Basic completion strategies as another application of the Maude strategy language

Alberto Verdejo   Narciso Martí-Oliet

Facultad de Informática
Universidad Complutense de Madrid
Madrid, Spain

`alberto@sip.ucm.es`   `narciso@esi.ucm.es`

The two levels of data and actions on those data provided by the separation between equations and rules in rewriting logic are completed by a third level of strategies to control the application of those actions. This level is implemented on top of Maude as a strategy language, which has been successfully used in a wide range of applications. First we summarize the Maude strategy language design and review some of its applications; then, we describe a new case study, namely the description of *completion procedures as transition rules + control*, as proposed by Lescanne.

1 Introduction

Strategies are pervasive in Computer Science; we have, among many others, control strategies, reduction strategies, deduction strategies, rewriting strategies, narrowing strategies, theorem-proving strategies, e-learning strategies, etc. This is just a reflection of the fact that strategies are essential ingredients in game and problem-solving. In general, in many settings we can identify three levels in the development of the solution for a given problem:

- definition of the data involved in the problem,

- identification of the basic actions that manipulate those data, and

- strategies to specify how those basic actions must be used to reach the desired solution.

For example, in the setting of business process modeling, those three levels correspond to the data (clients, reservations, money, etc.), the business activities or web services involved, and the composition of such activities or services to design more complex interactions through languages such as BPEL [12] and BPMN [13].

Since its introduction by Meseguer in the early nineties [39], rewriting logic addressed the separation between the first two levels above by distinguishing at the logic level equations from rules. Equations are used to define data (possibly including states), while rules are used to define transitions, activities, actions, and so on, that use data and allow to move from one state to another. Different specification languages are directly based on rewriting logic, including ELAN [9, 8] and Maude [20, 19]; in those languages, the distinction between equations and rules is emphasized by requiring, although both are implemented in terms of rewriting, equations to be confluent and terminating (and thus, any reduction strategy will give rise to the same unique result), while rules need not be either confluent or terminating. Then, if the user is interested in controlling the application of rules to avoid undesired directions, either the control is introduced into the rules in an ad hoc way depending on the problem, or it is necessary to introduce somehow the third level above to control the rule application by means of strategies. In the case of ELAN, this was part of its design, so that strategies become an essential part of the ELAN system,

which provides a basic set of strategies which can be used when writing rewrite rules, in such a way that at the specification level it is not enforced a separation between rules and strategies. On the other hand, in the case of Maude, for a while the introduction of explicit strategies was avoided by means of its direct access to the metalevel. Indeed, the Maude system provides `rewrite` commands for getting only an execution path, as well as a `search` command for exploring all possible execution paths from a starting term [19]; if one is interested in the results of only some execution paths satisfying some constraints, these can typically be specified at the metalevel, where both equations and rules become simply more data and can be manipulated in different ways. Even more, several strategy languages at the metalevel have been considered for different applications, such as, for example [21] for completion.

Taking into account our own previous experience designing strategy languages in Maude, and also from the experience of other languages like the already mentioned ELAN, TOM [7], and Stratego [47, 48] we decided to design a strategy language for Maude [36], to be used *at the object level* instead of at the metalevel, thus avoiding the need to know this more complex framework, and completing at the same time in the case of Maude the third level in problem solving as described above.

The Maude strategy language allows the definition of strategy expressions that control the way a term is rewritten. Differently from ELAN, our design was based on a strict separation between the rewrite rules and the strategy expressions, that are provided in separate modules. Thus, in our language it is not possible to use strategy expressions in the rewrite rules of a system module. 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. The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term, and allowing variables in a rule to be instantiated before its application by means of a substitution. For conditional rules, rewrite conditions can be controlled also by means of strategies. Basic strategies are combined by means of, among others, typical regular expression constructions (concatenation, union, and iteration), if-then-else, combinators to control the way subterms of a given term are rewritten, and recursion [36].

Since its proposal, the language has been successfully applied to a wide range of examples, from operational semantics representations to the formalization of web services; in particular, in the context of business process modelling that we have mentioned above, the Maude strategy language has been used to represent in Maude fragments of BPEL [38] and also of BPMN [25].

