Towards a strategy language for Maude ⋆

Narciso Martí-Oliet [a], José Meseguer [b], and Alberto Verdejo [a]

[a] *Facultad de Informática, UCM, Madrid, Spain*

[b] *Department of Computer Science, UIUC, Urbana-Champaign, USA*

Abstract

We describe a first proposal for a strategy language for Maude, to control the rewriting process and to be used at the object level instead of at the metalevel. We also describe a prototype implementation built over Full Maude using the metalevel and the metalanguage facilities provided by Maude. Finally, we include a series of examples that illustrate the main features of the proposed language.

Key words: Maude, rewrite strategies, search, strategy languages.

1 Introduction

The passage from equational logic to rewriting logic allows the specification of systems by means of rules that need not be confluent or terminating, opening up in this way a whole world of new applications. However, this theoretical generality needs some control when the specifications become executable, because the user needs to make sure that the rewriting process does not go in undesired directions. In some cases, given a specification and a starting state term, an execution path is enough for testing executability; for this, the Maude system provides `rewrite` and `frewrite` commands [4, Chapter 5]. In other cases, the user might be interested in exploring all possible execution paths from the starting term; this can be accomplished in Maude by means of a `search` command, that looks for states satisfying some properties by doing a breadth-first exploration of the conceptual computation tree [1] of possible rewrites. This search process is also triggered by Maude to check that

⋆ Research partially supported by CICYT project *MIDAS: Metalenguajes para el diseño y análisis integrado de sistemas móviles y distribuidos* (TIC2003–01000), by ONR Grant N00014-02-1-0715, by NSF Grant CCR-0234524, and by DARPA through Air Force Research Laboratory Contract F30602-02-C-0130.

[1] The nodes of such a tree are terms and its branches represent the one-step rewrites. We refer to it as conceptual in the sense that we do not build the whole tree as a data structure, even though we explore parts of it.

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

rewrite conditions are satisfied in the application of conditional rewrite rules [4, Chapter 5].

But there is a very general additional possibility of being interested in the results of only some execution paths satisfying some constraints. For example, in Real Time Maude modules there is a distinction between eager and lazy rules, and only rewriting paths that satisfy the requirement that lazy rules are only applied when no eager rule can be applied make sense for this kind of modules [6]. Another simple example appears in the paper by Pita and Martí-Oliet on object-oriented network models, where at the object (in both senses) level there is a set of rules that must be applied following a specific order that is controlled by a metaobject [8].

The need to use *strategies* to control the rewriting process under these conditions was recognized from the beginning in the development of rewriting logic and the systems implementing rewriting logic computation. In particular, strategies are an essential part of the ELAN system, that provides a basic set of strategies that the user can use in writing rewrite rules, so that at the specification level it is not enforced a separation between rules and strategies [1,2].

In the Maude system, this need for providing strategies for controlling the rewriting process has been satisfied by developing strategies at the *metalevel*. Taking advantage of the reflective properties of rewriting logic, the `META-LEVEL` module in Maude provides basic operations (also called descent functions) that reflect at the metalevel the processes of rule application and rewriting. Using these operations as basic building blocks, it is possible to define at the metalevel a whole variety of *internal strategy languages* [5,3], that is, the strategy language is defined inside the same rewriting logic framework, instead of being defined as an add-on extralogical feature. Although reflection allows a complete control of the rewriting of a given term using the rewrite rules in a theory, for users unfamiliar with the metalevel there is a price to be paid both conceptually and notationally.

Therefore, we have undertaken the project of providing a basic strategy language for Maude, to be used *at the object level* instead of at the metalevel. This language allows the definition of strategy expressions that control the way a term is rewritten. Although ELAN provided a very good starting point for the development of our language, including both ideas and examples, our design is based on a strict separation between the rewrite rules and the strategy expressions, that are provided in separate modules. Thus, in our proposal it is not possible to use strategy expressions in the rewrite rules of a system module.

Stratego [11,12] is another system that has provided much inspiration. However, we have not taken up their ideas on strategies for term traversal. We do not want to complicate the language for the sake of "completeness," to support, say, everything we have done before in previous examples of strategy languages. We think that the strategy language should provide expressive and

basic enough functionality. Since our language is *extensible*, allowing the user to define new functionality in metalevel language extensions, anybody who wants to do even more things can always use the basic functionality of our language and define the extra functionality needed at the metalevel.

Our initial starting point was to design a language for expressing different forms of search, including strategies for restricting it, but the interaction between search and rewriting goes in both directions because, as mentioned above, search is used when applying a conditional rewrite rule for checking the corresponding rewrite conditions. Therefore, in our strategy+search language there are two kinds of expressions: *strategy expressions* and *search expressions*; both are mutually recursive, because a search expression can include a restriction on the rewrite path being searched for by means of a strategy expression, and a basic strategy expression saying that a conditional rule is applied can be qualified with search expressions specifying which kind of search should be used to check the rewrites in the rule's condition.

As also mentioned above, a key modularity principle followed in our language design is the strict separation between strategies and rules. The language allows defining *strategy modules* that associate specific strategies with system modules. In a system module (at the object level) there are no strategy expressions at all. Moreover, we can have different strategy modules associated with the same system module.

The following section describes our proposed language design. Then, after explaining how we have built a prototype implementation over Full Maude using the metalevel and the metalanguage facilities provided by Maude, we include a series of examples that illustrate the main features of the proposed language in Section 5.

This paper assumes knowledge of the Maude language and system. We refer to the Maude manual for detailed information about both [4]. Moreover, there is a lot of work on strategies in the rewriting community that we do not mention; we refer, among many others, to the survey by Eelco Visser [11].

The web page `http://maude.sip.ucm.es/strategies/` contains the full code for all the examples in this paper and some more, as well as the Maude code of the prototype described in Section 4.

2 The strategy+search language

In this section we explain the design of the language, describing its syntax by means of a Maude presentation of the language; that is, the metalanguage facilities provided by the Maude system allow defining a language inside Maude in a very easy way so that the grammar of the language is given as a Maude signature. Moreover, equations are used to define derived operations in terms of the more basic ones. Since our prototype, described later in Section 4, is implemented as an extension of Full Maude by means of the Maude metalevel, we use the same ideas in the presentation of the syntax. However, we must

point out that using the strategy language does not require any knowledge of the metalevel, and that our idea is to implement the language in the future as part of the Maude system.

