet .
  subsort Qid < Sort .
  subsort MachineInt < MachineIntList .
  subsort SubsortDecl < SubsortDeclSet .
  subsort Attr < AttrSet .
  subsort OpDecl < OpDeclSet .
  subsort VarDecl < VarDeclSet .
  subsort Equation < EquationSet .
  subsort Rule < RuleSet .
  subsort MembAx < MembAxSet .
  subsort Hook < HookList .
```

[4] All terms of sort `Module` are supposed to be completely *flattened*, so that they contain no submodules. Therefore, their `ImportList` must always be `nil`.

```
op nil : -> QidList .
op __ : QidList QidList -> QidList [assoc id: nil] .
op none : -> QidSet .
op _;_ : QidSet QidSet -> QidSet [assoc comm id: none] .
op nil : -> MachineIntList .
op __ : MachineIntList MachineIntList -> MachineIntList
                                       [assoc id: nil] .

op nil : -> ImportList .

op sorts_. : QidSet -> SortDecl .
op subsort_<_. : Qid Qid -> SubsortDecl .
op none : -> SubsortDeclSet .
op __ : SubsortDeclSet SubsortDeclSet -> SubsortDeclSet
                                [assoc comm id: none] .

op (op_:_->_[_].) : Qid QidList Qid AttrSet -> OpDecl .
op none : -> OpDeclSet .
op __ : OpDeclSet OpDeclSet -> OpDeclSet [assoc comm id: none] .

op none : -> AttrSet .
op __ : AttrSet AttrSet -> AttrSet [assoc comm id: none] .
ops assoc comm idem : -> Attr .
ops id left-id right-id : Term -> Attr .
op strat : MachineIntList -> Attr .
op prec : MachineInt -> Attr .
op gather : QidList -> Attr .
op special : HookList -> Attr .

op var_:_. : Qid Qid -> VarDecl .
op none : -> VarDeclSet .
op __ : VarDeclSet VarDeclSet -> VarDeclSet [assoc comm id: none] .

op mb_:_. : Term Qid -> MembAx .
op cmb_:_if_=_. : Term Qid Term Term -> MembAx .
op none : -> MembAxSet .
op __ : MembAxSet MembAxSet -> MembAxSet [assoc comm id: none] .

op eq_=_. : Term Term -> Equation .
op ceq_=_if_=_. : Term Term Term Term -> Equation .
op none : -> EquationSet .
op __ : EquationSet EquationSet -> EquationSet
                                 [assoc comm id: none] .

op rl[_]:_=>_. : Qid Term Term -> Rule .
op crl[_]:_=>_if_=_. : Qid Term Term Term Term -> Rule .
```

```
op none : -> RuleSet .
op __ : RuleSet RuleSet -> RuleSet [assoc comm id: none] .
```

Note that—just as in the case of terms—terms of sort `Module` can be meta-represented again, yielding then a term of sort `Term`, and this can be iterated an arbitrary number of times. This is in fact necessary when a metalevel computation has to operate at higher levels. A good example is the inductive theorem prover described in [10], where modules are meta-represented as terms in the inference rules for induction, but they have to be meta-meta-represented as terms of sort `Term` when used in strategies that control the application of the inductive inference rules. We illustrate the meta-meta-representation of modules with a simple example, namely a module `TRUTH` of truth values

```
fmod TRUTH is
  sorts Truth .
  ops t f : -> Truth .
endfm
```

whose meta-meta-representation $\overline{\overline{\texttt{TRUTH}}}$ is the following term of sort `Term`

```
'fmod_is_______endfm[
   {''TRUTH}'Qid,
   {'nil}'ImportList,
   'sorts_.[{''Truth}'Qid],
   {'none}'SubsortDeclSet,
   '__['op_:_->_`[_`].[{''t}'Qid,{'nil}'QidList,
                                   {''Truth}'Qid,{'none}'AttrSet],
       'op_:_->_`[_`].[{''f}'Qid,{'nil}'QidList,
                                   {''Truth}'Qid,{'none}'AttrSet]],
   {'none}'VarDeclSet,
   {'none}'MembAxSet,
   {'none}'EquationSet]
```

3.3 Descent Functions

`META-LEVEL` has three built-in descent functions, `meta-reduce`, `meta-rewrite`, and `meta-apply`, that provide three useful and efficient ways of reducing metalevel computations to object-level ones.

The operation `meta-reduce` takes as arguments the representations of a module $\mathcal{R}$, and of a term t or a membership predicate $t : s$, or lazy membership predicate $t :: s$, in that module. It has syntax