In the first part of this paper, after a quick introduction to Maude, we summarize the Maude strategy language design, by reviewing its syntax and set-theoretic semantics in Section 3, and then survey the main applications in Section 4. In the second part of the paper, Section 5, we present a new case study, namely the description of *completion procedures as transition rules + control*, as proposed by Lescanne in [32]. Equational systems are represented as data that is going to be manipulated by rewrite rules implementing the completion inference rules, following a well-known approach. In order to get a completion algorithm, one needs to apply these rules in a controlled way. In his paper, Lescanne does this by means of CAML programs, while our approach is more abstract, emphasizing the fact that inference rules do not change at all in the different algorithms.

2   Maude in a nutshell

In Maude the state of a system is formally specified as an algebraic data type by means of an equational specification. Maude uses a very expressive version of equational logic, namely *membership equational logic* [11]. In this kind of specifications we can define new types (by means of the keyword `sort`); subtype relations between types (`subsort`); operators (`op`) for building values of these types, giving

the types of their arguments and result, and which may have attributes as being associative (`assoc`) or commutative (`comm`), for example; equations (`eq`) that identify terms built with these operators; and memberships (`mb`) *t* : *s* stating that the term *t* has sort *s*. Both equations and memberships can be conditional. Conditions are formed by a conjunction (written $/\backslash$) of equations and memberships. Equations are assumed to be confluent and terminating, that is, we can use the equations from left to right to reduce a term *t* to a unique (modulo the operator attributes such as associativity or commutativity) canonical form *t′* that is equivalent to *t*, i.e. they represent the same value.

The *dynamic* behavior of a system is specified by rewrite rules of the form

$$l : t \longrightarrow t' \ if \ (\bigwedge_i u_i = v_i) \wedge (\bigwedge_j w_j : s_j) \wedge (\bigwedge_k p_k \longrightarrow q_k)$$

that describe the local, concurrent transitions of the system. That is, when part of a system matches the pattern *t* and the conditions are fulfilled, it can be transformed into the corresponding instance of the pattern *t′*.

2.1 Crossing the river

This example is taken from [20, Section 7.8], although the presentation here is slightly different.

A shepherd needs to transport to the other side of a river a wolf, a goat, and a cabbage. He has only a boat with room for the shepherd himself and another item. The problem is that in the absence of the shepherd the wolf would eat the goat, and the goat would eat the cabbage.

We represent with constants `left` and `right` the two sides of the river. The shepherd and his belongings are represented as objects with an attribute indicating the side of the river in which each is located and are grouped together with a multiset operator `__`. The rules represent how the wolf or the goat eat and the ways of crossing the river allowed by the capacity of the boat; an auxiliary `change` operation is used to modify the corresponding attributes.

```
mod RIVER-CROSSING is
  sorts Side Group .
  ops left right : -> Side .
  op change : Side -> Side .
  ops s w g c : Side -> Group .   --- shepherd, wolf, goat, cabbage
  op __ : Group Group -> Group [assoc comm] .

  vars S S' : Side .
  eq change(left) = right .
  eq change(right) = left .

  crl [wolf-eats] : w(S) g(S) s(S') => w(S) s(S') if S =/= S' .
  crl [goat-eats] : c(S) g(S) s(S') => g(S) s(S') if S =/= S' .
  rl [shepherd-alone] : s(S) => s(change(S)) .
  rl [wolf] : s(S) w(S) => s(change(S)) w(change(S)) .
  rl [goat] : s(S) g(S) => s(change(S)) g(change(S)) .
  rl [cabbage] : s(S) c(S) => s(change(S)) c(change(S)) .
endm
```

We want to know if there is a way the shepherd can safely take his belongings to the other side. But if we search if a state where everybody is on the right is reachable from a state where everybody is on the left, we cannot be sure that an intermediate state where, for example, the wolf has the posibility of eating but it has not eaten, is also reached. That is, with the rewrite or search commands of Maude we

cannot ensure that the priority of the first two rules is taken into account. In Section 3.9 we will present a strategy that ensures that priority.