2.1 Strategies syntax

A strategy is described as an operation that, when applied to a given term, produces a set of terms as a result, given that the process is nondeterministic in general. A simple set-theoretic semantics for the language will be described in Section 3.

2.1.1 Idle and fail

The simplest strategies are the constants `idle` and `fail`. The first always succeeds, but without modifying the term to which it is applied, while the second always fails, that is, its set of results is empty.

2.1.2 Basic strategies

The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term.

```
subsort Label < BasicStrat < Strat .
```

In this case a rule is applied *anywhere* in the term where it matches satisfying its condition, with no further constraints on the substitution instantiation. In case of conditional rules the default breadth-first search strategy is used for checking the rewrites in the condition. A slightly more general variant allows variables in a rule to be instantiated before its application by means of a substitution, so that the user has more control on the way the rule is applied.

```
op _[_] : Label Substitution -> BasicStrat .
```

The unconstrained case `L` can also be expressed as `L[none]`, where `none` denotes the identity (empty) substitution.

For conditional rules, rewrite conditions can be controlled by means of search expressions (see Section 2.2). As before, the substitution can be omitted if it is empty.

```
op _[_]{_} : Label Substitution List(Search) -> BasicStrat .
op _{_} : Label List(Search) -> BasicStrat .
```

A strategy expression of the form `L[S]{B1 ... Bn}` denotes a basic strategy that applies *anywhere* in a given state term the rule `L` with variables instantiated by means of the substitution `S` and using `B1`, ..., `Bn` as search expressions to check the rewrites in the condition of `L`. The number of such rewrites must be n for the expression to be meaningful.

2.1.3 Top

We consider that the most common case allows applying a rule anywhere in a given term, as explained above, but we also provide an operation to restrict the application of a rule just to the *top* of the term, because in some examples like structural operational semantics, the only interesting or allowed rewrite steps happen at the top (see Section 5.3).

```
op top : BasicStrat  -> Strat .
```

`top(BE)` applies the basic strategy `BE` only at the top of a given state term. Note however that even applying a rule at the top is nondeterministic due to multiple matchings, which are possible because matching takes place modulo the equational attributes of the operators, such as associativity or commutativity.

2.1.4 Tests

Since matching is one of the basic steps that take place when applying a rule, the strategies that test some property of a given state term are based on matching. As in applying a rule, we distinguish between matching anywhere and matching only at the top of a given term.

```
subsort Test < Strat .
op xmatch_s.t._ : Term EqCondition -> Test .
op match_s.t._ : Term EqCondition -> Test .
```

`xmatch T s.t. C` is a test that, when applied to a given state term `T'`, is successful if there is a subterm of `T'` that matches the pattern `T` (that is, matching is allowed *anywhere* in the state term) and then the condition `C` is satisfied with the substitution for the variables obtained in the matching, and is false otherwise. `match T s.t. C` corresponds to matching only at the *top*. When the condition `C` is simply `true`, it can be omitted.

Tests are seen as strategies that check a property on a state, so that the test *qua* strategy is successful if true and fails if false. In the first case, the state is not changed.

In particular, the strategy constants `idle`, that is the identity, and `fail`, that always fails, correspond respectively to basic tests as follows. The `idle` strategy can be represented by a match with a variable that always succeeds and a condition that is trivially true, so that any state passes this test. On the other hand, the `fail` strategy can be represented by a match with a variable that always succeeds and a condition that is trivially false, and therefore is never satisfied and thus no state can pass this test.

2.1.5 Regular expressions

Basic strategies are combined so that strategies are applied to execution paths. The first strategy combinators we consider are the typical regular expression constructions: concatenation, union, and iteration.

```
op _;_ : Strat Strat -> Strat [assoc] .      *** concatenation
op _|_ : Strat Strat -> Strat [assoc comm] . *** union
op _* : Strat -> Strat .                    *** iteration (0 or more)
op _+ : Strat -> Strat .                    *** iteration (1 or more)
```

Notice the attributes in the concatenation and union operators. In particular, the commutativity property of the union provides a form of nondeterminism in the way the solutions are found.

A strategy of the form `E ; P` (with `P` a test) filters out all those results from `E` that do not satisfy a test `P`.

In order to avoid writing long expressions of the form `L1 | ... | Ln` where `Li` are the labels of all rules in a module, we provide some abbreviations:

```
op all : ModuleName -> Strat .
op all# : ModuleName -> Strat .
```

`all(M)` denotes the strategy union of all the rule labels (understood as basic strategies) declared in module `M`, while `all#(M)` denotes the strategy union of all the rule labels declared in module `M` and all its imported submodules.

2.1.6 If-then-else and its derived strategies

Our next strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy. We have borrowed this idea from Stratego [12], but it also appears in ELAN [2, Example 5.2].

```
op if_then_else_fi : Strat Strat Strat -> Strat .
```

The behaviour of the strategy expression `if E then E' else E'' fi` is as follows: in a given state term, the strategy `E` is evaluated; if `E` is successful, the strategy `E'` is evaluated in the resulting states, otherwise `E''` is evaluated in the *initial* state.

Note that, as mentioned above, the first argument is also a strategy term and not a test. Since `Test` is a subsort of `Strat`, we have the particular case `if P then E' else E'' fi` for a test `P` where evaluation coincides with the typical Boolean case distinction: `E'` is evaluated when the test `P` is true and `E''` when the test is false, taking into account that a test *qua* strategy fails when false.

Using the if-then-else combinator, we can define many other useful strategy combinators as derived operations. `E orelse E'` evaluates `E` in a given state; if such evaluation is successful, its results are the final ones, but if it fails, then `E'` is evaluated in the initial state.

```
op _orelse_ : Strat Strat -> Strat .
eq E orelse E' = if E then idle else E' fi .
```

`not(E)` reverses the result of evaluating `E`, so that `not(E)` fails when `E` is successful and vice versa.

```
op not : Strat -> Strat .
eq not(E) = if E then fail else idle fi .
```

An interesting use of `not(E)` is the following "normalization" (or "repeat until the end") operation:

```
op _! : Strat -> Strat .
eq E ! = E * ; not(E) .
```

`try(E)` evaluates `E` in a given state; if it is successful, the corresponding result is given, but if it fails, the initial state is returned.

```
op try : Strat -> Strat .
eq try(E) = if E then idle else idle fi .
```

Evaluation of `test(E)` checks the success/failure result of `E`, but it does not change the given initial state.

```
op test : Strat -> Strat .
eq test(E) = if not(E) then fail else idle fi .
```

2.1.7 Depth

In order to be able to require a depth bound, that is, a bound on the length of the computation that corresponds to the depth in the computation tree, we introduce the following operation:

```
sort DStrat .   subsort Strat < DStrat .
op depth : Nat Strat -> DStrat .
```

Note that the depth operation can only be applied at the top of a strategy expression, since the result is not again of sort `Strat`, but of sort `DStrat` (strategy with depth).