```
op meta-reduce : Module Term -> Term [special ... ] .
```

where we have omitted the lengthy list of bindings to C++ code in its declaration as a `special` built-in operator. When the second argument is the representation $\overline{t}$ of a term t in $\mathcal{R}$, `meta-reduce` returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, e.g,

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{NAT}}\texttt{, }\overline{\texttt{s(0) + 0}}\texttt{) .}$$
$$\texttt{result Term: }\overline{\texttt{s(0)}}$$

Similarly, when the second argument of `meta-reduce` is the representation of a membership predicate $t : s$ (or a lazy membership predicate $t :: s$) the term t is fully reduced using the equations in $\mathcal{R}$ (respectively, the least sort of the term t in the signature of the module $\mathcal{R}$ is computed) and then the representation of the Boolean value of the corresponding predicate is returned.

The corresponding interpreter function for $\texttt{meta-reduce}(\overline{\mathcal{R}}, \overline{t})$ rewrites the term t to normal form using only the equations in $\mathcal{R}$, and does so according to the evaluation strategies [11,19] declared for each operator in the signature of $\mathcal{R}$—which by default is bottom-up for operators with no such strategies declared. In other words, the interpreter strategy for this function coincides with that of the `red` command in Maude, that is,

$$\texttt{meta-reduce}(\overline{\mathcal{R}}, \overline{t}) = \overline{I_{Maude}(\mathcal{R}, t, \texttt{red})}.$$

The operation `meta-rewrite` has syntax

```
op meta-rewrite : Module Term MachineInt -> Term [special ... ] .
```

It is entirely analogous to `meta-reduce`, but instead of using only the equational part of a module it now uses both the equations and the rules to rewrite the term using Maude's default strategy. Its first two arguments are the representations of a module $\mathcal{R}$ and of a term t, and its third argument is a positive machine integer n. Its result is the representation of the term obtained from t after at most n applications of the rules in $\mathcal{R}$ using the strategy of Maude's default interpreter, which applies the rules in a fair, top-down fashion. When the value 0 is given as the third argument, no bound is given to the number of rewrites, and rewriting proceeds to the bitter end. Again, `meta-rewrite` is a paradigmatic example of a descent function; its corresponding interpreter strategy is that of the `rewrite` command in Maude [11], that is,

$$\texttt{meta-rewrite}(\overline{\mathcal{R}}, \overline{t}, n) = \overline{I_{Maude}(\mathcal{R}, t, \texttt{rewrite}\ [n])}.$$

The operation `meta-apply` has syntax:

```
op meta-apply : Module Term Qid Substitution
                MachineInt -> ResultPair [special ... ] .
```

The first four arguments are representations in `META-LEVEL` of a module $\mathcal{R}$, a term t in $\mathcal{R}$, a label l of some rules in $\mathcal{R}$, and a set of assignments (possibly empty) defining a partial substitution σ for the variables in those rules. The last argument is a natural number n. `meta-apply` then returns a pair of sort `ResultPair` consisting of a term and a substitution. The syntax for substitutions and for results is

```
sorts Assignment Substitution ResultPair .
subsort Assignment < Substitution .