3   The Maude strategy language

In this section we describe the syntax of our strategy language and its set-theoretic semantics. 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. If the strategy *fails* (it cannot be applied to the given term), the empty set of results is returned. Otherwise, we say that the strategy *succeeds* (possibly returning several results). For the strategies of a rewrite theory $\mathscr{R}$ with signature Σ we have a function

$$\_@\_ : Strat \times T_\Sigma(X) \longrightarrow \mathscr{P}(T_\Sigma(X)),$$

where $T_\Sigma(X)$ denotes the set of Σ-terms with variables in X. This function has an obvious extension to a function

$$\_@\_ : Strat \times \mathscr{P}(T_\Sigma(X)) \longrightarrow \mathscr{P}(T_\Sigma(X)),$$

where, if $\sigma \in Strat$ and $U \subseteq T_\Sigma(X)$, we have $\sigma @ U = \bigcup_{t \in U} \sigma @ t$.

3.1   Idle and fail

The simplest strategies are the constants `idle` and `fail`. The first always succeeds, but without modifying the term t to which it is applied, that is, `idle` $@ t = \{t\}$, while the second always fails, that is, `fail` $@ t = \emptyset$.

3.2   Basic strategies

The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term. Rule variables can be instantiated before its application by means of a substitution, that is, a mapping of variables to terms, so that the user has more control on the way the rule is applied. In case of conditional rules, the default breadth-first search strategy is used for checking the rewrites in the condition. Therefore, if l is a rule label, s a substitution, and t a term, the semantics of $l\,[s]\,@ t$ is the set of terms to which t rewrites in one step using the rule with label l instantiated by substitution s *anywhere* where it matches and satisfies the rule's condition. The substitution can be omitted if it is empty.

For conditional rules, rewrite conditions can be controlled by means of strategy expressions. A strategy expression of the form $l\,[s]\,\{\sigma_1 \ldots \sigma_n\}$ 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 $\sigma_1, \ldots, \sigma_n$ as strategy expressions to check the rewrites in the condition of l. The number of rewrite condition fragments appearing in the condition of rule l must be exactly n for the expression to be meaningful.

3.3   Top

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 only 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. `top(`β`)` applies the basic strategy β only at the top of a given state term. Note, however, that even applying a rule at the top is nondeterministic due to the possibility of multiple matches, because matching

takes place modulo the equational attributes of the operators, such as associativity, commutativity, or identity.

3.4 Tests

Tests are considered as strategies that check a property on a state, so that the strategy applied to a state is successful when the test is true on such a state, and the strategy fails when the test is false; moreover, in the first case the state is not changed. That is, for τ a test and t a term, $\tau @ t$ will evaluate to $\{t\}$ if τ succeeds on t, and to $\emptyset$ if it fails, so that τ acts as a filter on its input.

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. `amatch` ρ `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 ρ (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 fails otherwise. `match` ρ `s.t.` C corresponds to matching only at the *top*. When the condition C is simply `true`, it can be omitted.

3.5 Regular expressions

Basic strategies can be 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. The concatenation operator is associative and the union operator is associative and conmutative. This commutativity of union provides a form of nondeterminism in the way the solutions are found.

If σ, σ' are strategy expressions and t is a term, then $(\sigma \,;\, \sigma') @ t = \sigma' @ (\sigma @ t)$, $(\sigma \mid \sigma') @ t = (\sigma @ t) \cup (\sigma' @ t)$, and $\sigma \,\text{+} @ t = \bigcup_{i \geq 1} \sigma^i @ t$, where $\sigma^1 = \sigma$ and $\sigma^n = (\sigma \,;\, \sigma^{n-1})$ for $n > 1$. Of course, $\sigma \,\text{*} = \text{idle} \mid \sigma \,\text{+}$. For example, a strategy of the form $\sigma \,;\, \tau$ (with τ a test) will filter out all those results from σ that do not satisfy the test τ.

3.6 Conditional strategy and its derivations

Our next strategy combinator is a typical if-then-else, but generalized so that the first argument is also a strategy following ideas from Stratego [48] and ELAN [10].
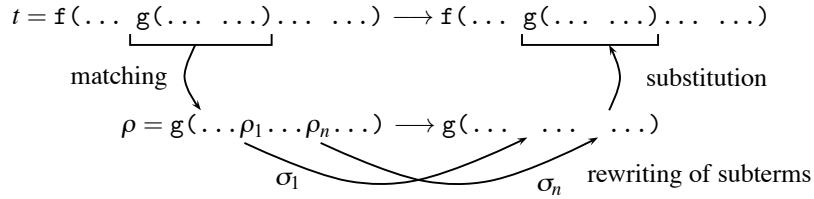
The behavior of the strategy expression $\sigma \,?\, \sigma' \,:\, \sigma''$ is as follows: in a given state term, the strategy σ is evaluated; if σ is successful, the strategy σ' is evaluated in the resulting states, otherwise σ'' is evaluated in the *initial* state. That is

$$(\sigma \,?\, \sigma' \,:\, \sigma'') @ t = \textbf{if } (\sigma @ t) \neq \emptyset \textbf{ then } \sigma' @ (\sigma @ t) \textbf{ else } \sigma'' @ t \textbf{ fi}.$$