2.1.8 Rewriting of subterms

With the previous combinators, we cannot force the application of a strategy to a specific subterm of the given initial term. In particular the scope of the substitution in the (`x`)`match` combinators is only the corresponding condition. We can have more control over the way different subterms of a given state are rewritten by means of the (`x`)`matchrew` combinators.

```
sort TermStrat .
op _using_ : Term Strat -> TermStrat .
op xmatchrew_s.t._by_ : Term EqCondition List(TermStrat) -> Strat .
op matchrew_s.t._by_  : Term EqCondition List(TermStrat) -> Strat .
```

When the strategy expression

```
xmatchrew T s.t. C by T1 using E1, ..., Tn using En
```

is applied to a state term `T'`, first a subterm of `T'` that matches `T` and satisfies `C` is selected. Then, the terms `T1`, ..., `Tn` (which must be disjoint subterms of `T`), instantiated appropriately, are rewritten as described by the strategy expressions `E1`, ..., `En`, respectively. The results are combined in `T` and then substituted in `T'`, in the way illustrated in Figure 1.

The strategy expressions `E1`, ..., `En` can make use of the variables instanti-

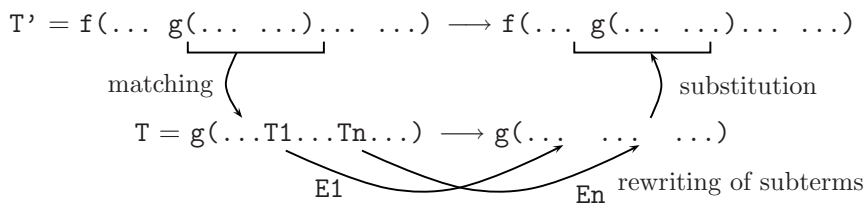

Fig. 1. Behaviour of the `xmatchrew` combinator.

ated in the matching, thus taking care of information extracted from the state term (see the examples in Sections 5.1 and 5.6).

The version `matchrew` works in the same way, but performing matching only at the top. In both versions, when the condition is `true` it can be omitted.

In ELAN and Stratego there is a strategy combination mechanism called *congruence operators* [2,12]. For each syntax constructor `C` there is a corresponding congruence operator, also denoted by `C`. If `C` is an n-ary constructor, then the corresponding congruence operator allows defining the strategy `C(E1,...,En)`. Such a strategy applies only to terms of the form `C(T1,...,Tn)`, and its results are the terms `C(T1',...,T2')`, provided the application of each strategy `Ei` to each term `Ti` succeeds with result `Ti'`. Congruence operators can be simulated in our language by means of the `matchrew` combinator, since the above strategy `C(E1,...,En)` can be represented as

```
matchrew C(X1,...,Xn) by X1 using E1, ..., Xn using En
```

where variables `Xi` of the appropriate sorts are used to match the arguments of the term `C(T1,...,Tn)`.

2.1.9 Recursion

Recursion is achieved by giving a name to a strategy expression and using this name in the strategy expression itself or in other related strategies. This is done in strategy+search modules (see Section 2.3). Concrete examples will be shown in Section 5.

2.2 Search syntax

There are basic constructors for the most usual kinds of search: breadth-first, depth-first, and iterated bounded depth-first. In all of them, the first argument represents the number of requested solutions (with `unbounded` for "all solutions"), while in the third case, the second argument is the increment between iterations.

```
op bfs : Bound DStrat -> Search .
op dfs : Bound DStrat -> Search .
op ibdfs : Bound Nat DStrat -> Search .
```

The current (default) Maude `search` command [4, Section 17.4]

```
search in M : _ =>* T such that C
```

is equivalent to the search expression

```
bfs(unbounded, all#(M)* ; match T s.t. C)
```

where the strategy expression given as second argument takes care of iteration (remember that `all#(M)` denotes the strategy union of all the rule labels declared in module `M` and all its imported submodules), final pattern matching at the top, and condition checking.

2.3 Strategies and search modules and commands

Given a Maude module `M`, the user can write one or more strategy+search modules to define strategies for `M`. In the current design, such strategy+search modules have the following form:

```
stratdef STRAT is
  including M .
  including STRAT1 .    ...    including STRATp .
  strat EI1 = E1 .      ...    strat EIn = En .
  search BI1 = B1 .     ...    search BIm = Bm .
endsd
```

where `STRAT1`, ..., `STRATp` are imported strategy+search modules, `EI1`, ..., `EIn`, `BI1`, ..., `BIm` are identifiers, `E1`, ..., `En` are strategy expressions (over the language of labels provided by `M`), and `B1`, ..., `Bm` are search expressions (over the same language). In the future, we plan to study parameterization mechanisms for strategy+search modules.

Note that the identifiers `EI1`, ..., `EIn`, `BI1`, ..., `BIm` can appear in the righthand expressions `E1`, ..., `En`, `B1`, ..., `Bm`, thus allowing (mutually) recursive definitions.

The idea is that these strategy and search declarations provide useful abbreviations for strategy and search expressions that the user can then use in commands over the module `M`. The two basic commands are `srew T using E` (strategy rewrite), which rewrites a term `T` using a strategy expression `E`, and `search T using B`, which performs a search starting from `T` according to the search expression `B`.

3 Semantics

We propose the following simple set-theoretic semantics at an abstract level, where we are only interested in the results of evaluating the strategy on a given state term, and not in the way the results of such an evaluation have been obtained. Everything that follows is said with respect to an implicit system module M where rules have been declared.

A strategy expression denotes a function from terms (as states) to (possibly infinite) sets of terms, denoting the successful states. More specifically, the resulting set of terms is a subset of the set of nodes of the computation

tree in module M whose root is the given initial term for the strategy. If the result set is empty, then the strategy has failed on the initial term. In particular, independently of the term t, for the constants `idle` and `fail` we have $\texttt{idle}(t) = \{t\}$, and $\texttt{fail}(t) = \{\}$, the empty set.