op _<-_ : Qid Term -> Assignment .
op none : -> Substitution .
op _;_ : Substitution Substitution -> Substitution
                                     [assoc comm id: none] .
op _,_ : Term Substitution -> ResultPair .
```

The operation `meta-apply` is evaluated as follows:

(i) the term t is first fully reduced using the equations in $\mathcal{R}$;

(ii) the resulting term is matched against all rules with label l partially instantiated with σ, with matches that fail to satisfy the condition of their rule discarded;

(iii) the first n successful matches are discarded; if there is an $(n+1)$th match, its rule is applied using that match and the steps 4 and 5 below are taken; otherwise {`error*, emptySubstitution`} is returned;

(iv) the term resulting from applying the given rule with the $(n+1)$th match is fully reduced using the equations in $\mathcal{R}$;

(v) the pair formed using the constructor $\{\_,\_\}$ whose first element is the representation of the resulting fully reduced term and whose second element is the representation of the match used in the reduction is returned.

The interpreter strategy associated to $\texttt{meta-apply}(\overline{\mathcal{R}}, \overline{t}, \overline{l}, \overline{\sigma}, n)$ is not that of a user-level command in the Maude interpreter. It is instead a built-in strategy internal to the interpreter that attempts one rewrite at the top as explained above.

We finish this section with an example illustrating the efficient support for a reflective tower in `META-LEVEL`, plus the fact that in the same computation we can descend to different theories at different levels. We meta-reduce the pair consisting of the meta-representation of `META-LEVEL` and the meta-representation of the term meta-reducing an expression in `NAT`, to get the meta-meta-representation of the result,

$$\texttt{Maude> red meta-reduce(}\overline{\texttt{META-LEVEL}}\texttt{, }\overline{\texttt{meta-reduce(}\overline{\texttt{NAT}}\texttt{,}\overline{\texttt{s(0) + 0}}\texttt{)}}\texttt{)}\ .$$
$$\texttt{result Term: }\ \overline{\overline{\overline{\texttt{s(0)}}}}$$

This capacity for supporting a reflective tower is used crucially in applications such as inductive theorem proving [10] that involve computations at levels 0–2.

3.4 Parsing, Pretty Printing, and Sort Functions

Besides the descent functions already discussed, `META-LEVEL` provides several other functions that naturally belong to the universal theory and could have been equationally axiomatized in such a theory. However, for efficiency reasons they are provided as built-in functions. These functions allow parsing and pretty printing a term in a module at the metalevel (see [9] for more discussion), and performing efficiently a number of useful operations on the sorts declared in a module's signature.

The function `meta-parse` has syntax

```
op meta-parse : Module QidList -> Term [special ... ] .
```

It takes as arguments the representation of a module and the representation of a list of tokens as a list of quoted identifiers. It returns the metarepresentation of the parsed term of that list of tokens for the signature of the module, which is assumed to be unambiguous. If such a parsed term does not exist, the error constant `error*` is returned instead.

The function `meta-pretty-print` (currently under implementation) has syntax

```
op meta-pretty-print : Module Term -> QidList [special ... ] .
```

It takes as arguments the representation of a module M and the representation of a term t. It returns a list of quoted identifiers that encode the string of tokens produced by pretty printing t in the syntax given by M. In the event of an error an empty list is returned.

The operations on sorts include `sortLeq`, `leastSort`, `lesserSorts` and `glbSorts`. They provide commonly needed functions on the poset of sorts of a module in a built-in way at the metalevel. They have been explained in detail in [11].

4 Metalevel Applications

Maude's metalevel is an extremely useful language feature. It has been used in a series of experiments demonstrating its expressive capabilities in four main areas: internal strategy languages, metaprogramming, the reification of languages and logics, and the development of theorem proving tools.

4.1 Internal Strategy Languages

One can easily use Maude's metalevel to program execution strategies with rewrite rules. In this way, strategies become internalized within rewriting logic—so that they have a logical semantics and can be reasoned about as other rewrite theories—and strategy languages become easily extensible [14,15,8]. As a very simple example, one can encode the most naive description of sorting as a single rewrite rule as follows:

```
sort([X,E1,Y,E2,Z]) =>  sort([X,E2,Y,E1,Z]) if E2 < E1 .
```

Then, various sorting algorithms can be encoded at the metalevel in very straightforward ways. In particular, bubblesort can be captured by searching for matches of the above rewrite rule where `Y` is empty. Other sorting algorithms can be encoded with slightly more elaborate strategies [8]. Internal strategies can be used to specify algorithms, to control the execution of interpreters and theorem provers, to encode winning strategies for games, and for many other purposes [15,8,10]. The importance of strategies has also been stressed in the ELAN language [23,2].

Strategies can also be used to emulate a model checker [28] or more limited finite-state exploration tools [17] in the analysis of finite- or infinite-state systems. In particular, systems such as secure communication protocols can be specified as object-oriented modules in Maude and can be subjected to formal analysis through strategies. Nondeterministic behavior such as the order of messages arrival in such protocols can be controlled at the metalevel so that all possible behaviors of a system are explored. Denker, Meseguer and Talcott [16] have specified the Needham-Schroeder cryptographic protocol plus an attacker object, and have put in evidence a previously-known attack. Similarly, Ölveczky, Denker, García-Luna, Meseguer, Smith and Talcott have found a number of flaws and deadlocks in the informal specification of a reliable broadcast protocol (see Appendix B in [11]). Strategies of this kind are also crucial for the rewriting execution of tile logic concurrency models, as shown in the work of Bruni, Meseguer, and Montanari [5].

4.2 Metaprogramming

"Metaprogramming" is programming at the metalevel, or in other words, writing programs that manipulate other programs as data. In the case of Maude, the metalevel equations and rewrite rules operate on representations of lower-level rewrite theories. The idea is not new, but its realization in Maude has a clear logical semantics by defining such operations by means of rewrite theories that extend the universal theory of rewriting logic with new data types and new functions. In this way, one can reason formally about the correctness of metaprogramming constructions by analyzing their associated rewrite theories. In particular, Maude's parameterized programming style in the Clear and OBJ tradition [7,22] becomes internalized in the logic as a very useful metaprogramming style that can now be *extended* with new modes of parameterization of theories, new methods of program composition, and new ways of defining views of other program modules [18]. Thus, Maude opens the meta-reasoning of the programming language to the user, allowing programmer- and program-controlled changes to basic language features such as the module system and the evaluation order, as we have seen above for strategies. For example, an interesting collection of user-definable parameterized constructions of this kind has been used in the design of the Cafe theorem proving tools to perform a variety of useful module transformations [10], and both OBJ-like module operations and the transformation from object-oriented modules to system modules have been implemented in Maude's metalevel [18]. As explained below, one can even use Maude's metaprogramming to define translations between languages and logics different from Maude and rewriting logic [32].

4.3 Reifying Languages and Logics

Rewriting logic has great potential as a logical framework that can represent many different languages and logics. By reflection, such representation maps can be *reified* and executed within rewriting logic. The key idea is to define a data type $\texttt{Module}_{\mathcal{L}}$ for the modules of the language or logic $\mathcal{L}$ in question, and then reify the representation map at the metalevel as an equationally-defined function mapping terms in $\texttt{Module}_{\mathcal{L}}$ to terms in `Module`. In a similar way, one can reify translations between two different languages or logics $\mathcal{L}$ and $\mathcal{L}'$ as functions from $\texttt{Module}_{\mathcal{L}}$ to $\texttt{Module}_{\mathcal{L}'}$. Rewriting logic definitions of several languages have been given by different authors: lambda calculus and mini-ML [24,25], Prolog and narrowing languages like BABEL [36], CCS (see [33,25], the work of Bruni and Clavel [8,6], and the tile rewrite rules in [20]), the π-calculus [35], and sketches of UNITY [29] and Gamma [30]. Similarly, many different logics have been shown to have a very natural conservative representation in rewriting logic [25]. The point is that all such representation maps can be reified using the reflective method described above. For the case of the representation map from linear logic into rewriting logic this was defined in [26] and has been implemented in Maude by Clavel and Martí-Oliet [8]. Finally, rewriting logic may also be used to reason about the models of computation expressed in different languages [8,6,27], providing a semantic foundation for building connections between diverse models of computation.

4.4 Developing Theorem Proving Tools

A particularly useful area of metalevel applications is representing in Maude an inference system for a logic of interest, to get in this way a theorem prover for it. In our experience this makes developing a theorem prover almost as easy as writing down its inference system in a scientific paper. Furthermore, tools designed this way are very easily modifiable and extensible, and, thanks to Maude's high performance implementation, they can often compete in efficiency with tools developed by conventional means. Reflection plays here a crucial role for two reasons. Firstly, the inference system itself may perform *theory transformations*, so that the theories themselves must be treated as data; therefore, the rules of our inference system may already be at the metalevel to begin with. Secondly, we need *strategies* to guide the application of the inference rules. Therefore, if the inference rules themselves are already at the metalevel, the strategies are then at the meta-meta-level. For tool development, this distinction of levels is of great practical importance, because it allows a very modular modification of the proof tactics with no change whatsoever to the inference system. In conventional implementations both aspects are sometimes so intertwined that modifications can be very difficult. As part of the CafeOBJ project, we have developed two theorem proving tools in Maude using these methods, namely an inductive theorem prover for membership equational logic, and a Church-Rosser checker for order-sorted equational specifications whose function symbols can be commutative [10].