Note that, as mentioned above, in general the first argument is a strategy expression and not just a test. Since a test is a strategy, we have the particular case $\tau \,?\, \sigma' \,:\, \sigma''$ for a test τ where evaluation coincides with the typical Boolean case distinction: σ' is evaluated when the test τ is true and σ'' when the test is false, taking into account that a test fails when false.

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

$$\sigma \text{ orelse } \sigma' = \sigma \,?\, \text{idle} \,:\, \sigma'$$

$$t = \mathtt{f(\ldots\ g(\ldots\ \ldots)\ldots\ \ldots)} \longrightarrow \mathtt{f(\ldots\ g(\ldots\ \ldots)\ldots\ \ldots)}$$

matching ( ) substitution

$$\rho = \mathtt{g(\ldots\rho_1\ldots\rho_n\ldots)} \longrightarrow \mathtt{g(\ldots\ \ \ldots\ \ \ldots)}$$

σ_1 σ_n rewriting of subterms

Figure 1: Behavior of the `amatchrew` combinator.

$\mathtt{not(\sigma)}$ reverses the result of evaluating σ, so that $\mathtt{not(\sigma)}$ fails when σ is successful and vice versa.

$$\mathtt{not(\sigma) = \sigma\ ?\ fail : idle}$$

An interesting use of $\mathtt{not(\sigma)}$ is the following "normalization" (or "repeat until the end") operation $\sigma\ \mathtt{!}$:

$$\mathtt{\sigma\ ! = \sigma\ *\ ;\ not(\sigma)}$$

$\mathtt{try(\sigma)}$ evaluates σ in a given state; if it is successful, the corresponding result is given, but if it fails, the initial state is returned.

$$\mathtt{try(\sigma) = \sigma\ ?\ idle : idle}$$

Evaluation of $\mathtt{test(\sigma)}$ checks the success/failure result of σ, but it does not change the given initial state.

$$\mathtt{test(\sigma) = not(\sigma)\ ?\ fail : idle}$$

Notice that $\mathtt{test(\sigma) = not(not(\sigma))}$.

3.7 Rewriting of subterms

With the previous combinators we cannot force the application of a strategy to a specific subterm of the given initial term. We can have more control over the way different subterms of a given state are rewritten by means of the `amatchrew` combinator.

When the strategy

$$\mathtt{amatchrew}\ \rho\ \mathtt{s.t.}\ C\ \mathtt{by}\ \rho_1\ \mathtt{using}\ \sigma_1, \ldots,\ \rho_n\ \mathtt{using}\ \sigma_n$$

is applied to a state term t, first a subterm of t that matches the pattern ρ and satisfies C is selected. Then, the patterns $\rho_1, \ldots, \rho_n$ (which must be disjoint subterms of ρ), instantiated appropriately, are rewritten as described by the strategy expressions $\sigma_1, \ldots, \sigma_n$, respectively. The results are combined in ρ and then substituted in t, in the way illustrated in Figure 1.

The strategy expressions $\sigma_1, \ldots, \sigma_n$ can make use of the variables instantiated in the matching, thus taking care of information extracted from the state term (see, for example, the strategies in Section 5.1).

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

The *congruence operators* used in ELAN and Stratego [10, 48] are special cases of the `matchrew` combinator, as shown in [36].