For a basic strategy BE and a term t, $BE(t)$ is the set of terms obtained as result of applying the basic strategy BE (that is, a rule possibly constrained by a given substitution) to t anywhere. The set $\texttt{top}(BE)(t)$ is the subset of $BE(t)$ obtained as result of applying the basic strategy BE to t at the top. Note that the restriction of applying the strategy only at the top does not force the resulting set to be either unitary or empty, because of multiple possible matches due to equational attributes of the operators in the module M.

For the basic tests, $(\texttt{xmatch}\ t'\ \texttt{s.t.}\ C)(t) = \{t\}$ if there is a subterm of t that matches the pattern t' with resulting substitution σ such that $\sigma(C)$ evaluates to *true*. Otherwise, either because no subterm of t matches t' or because the condition is not satisfied, $(\texttt{xmatch}\ t'\ \texttt{s.t.}\ C)(t) = \{\}$. $(\texttt{match}\ t'\ \texttt{s.t.}\ C)(t)$ is the special case in which the subterm must coincide with t because matching is only allowed at the top.

The regular expression combinators have the expected semantics, where E^n, for $n \in \mathit{Nat}$, is an auxiliary definition:

$$
\begin{aligned}
(E|E')(t) &= E(t) \cup E'(t) \\
(E; E')(t) &= \bigcup \{E'(t') \mid t' \in E(t)\} \\
E^0(t) &= \{t\} \\
E^{n+1}(t) &= (E; E^n)(t) \\
E^*(t) &= \bigcup \{E^n(t) \mid n \in \mathit{Nat}\} \\
E^+(t) &= \bigcup \{E^n(t) \mid n \in \mathit{Nat}, n \neq 0\}
\end{aligned}
$$

For the if-then-else combinator, we test the strategy in the first argument. If $E(t)$ is not empty, $(\texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E''\ \texttt{fi})(t) = (E; E')(t)$; otherwise, $(\texttt{if}\ E\ \texttt{then}\ E'\ \texttt{else}\ E''\ \texttt{fi})(t) = E''(t)$.

The semantics of subterm rewriting requires the notion of context. In the strategy expression $E = \texttt{xmatchrew}\ t\ \texttt{s.t.}\ C\ \texttt{by}\ t_1\ \texttt{using}\ E_1, \ldots, t_n\ \texttt{using}\ E_n$, the terms t_i, for $1 \leq i \leq n$, must be disjoint subterms of the pattern t, which is therefore of the form $c[t_1, \ldots, t_n]$ for some context c (modulo structural axioms, like associativity or commutativity). When the strategy expression E is applied to a term t', we first match a subterm of t' to t, (that is, $t' = c'[\sigma(t)]$ for another context c') and the resulting substitution σ must satisfy the condition C, that is, $\sigma(C)$ evaluates to *true*; otherwise, E fails. In the former case, each strategy E_i is applied to $\sigma(t_i)$, so that the final result is the set of terms of the form $c'[\sigma(c)[t'_1, \ldots, t'_n]]$, for $t'_i \in E_i(\sigma(t_i))$. Note that if one of these sets is empty, that is, the corresponding strategy fails, then the whole strategy E also fails. The strategy expression $E = \texttt{matchrew}\ t\ \texttt{s.t.}\ C\ \texttt{by}\ t_1\ \texttt{using}\ E_1, \ldots, t_n\ \texttt{using}\ E_n$ is the special case in which matching can

only happen at the top, that is, the context c' is empty.

The semantics of recursive definitions is obtained in the usual way, as least fixpoints of transformations over strategies.

`depth`$(n, E)(t)$ is the subset of $E(t)$ formed by those terms $t' \in E(t)$ whose minimum path length from t to t' is less than or equal to n.

The meaning of a search expression is less abstract than the meaning of a strategy expression, because the requirement of doing the search in a concrete way forces the results to be obtained in a concrete order. Therefore, a search expression denotes a function from terms (as states) to (possibly infinite) lists (sequences) of terms. Given a search expression $B(n, DE)$, its meaning $B(n, DE)(t)$ consists of elements in $DE(t)$ ordered in the way the search B takes place over the computation tree, and with $size(B(n, DE)(t)) \leq n$.

4 Prototype implementation in Maude

Using the Maude metalevel, we have implemented a prototype of the strategy+search language as an extension of Full Maude. It consists of several functions that work with a labelled version of the conceptual computation tree produced when applying a strategy `E` to a given state term `T`. Nodes in this tree are tuples formed by a term, a strategy, and possibly other information. The root is `< T, E >`, and the children of a node `< T', E' >` are the terms obtained from `T'` by rewriting as described by `E'`, paired with the corresponding remainder of `E'`. In a successful path, the strategy at the leaf node is empty, meaning that nothing is left to do, which corresponds to a complete successful application of the strategy `E`.

Concretely, the functions implementing the language work on paths on this kind of trees. Internal nodes are labelled with enough information to know which is the next child to be explored. The specific information saved depends on the top constructor in the strategy expression, and it is used to traverse the conceptual tree by backtracking in a depth-first way (the only search we have implemented for the time being). Part of the syntax used to build paths is as follows:

```
sorts Node Path .
subsorts Term Node < Path .
op <_,_> : Term Strat -> Node .
op <_,_,_> : Term Strat Nat -> Node .
op <_,_,_> : Term Strat Path -> Node .
op emptyP : -> Path .
op p : Path Path -> Path [assoc id: emptyP] .
op fail : -> Path .
```

The two main functions are `first` and `next`. The combination of these two functions serves to find (in a depth-first order) all the solutions for the application of a strategy to a given state term.

```
op first : Module SSModule Path -> Path .
op next  : Module SSModule Path -> Path .
```

The function `first` receives a system module, a strategy+search module, and a path (initially this path is formed only by the root of the tree), and it returns the first successful path obtained from the given path. The function `next` receives initially a successful path and returns the path to the next solution in the tree (or `fail` if there are no more solutions). They are implemented in a mutually recursive way, distinguishing cases on the strategy expression in the last node of the given path, and with the help of the metalevel descent functions `metaApply`, `metaXapply`, `metaMatch`, and `metaXmatch` [4, Section 10.4].

We present below the handling of two cases: the application of a rule with label `L` instantiated with substitution `Sb`, `L[Sb]`, and the sequential composition of two strategies, `E ; E'`.