5 Implementation

The metalevel is implemented outside of the Maude interpreter's core rewriting engine but makes heavy use of its extensible design. We discuss below the implementation of two key built in operations, `meta-reduce` and `meta-apply`, and the optimizations currently implemented as well as those planned for the future.

5.1 `meta-reduce`

An operation f inside the Maude interpreter is represented by an object which belongs to a C++ class which is ultimately derived from the abstract base class *Symbol*. This base class contains a large number of virtual functions including one called *eqRewrite()* which is responsible for equational rewriting on terms headed by f. For user-defined operations f this function is defined in some class derived from *Symbol* that is specific to the theory to which f belongs—such as *FreeSymbol* for the free theory. For built-in operations such as `meta-reduce`, however, this function is defined in some class derived from a theory-specific class and is typically outside the rewrite engine proper. For `meta-reduce` this function performs the following steps (we make the simplifying assumption that no errors occur):

(i) Call the rewrite engine to reduce both arguments by the equations in the extension of `META-LEVEL` defined by the user for the given application (for example, a strategy language).

(ii) Parse the first argument, creating a decoded internal representation of the meta-encoded module just as if the module had been typed in. In particular this means that for subterms which are quoted identifiers, the identifiers without the quotes are entered into and looked up in the symbol table.

(iii) Call the rewrite engine to generate a set of matching and replacement automata for the module.

(iv) Parse the second argument, creating a decoded internal representation of the meta-encoded term.

(v) Generate the optimal term-graph representation G of the term.

(vi) Call the rewrite engine to reduce G with the module's matching and replacement automata, resulting in a term-graph H.

(vii) Create a new term-graph H' which meta-encodes H.

(viii) Overwrite the `meta-reduce` node with the top node of H'.

Note that steps 2 and 4 correspond to reversing the meta-representation process to get $\mathcal{R}$ and t from $\overline{\mathcal{R}}$ and $\overline{t}$, whereas step 7 corresponds to meta-representing the result. Steps 1 and 6 correspond, respectively, to equational rewriting in the given extension of `META-LEVEL` and in the object theory. Since they are computations for which the rewrite engine was optimized, they are relatively cheap. Step 5 is moderately expensive, but it is considered necessary in order to share any common subexpressions. Step 8 is trivial.

Step 7 is a term-graph to term-graph transformation that preserves sharing; i.e., if the source term graph is `f(g(a),g(a))`, with `g(a)` shared, then the result term graph will be `'f['g[{'a}'S],'g[{'a}'S]]`, with `'g[{'a}'S]` shared. This is achieved by keeping a hash table of pointers to nodes in the source term graph. The main expense turns out to be walking the source term graph since this must be done using *iterators*. (This is because each theory may have its own private representation for term-graph nodes containing its operators.)

Steps 2, 3 and 4 are the really expensive parts. However in the common case in which the user wishes to do many reductions in the same module, steps 2 and 3 can be dramatically sped up by caching recently compiled meta-modules along with their meta-representation. Since meta-representations of modules are big term-graphs, it might seem that comparing the reduced first argument of `meta-reduce` to the meta-representations of the cached modules would be expensive. However this can usually be avoided, since when the meta-representations are passed around by binding them to variables in equations the machine address of nodes in the meta-representation does not change so we first try comparing pointers and only try comparing the meta-representations themselves if that fails.

5.2 `meta-apply`

The overall implementation of `meta-apply` is along the same lines as that of `meta-reduce`. Substitutions are moved down and up using a slight entension to the code that moves terms down and up for `meta-reduce`. The major extra task is finding the $(n+1)$th successful match for the reduced form of the term meta-encoded by the second argument against all rules having the label given by the third argument. The rules in any module—unlike equations and membership axioms that can only be applied in a rigidly controlled way—are compiled for maximum flexibility at the cost of performance. In particular they always use the most general matching algorithms, which accept partial substitutions and produce *match objects* that encode all matching substitutions. The current implementation simply goes through the rules in text order. For each rule, it calls the matcher to generate the match object, and then extracts matches one by one and checks if they satisfy the condition (if any) of the rule. The first n successful matches are discarded.

We new sketch some planned future optimizations. We imagine that repeated calls to `meta-apply` will generally follow one of two distinct patterns. In the breadth-first pattern the user is essentially looking for all one-step rewrites available from a given term. In this case consecutive calls to `meta-apply` will usually differ only in their fifth argument. Here we could gain a big performance improvement by caching all the arguments together with the last match object and the index of the rule used. Typically the user will increment the last argument starting from 0 by 1 each time (since the user has no direct control over the order of generation of solutions—this depends on the matching algorithms). Thus we would often be able to satisfy a `meta-apply` call by fetching the appropriate match object from cache without having to move any meta-encoded argument (module, term or substitution) down or redo the reduction, matching and discarding steps.