If we look for the first solution extending path `p(PA, < T, L[Sb] >)` then we ask for the first solution (`0`) of the application of `L` using `metaXapply`. If there is such a solution `T'`, then we are finished, and a successful path terminating in `T'` is returned. Note how the solution number is kept in the internal node. Otherwise, we have to backtrack by looking for the next solution of path `PA`. If we look for the next solution, we ask `metaXapply` for the next solution.

```
ceq first(M, SSM, p(PA, < T, L[Sb] >)) =
      p(p(PA, < T, L[Sb], 0 >), T')
 if { T', Ty, Sb', CX } :=
      metaXapply(M, T, L, Sb, 0, unbounded, 0) .

eq first(M, SSM, p(PA, < T, L[Sb] >)) = next(M, SSM, PA) [owise] .

ceq next(M, SSM, p(PA, < T, L[Sb], N >)) =
      p(p(PA, < T, L[Sb], N + 1 >), T')
 if { T', Ty, Sb', CX } :=
      metaXapply(M, T, L, Sb, 0, unbounded, N + 1) .

eq next(M, SSM, p(PA, < T, L[Sb], N >)) = next(M, SSM, PA) [owise] .
```

In order to apply a strategy `E ; E'` to a state term `T`, first `E` is applied to `T`. If this application is successful and it returns `T'` (as the first obtained solution), then `E'` is applied to `T'`. Note how the path `p(PA',T')` obtained when applying `E` to `T` is saved in the node `< T, E ; E', p(PA',T') >`, because it is needed when more solutions are searched for (and all the solutions of `< T', E' >` have been already explored), or when `E'` fails. In both cases the next solution of `< T, E >` is searched.

```
ceq first(M, SSM, p(PA, < T, E ; E' >)) =
    first(M, SSM, p(p(PA, < T, E ; E', p(PA',T') >), < T', E' >))
 if p(PA', T') := first(M, SSM, < T, E >) .
```

```
eq first(M, SSM, p(PA, < T, E ; E' >)) = next(M, SSM, PA) [owise] .

ceq next(M, SSM, p(PA, < T, E ; E', PA'' >)) =
    first(M, SSM, p(p(PA, < T, E ; E', p(PA',T')>), < T', E' >))
 if p(PA', T') := next(M, SSM, PA'') .

eq next(M, SSM, p(PA, < T, E ; E', PA'' >)) =
     next(M, SSM, PA) [owise] .
```

The metalanguage features of Maude allow completing the prototype with a user interface where strategy+search modules can be loaded, and commands to rewrite a term using a strategy or to search according to a search expression can be executed. These commands allow a step-by-step generation of all the possible results of rewriting a term using a strategy.

The *syntax definition* for the strategy+search language is accomplished by defining a data type `StratDefModule`, which can be done with very flexible user-definable *mixfix* syntax, that can mirror the concrete syntax explained in Section 2. Particularities at the lexical level can be accommodated by user-definable *bubble sorts*, that tailor the adequate notions of token and identifier. Parsing and pretty printing are accomplished by the functions `metaParse` and `metaPrettyPrint` in `META-LEVEL` [4, Chapter 11].

Input/output of strategy+search modules and of commands for execution is accomplished by the predefined module `LOOP-MODE`, that provides a generic read-eval-print loop. This module has an operator `[_,_,_]` that can be seen as a persistent object with an input and output channel (the first and third arguments, respectively), and a state (given by its second argument). Our prototype user interface has been implemented as an extension of Full Maude. Full Maude maintains as the state of the loop object a database of modules entered into the system. We have extended this state to maintain values of sort `Path` to remember the last result found. Then, we defined rewrite rules that describe the behaviour associated with the new commands. All the examples in the following section have been executed using this extension of Full Maude.

5 Some examples

In this section we show some examples to illustrate the use of strategies. The search+strategy modules and commands are written between parentheses because they are used as input to the loop object of Full Maude.

5.1 Blackboard

The first example is a simple game. You have a blackboard on which several natural numbers have been written. A legal move consists in selecting two numbers in the blackboard, removing them, and writing their arithmetic mean. The objective of the game is to get the greatest possible number written on

the blackboard at the end. The specification of the game in Maude is also quite simple.

```
(mod BLACKBOARD is
  pr NAT .
  sort BB .
  subsort Nat < BB .
  op __ : BB BB -> BB [assoc comm] .
  vars M N : Nat .
  rl [play] : M N => (M + N) quo 2 .
endm)
```

A player can choose the numbers randomly, or can follow some strategy. Possible strategies consist in taking always the two greatest numbers, or the two smallest, or taking the greatest and the smallest. The following module extends the `BLACKBOARD` module with operations to get the maximum or minimum number in a blackboard, and for removing an element in the blackboard.

```
(mod EXT-BB is
  pr BLACKBOARD .
  ops max min : BB -> Nat .
  op remove : Nat BB -> BB .
  vars M N X Y : Nat .  var B : BB .
  eq max(N) = N .
  eq max(N B) = if N > max(B) then N else max(B) fi .
  eq min(N) = N .
  eq min(N B) = if N < min(B) then N else min(B) fi .
  eq remove(X, X B) = B .
endm)
```

The module `BB-STRAT` below defines the three mentioned strategies. Note how the `matchrew` strategies constructor is used to get information about the state term that is then used in the definition of how the rule `play` has to be applied.

```
(stratdef BB-STRAT is
  including EXT-BB .
  strat maxmin = (matchrew B s.t. X := max(B) /\
                                  Y := min(B) by
                     B using play[M <- X ; N <- Y] ) ! .
  strat maxmax = (matchrew B s.t. X := max(B) /\
                                  Y := max(remove(X,B)) by
                     B using play[M <- X ; N <- Y] ) ! .
  strat minmin = (matchrew B s.t. X := min(B) /\
                                  Y := min(remove(X,B)) by
                     B using play[M <- X ; N <- Y] ) ! .
endsd)
```

```
Maude> (srew 2000 20 2 200 10 50 using maxmin .)
result NzNat :  178
```

```
Maude> (srew 2000 20 2 200 10 50 using maxmax .)
result NzNat :  77
Maude> (srew 2000 20 2 200 10 50 using minmin .)
result NzNat :  1057
```

5.2 Map

This example illustrates how a strategy $map(S)$, that applies a strategy S once to every element in a list, can be defined in our language. First we define a system module declaring lists of elements and a conditional rewrite rule that decomposes a list in its head and tail and whose conditions rewrite these components. This rule has been defined only for the purpose of defining the strategy and, as we shall see below, can be avoided.

```
(mod MAP is
  inc ELEM .
  sort List .  subsort Elem < List .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
  vars E E' : Elem .  vars L L' : List .
  crl [list] : E L => E' L' if E => E' /\ L => L' .
endm)
```

The first possible implementation of $map(S)$ (strategy `map1` below) uses an if-then-else to distinguish between the empty and nonempty lists. If the state list matches the `nil` constructor then the constant `idle` strategy is used. If the list is nonempty then the rule `list` is used and the strategy says how: it has to be applied at the top and its first rewrite condition has to be solved using the elements strategy S to rewrite the head of the list, and the second rewrite condition has to be solved using recursively the list strategy $map(S)$.

The problem with this kind of implementation is that a rewrite rule has to be included for the only purpose of separating the components of the state term that have to be rewritten in a controlled way. The `matchrew` constructor can be used to solve this problem. It is used in the second implementation (strategy `map2` below). In this case the `orelse` constructor is used to distinguish cases (only for illustrating different possibilities). If the state list matches the `nil` constructor, then it is trivially successful. Otherwise, the list is decomposed with the pattern `E L`, and then `E` is rewritten using S and `L` is rewritten using $map(S)$.

```
(stratdef MAP-STRAT is
  including MAP .
  strat S = [...] . *** strategy for elements
  strat map1 = if (match nil) then idle
               else top(list{dfs(S) dfs(map1)}) fi .
  strat map2 = (match nil) orelse
               (matchrew E L by E using S, L using map2) .
endsd)
```

5.3 CCS operational semantics

In this section we show how the rewrite rules implementing a structural operational semantics can be controlled with our strategy+search language. We have studied how Maude can be used to represent and implement the CCS operational semantics elsewhere [10,9]. In [10] we showed which implementation problems can be found with this kind of representations, and how they can be solved in Maude 2.0 using some "tricks", like the `frozen` attribute that disallows rewriting of subterms, or dummy operators used to control what rules can be used to solve a rewrite condition.

The module `CCS-SEMANTICS` below contains the CCS semantics representation without these tricks. [2] In this kind of representation, semantic transitions are represented as rewrites, and semantic rules are represented as conditional rewrite rules, where the main rewrite corresponds to the transition in the conclusion of the semantic rule, and the condition rewrites correspond to transitions in the premises. In CCS transitions are labelled with actions; in our Maude representation this label is part of the righthand side term, built with the `{_}_` operator.

```
(mod CCS-SEMANTICS is
  protecting CCS-CONTEXT .
  sort ActProcess .  subsort Process < ActProcess .
  op {_}_ : Act ActProcess -> ActProcess .
  var L : Label .  var X : ProcessId .  vars P P' Q Q' : Process .
  var A : Act .    var AP : ActProcess .
  rl  [prefix] : A . P => {A}P .
  crl [sum]  : P + Q => {A}P' if P => {A}P' .
  crl [par1] : P | Q => {A}(P' | Q) if P => {A}P' .
  crl [par2] : P | Q => {tau}(P' | Q')
           if P => {L}P' /\ Q => {~ L}Q' .
  crl [res]  : P \ L => {A}(P' \ L)
           if P => {A}P' /\ A =/= L /\ A =/= ~ L .
  crl [def]  : X => {A}P
           if (X definedIn context) /\ def(X,context) => {A}P .
*** transitive closure
  crl [more] : P => {A}AP if P => {A}Q /\ Q => AP .
endm)
```

The first six rules correspond to CCS semantic rules. These rules should be applied only at the top of a process term. Rule `more` represents the transitive closure of the CCS transition relation, defined in a mathematical way by

$$\frac{P \to P' \qquad P' \to^* Q}{P \to^* Q}$$

Here we have two kinds of transitions, $\to$ and $\to^*$, and when trying to solve

[2] We have omitted the modules specifying the CCS syntax and contexts, and rules corresponding to the relabelling operator. They can be found, for example, in [10].

the first premise we know that the rules to be used are the ones defining CCS "one-step" transitions. But when both kinds of transitions are represented in Maude, the same rewrite relation is used (`=>`). That is the reason why we need to control which rules are used when solving the rewrite conditions in rule `more` above. The following module defines the strategies used to control the rewriting process in the desired way.

```
(stratdef STRAT is
  strat ccs = top(prefix) |
              top(sum{dfs(ccs)}) |
              top(par1{dfs(ccs)}) |
              top(par2{dfs(ccs) dfs(ccs)}) |
              top(res{dfs(ccs)}) |
              top(def{dfs(ccs)}) .
  strat trans = idle | top(more{dfs(ccs) dfs(trans)}) .
endsd)
```

A simple vending machine, where two kinds of coins can be inserted and depending on the inserted coin a big or little cake can be collected, can be defined in a CCS context in the following way:

```
eq context = ('Ven  =def '2p . 'VenB  +  '1p . 'VenL)  &
             ('VenB =def 'big . 'collectB . 'Ven)    &
             ('VenL =def 'little . 'collectL . 'Ven)  .
```

We can rewrite `'Ven` with strategy `trans` to check if the trace

```
{'2p}{'big}{'collectB}
```

is possible in CCS.

```
Maude> (srew 'Ven using trans ; (match {'2p}{'big}{'collectB}AP) .)
result ActProcess :
  {'2p}{'big}{'collectB}'Ven
```

This command succeeds because the trace `{'2p}{'big}{'collectB}` is in the first path explored by the current implementation of the strategy language. Since process `'Ven` is infinite, the rewriting tree produced by strategy `trans` has infinite branches. The depth-first search fails to find traces which are not an extension of `{'2p}`. Although `{'1p}{'little}` is also a correct trace, it will not be found. Here a (not implemented yet) breadth-first search should be used:

```
(search 'Ven using bfs(trans ; (match {'1p}{'little}AP)) .)
```

5.4 Backtracking: labyrinth

In this section we show a generic strategy useful for solving a problem using backtracking. It assumes that partial solutions are represented as lists of decisions, and that there are predicates `isOk`, to check if a partial solution is extensible to a (complete) solution, and `isSolution`, to check if we already

have a solution. It also assumes a rule `expand` that extends a partial solution:

```
crl [expand] : L => L P if next(L) => P .
```

and possibly several rules `next` that specify how a term like `next(L)` can be rewriten to a value that extends `L`. We give an example below.