In the depth-first pattern the user is pursuing one chain of rewrites to some conclusion. In this case consecutive calls to `meta-apply` will often have the property that the second argument of one `meta-apply` call will be identical to the term part of the result of the previous `meta-apply` call. In this case we can improve performance by caching the result term-graph along with its meta-encoding. Then in a subsequent `meta-apply` call if the second argument is equal to the cached meta-encoding we can save the cost of parsing the second argument, generating its optimal term graph representation and reducing it. These last two caching optimizations have not yet been implemented.

6 Concluding Remarks

We have explained the semantic principles on which Maude's metalevel and its efficient implementation are based. We have also explained in detail how terms and modules are meta-represented, and the built-in functions provided by `META-LEVEL`. And we have given a brief overview of useful metalevel applications that have already been carried out.

The results and experience obtained so far are encouraging. They suggest a number of future developments that we plan to pursue in the near future. Firstly, the `META-LEVEL` module should be further optimized to make metalevel computations even more efficient; and should be extended with new built-in functions to increase its usability and range of applications. Secondly, with the upcoming Maude release we expect that many new metalevel applications will be developed by different researchers. This will both heighten the importance of these ideas and will provide fresh practical stimulus for their further development. Finally, the area of formal metalevel reasoning opened up by the logical reflective semantics of Maude's metalevel should be systematically explored.

Acknowledgement

We are indebted to José F. Quesada for his excellent work on the MSCP context-free parser for Maude, that—besides being used for different parsing functions in Maude—is used as a key component of the built-in function `meta-parse` in `META-LEVEL`. We cordially thank Carolyn Talcott for many discussions on metalevel issues that have contributed to the development of our ideas.

References

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

[2] P. Borovanský, C. Kirchner, and H. Kirchner. Strategies and rewriting in ELAN. In B. Gramlich and H. Kirchner, editors, *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction (Townsville, Australia, July 1997)*, 1997.

[3] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. Manuscript, SRI International, November 1997, submitted for publication.

[4] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97*, volume 1214 of *Lecture Notes in Computer Science*, pages 67–92. Springer-Verlag, 1997.

[5] R. Bruni, J. Meseguer, and U. Montanari. Internal strategies in a rewriting implementation of tile systems. This volume.

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

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

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

[9] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Marí-Oliet, J. Meseguer, and J. Quesada. Maude as a metalanguage. This volume.

[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, F. Durán, S. Eker, J. Meseguer, and P. Lincoln. An introduction to Maude (beta version). Manuscript, SRI International, March 1998.

[12] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. 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`.

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

[14] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. 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`.

[15] 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.

[16] G. Denker, J. Meseguer, and C. Talcott. Protocol Specification and Analysis in Maude. In N. Heintze and J. Wing, editors, *Proc. of Workshop on Formal Methods and Security Protocols, 25 June 1998, Indianapolis, Indiana*, 1998.

[17] D. L. Dill. The Murϕ verification system. In R. Alur and T. A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, NJ, July/Aug. 1996. Springer-Verlag.

[18] F. Durán and J. Meseguer. An extensible module algebra for Maude. This volume.

[19] S. Eker. Term rewriting with operator evaluation strategy. This volume.

[20] F. Gadducci and U. Montanari. The tile model. To appear in G. Plotkin, C. Stirling and M. Tofte, eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press. Also, TR-96-27, C.S. Dept., Univ. of Pisa, 1996.

[21] J. Goguen and J. Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theoretical Computer Science*, 105:217–273, 1992.

[22] 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.

[23] 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.

[24] C. Laneve and U. Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6:219–249, 1996.

[25] 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.

[26] 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`.

[27] I. Mason and C. Talcott. A semantics preserving actor translation. In *Proc. ICALP'97*, pages 369–378. Springer LNCS 1256, 1997.

[28] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.

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

[30] 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.

[31] 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.

[32] 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.

[33] J. Meseguer, K. Futatsugi, and T. Winkler. Using rewriting logic to specify, program, integrate, and reuse open concurrent systems of cooperating agents. In *Proceedings of the 1992 International Symposium on New Models for Software Architecture, Tokyo, Japan, November 1992*, pages 61–106. Research Institute of Software Engineering, 1992.

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

[35] P. Viry. Input/output for ELAN. 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`.

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