With these ingredients we can define a generic strategy that defines how a problem has to be solved by means of backtracking.

```
(stratdef BACKTRACKING-STRAT is
  strat solve =  if (match L s.t. isSolution(L))
                 then idle
                 else top(expand{dfs(next)}) ;
                      (match L s.t. isOk(L)) ;
                      solve
                 fi .
endsd)
```

This strategy first checks if it has already obtained a solution. If this is the case, it finishes. Otherwise, it applies at the top the `expand` rule, using rules `next` to solve the condition; then, it checks if the extension is right, and continues recursively.

The following module instantiates the components described above for solving a labyrinth.

```
(mod LABYRINTH is
  pr NAT .
  sorts Pos List .  subsort Pos < List .
  op [_,_] : Nat Nat -> Pos .
  op nil : -> List .
  op __ : List List -> List [assoc id: nil] .
  op contains : List Pos -> Bool .
  ops isSolution isOk : List -> Bool .
  op next : List -> Pos .
  op wall : -> List .
  vars X Y : Nat .  var P Q : Pos .  var L : List .
  eq isSolution(L [8,8]) = true .
  eq isSolution(L) = false [owise] .
  eq contains(nil, P) = false .
  eq contains(Q L, P) = if P == Q then true else contains(L, P) fi .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
                     and not(contains(L, [X,Y])) and
                         not(contains(wall, [X,Y])) .
  crl [expand] : L => L P if next(L) => P .
  rl [next] : next(L [X,Y]) => [X + 1, Y] .
  rl [next] : next(L [X,Y]) => [X, Y + 1] .
  rl [next] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [next] : next(L [X,Y]) => [X, sd(Y, 1)] .
endm)
```

The same strategy has been used to solve the ubiquitous queens problem (see http://maude.sip.ucm.es/strategies/).

5.5 Network management

Strategies (at the metalevel) in Maude have also been studied in the context of an object-oriented model for broadband telecommunication networks [8,7]. The basic objects of the model are nodes, links, and connections. Nodes represent the network points where the communication signals are treated, and a network is formed by a set of links together with the nodes that they join and the corresponding connections between nodes. The system evolves by requests (queries, modifications, deletions) that produce a chain of messages between the objects, until a new stable configuration corresponding to the request is reached. In [8] different specifications of these evolutions are studied. For example, when a modification message is received by the network, a protocol has to be followed. If this protocol cannot be followed, error messages have to be generated. This can be specified at the object level by complicating the specification of the protocol, or at the metalevel by specifying a `Mediator` object that controls the network by using a concrete strategy language defined for this case also at the metalevel [8].

By using the strategy language defined in this paper, we can define the `Mediator` at the object level, and can control the rules specifying its behaviour by means of strategies specified in a separate strategy module. The abstract specification of the `Mediator` object is as follows:

```
(omod MEDIATOR is
  including NETWORK .
  class Mediator | Config : Configuration .  [...]
  crl [ChDemand-ok] : ChDemand(O, N, No1, No2, << S ; D >>)
      < N : Mediator | Config : C >
   => < N : Mediator | Config : C' >
      (To O AckChDemand No1 and No2 in N)
   if  C MCom(O, N, No1, No2, << S ; D >>) => C' .
  rl [ChDemand-NoConn] : ChDemand(O, N, No1, No2, << S ; D >>)
      < N : Mediator | Config : C >
   => < N : Mediator | >
      (To O NoConnectionBetween No1 and No2 in N) .
  rl [ChDemand-NoCap] : ChDemand(O, N, No1, No2, << S ; D >>)
      < N : Mediator | Config : C >
   => < N : Mediator | > (To O ServiceCapacityNotSupported) .
endom)
```

Rule `ChDemand-ok` expresses a successful execution. If the rewrite condition can be solved in the desired way (following the correct protocol) then the modification request `ChDemand` can be attended. Otherwise, error messages will be generated depending on where the protocol fails. The two possible errors are that there is no connection between the given nodes, and that there is

no port of the needed capacity in one of the nodes traversed by the connection.

The strategies that describe how the protocol works and when the above rules have to be applied are as follows:

```
(stratdef MEDIATOR-STRATEGIES is
  strat iterate = if LinkListLoad then PortNode ; PortNode ; iterate
                  else idle fi .
  strat Sconf = if MCom then iterate else MComNS ; iterate fi .
  strat checkNoConn = xmatchrew ChDemand(O, N, No1, No2, << S ; D >>)
                                < N : Mediator | Config : C > by
        C MCom(O, N, No1, No2, << S ; D >>) using not(MCom | MComNS) .
  strat Smediator = ChDemand-ok{dfs(Sconf)}
                    orelse ((checkNoConn ; ChDemand-NoConn)
                            orelse ChDemand-NoCap) .
endsd)
```

Strategy `Sconf` describes the correct protocol. The rules `Mcom`, `McomNS`, `LinkListLoad`, and `PortNode` (in the `NETWORK` module) describe the behaviour of the network [8,7]. Strategy `Smediator` controls the mediator. First, it tries to apply rule `ChDemand-ok` ensuring that `Sconf` is used to rewrite the condition. If this is not possible, then there is an error. Strategy `checkNoConn` checks that the rules `Mcom` or `McomNS` cannot be applied to the controlled network. This means that the desired connection does not exist, and the rule `ChDemand-NoConn` is applied. Otherwise, the problem is the lack of capacity, and the rule `ChDemand-NoCap` is applied.

We point out that the use of the `xmatchrew` combinator in the strategy `checkNoConn` above does not follow the general pattern explained in Section 2.1.8, because the term that is "rewritten" after the matching test is not a subterm of the pattern. However, the strategy works in the expected way because it only checks whether a property is true or not, without really rewriting the term. We think that this is not the most appropriate way of using this combinator and thus we have not included this case in our general explanation, leaving for future work a detailed study of the usefulness these special cases.

5.6 Strategies with memory: insertion sort

Sometimes a strategy needs to remember some information about what it has already done in order to know what it has to do next. In our current proposal, this "memory" keeping auxiliary information can only be maintained as part of the term being rewritten. We propose introducing a free constructor

```
op <_,_> : State Memory -> Conf .
```

where `State` is the sort of terms the strategy is intended for, and values of sort `Memory` keep the extra information needed by the strategy. There must also be rules describing the behaviour of this memory, whose application can be controlled by a strategy, just as one controls the rules rewriting states.

$Y := 2$
while $Y \leq N$ do
  $X := Y$
  while $X > 1 \ \wedge \ V[X-1] > V[X]$ do
    switch $V[X-1]$ and $V[X]$
    $X := X - 1$
  $Y := Y + 1$

Fig. 2. Insertion sort.

The strategy that implements the insertion sort algorithm follows this approach. First we have a module that defines arrays as sets of pairs and a rule to switch the values in two positions of the array.

```
(mod SORTING is
  pr NAT .
  sorts Pair PairSet .
  subsort Pair < PairSet .
  op (_,_) : Nat Nat -> Pair .
  op empty : -> PairSet .
  op __ : PairSet PairSet -> PairSet [assoc comm id: empty] .
  op length : PairSet -> Nat .
  vars I J V W : Nat .  var PS : PairSet .
  eq length(empty) = 0 .
  eq length((I, V) PS ) = length(PS) + 1 .
  rl [switch] : (J, V) (I, W) => (J, W) (I, V) .
endm)
```

The imperative pseudocode for the insertion sort algorithm is shown in Figure 2 (for sorting an array $V[1..N]$).

The algorithm keeps two indices, one pointing to the next element to be inserted between the already sorted elements, and another pointing to the element which is being inserted. So in this case the memory needed by the strategy consists of two natural numbers. The following module defines the memory and the needed operations to change it as rewrite rules.

```
(mod EXT-SORTING is
  pr SORTING .
  sorts Memory Conf .
  op [_|_] : Nat Nat -> Memory .
  op <_,_> : PairSet Memory -> Conf .
  vars X Y J I V W : Nat .  var PS : PairSet .
  rl [setY] : [ Y | X ] => [ 2 | X ] .
  rl [setX] : [ Y | X ] => [ Y | Y ] .
  rl [decX] : [ Y | X ] => [ Y | sd(X,1) ] .
  rl [incY] : [ Y | X ] => [ Y + 1 | X ] .
endm)
```

The following module defines the strategy `insort` that rewrites terms of sort `Conf`. The algorithm can be represented as a strategy in several different

ways; this particular way is just one example that tries to mimic the pseudocode in Figure 2. Loops are represented by means of the "repeat while possible" operator `_!` and they are broken when the strategy cannot be applied; the expression $X - 1$ is represented as `sd(X, 1)`; and the condition in the inner loop is separated in two matching conditions.

```
(stratdef INSERTION-SORT-STRAT is
  strat insort =
   setY ;
   ((match < PS , [ Y | X ] > s.t. Y <= length(PS)) ;
     setX ;
     (matchrew < PS , [ Y | X ] > s.t. X > 1 by
         PS using ((xmatch (sd(X, 1), V) (X, W) s.t. (V > W)) ;
                   switch[J <- sd(X, 1) ; I <- X]),
         [ Y | X ] using decX  ) ! ;
     incY
   ) ! .
endsd)
```

```
Maude> (srew < (1, 18) (2, 14) (3, 11)
        (4, 15) (5, 12), [ 0 | 0 ] > using insort .)
result Conf :
  < < (1, 11) (2, 12) (3, 14) (4, 15) (5, 18), [6 | 2] >, [6 | 2] >
```

6 Future work

We have described and illustrated by means of examples a first proposal for a strategy+search language for Maude, to be used at the object level (as opposed to the metalevel) to control the rewriting process. We have also presented a prototype implementation built over Full Maude using the metalevel and the metalanguage facilities provided by Maude. There is however much more work to do. To begin with, the current prototype has to be extended in order to implement the missing capabilities, such as breadth-first search and depth bounds. Also more examples need to be developed in order to validate the current proposal. Since this is still work in progress, it is now difficult to do a full comparison with other languages such as ELAN and Stratego, which will be considered in the future.

The current design of the strategy language could be extended by including new combinators; for example, congruence operators could be made available to the user, instead of having to simulate them by means of the `matchrew` combinator. Stratego also provides combinators for composing *generic traversals*. The operator `all(E)` applies the strategy `E` to each of the direct subterms `Ti` of a constructor application `C(T1,...,Tn)`. By using the `all` combinator generic traversals can be easily defined [12]:

```
bottomup(E)  =  all(bottomup(E)) ; E
topdown(E)   =  E ; all(topdown(E))
```

```
innermost(E) =  bottomup(try(E ; innermost(E)))
```

Our current language cannot simulate directly this combinator, since in order to use the `matchrew` combinator we need to know how to build the matching pattern. Once we know that the strategy `all(E)` is applied to term `C(T1,...,Tn)`, it is equivalent to the application of strategy

```
matchrew C(X1,...,Xn) by X1 using E, ..., Xn using E
```

Acknowledgements
We would like to thank the rest of the Maude team for all their comments on the strategy language design, Miguel Palomino for suggesting the blackboard game example and for testing our prototype, Isabel Pita for her help in the network management example, and the referees for all their very helpful comments.

References

[1] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185, 2002.

[2] P. Borovanský, C. Kirchner, H. Kirchner, and C. Ringeissen. Rewriting with strategies in ELAN: A functional semantics. *International Journal of Foundations of Computer Science*, 12:69–95, 2001.

[3] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications, 2000.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 Manual. Version 1.0*, June 2003. http://maude.cs.uiuc.edu/manual.

[5] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Sept. 1996. http://www.elsevier.nl/locate/entcs/volume4.html.

[6] P. C. Ölveczky and J. Meseguer. Specification of real-time and hybrid systems in rewriting logic. *Theoretical Computer Science*, 285(2):359–405, 2002.

[7] I. Pita. *Técnicas de especificación formal de sistemas orientados a objetos basadas en lógica de reescritura*. PhD thesis, Facultad de Matemáticas, Universidad Complutense de Madrid, 2003.

[8] I. Pita and N. Martí-Oliet. A Maude specification of an object-oriented model for telecommunication networks. *Theoretical Computer Science*, 285(2):407–439, 2002.

[9] A. Verdejo. *Maude como marco semántico ejecutable*. PhD thesis, Facultad de Informática, Universidad Complutense de Madrid, 2003.

[10] A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci and U. Montanari, editors, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19–21, 2002*, volume 71 of *Electronic Notes in Theoretical Computer Science*, pages 239–257. Elsevier, 2002.
http://www.elsevier.nl/locate/entcs/volume71.html.

[11] E. Visser. A survey of strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Workshop on Reduction Strategies in Rewriting and Programming (WRS'01)*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.
http://www.elsevier.nl/locate/entcs/volume57.html.

[12] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. Technical report, Institute of Information and Computing Sciences, Utrecht University, Nov. 2003.