3.8   Strategy modules and commands

Given a Maude system module *M*, the user can write one or more strategy modules to define strategies for *M*. Such strategy modules have the following form:

```
smod STRAT is
  protecting M .
  including STRAT₁ .   ...      including STRATⱼ .
  strat sid₁ : T₁₁ ... T₁ₘ @ K₁ .
  sd sid₁(ρ₁₁,...,ρ₁ₘ) := σ₁ .
     ...
  strat sidₙ : Tₙ₁ ... Tₙₚ @ Kₙ .
  sd sidₙ(ρₙ₁,...,ρₙₚ) := σₙ .
  csd sidₙ(ρ'ₙ₁,...,ρ'ₙₚ) := σ'ₙ if C .
endsm
```

where *M* is the system module whose rewrites are being controlled, $STRAT_1,\ldots,STRAT_j$ are imported strategy submodules, $sid_1,\ldots,sid_n$ are identifiers, and $\sigma_1,\ldots,\sigma_n$ are strategy expressions (over the language of labels provided by *M*), where the identifiers can appear, thus allowing (mutually) recursive definitions. A strategy identifier can have *data arguments*, that are terms built with the syntax defined in the system module M. When a strategy identifier is declared (with the keyword `strat`), the types of its arguments (if any) are specified between the symbols `:` and `@`. After the symbol `@`, the type of the terms to which this strategy can be applied is also specified. A strategy definition (introduced with the keyword `sd`) associates a strategy expression (on the righthand side of the symbol `:=`) with a strategy identifier (on the lefthand side) with patterns as arguments, used to capture the values passed when the strategy is invoked. These strategy definitions can be conditional (with keyword `csd`). A strategy module can be *parametric* (see [23] for details).

3.9   Example

Here we show how to control by means of strategies the rewriting in module `RIVER-CROSSING` from Section 2.1. In [20] the first two rules were presented as equations in order to force Maude to apply them before any other rule if it is possible. But besides the fact that that solution introduced a coherence problem that had to be solved, it changed the semantics of the problem. Here we can guarantee the priority of these two rules by means of strategies. The `eating` strategy below performs all possible eatings; the `oneCrossing` strategy applies one of the other rules once; and the `allCE` strategy returns all the possible reachable states where eating has had the higher priority. That is, `allCE` ensures that when someone can eat, it eats for sure; we cannot recover from a disaster situation. Finally, if the strategy `solve` applied to the initial state (where we assume that all the objects are located on the left riverbank) returns a solution, it means that there is a safe way in which the shepherd can transport all his belongings to the other side of the river.

```
smod RIVER-CROSSING-STRAT is
  protecting RIVER-CROSSING .

  strat eating : @ Group .
  sd eating := (wolf-eats | goat-eats) ! .

  strat oneCrossing : @ Group .
  sd oneCrossing := shepherd-alone | wolf | goat | cabbage .
```

```
  strat allCE : @ Group .
  sd allCE := (eating ; oneCrossing) * .

  strat solve : @ Group .
  sd solve := allCE ; match (s(right) w(right) g(right) c(right)) .
endsm
```

3.10 Implementations of the strategy language

Using the metalevel mechanisms provided by the Maude system as a *metalanguage*, we have implemented a prototype of the Maude strategy language [36]. The metalevel features of Maude allow the definition of operations to work with modules and computations as objects, as is the case with strategies. The prototype works internally with a labelled version of the computation tree obtained by applying a strategy to a term. The prototype is completed with a user interface providing commands to load modules, execute strategy expressions on states, and show results. The prototype and some examples can be obtained from `http://maude.sip.ucm.es/strategies`.

After validating the language experimentally and reaching a mature language design, a direct implementation of our strategy language at the C++ level, at which the Maude system itself is implemented, is currently being developed [23]. This will make the language a stable new feature of Maude and will allow a more efficient execution.

In the meantime, we have advanced on the semantic foundations of the Maude strategy language in [37]. We have shown that a strategy language $\mathscr{S}$ can be seen as a rewrite theory transformation $\mathscr{R} \mapsto \mathscr{S}(\mathscr{R})$ such that $\mathscr{S}(\mathscr{R})$ provides a way of executing $\mathscr{R}$ in a controlled way. One such theory transformation for the Maude strategy language is presented in detail in [37], providing in this way a rewriting semantics for the strategy language; since this rewriting semantics is executable, we obtained a different metalevel implementation. We also studied in [37] some general requirements for strategy languages. Some of these requirements, like *soundness* and *completeness* with respect to the rewrites in $\mathscr{R}$, are absolute requirements that every strategy language should fulfill. Other more optional requirements, that we call *monotonicity* and *persistence*, represent the fact that no solution is ever lost. A future research direction is the *increased performance of strategy evaluations through parallelism*. The point is that in $\mathscr{S}(\mathscr{R})$ a term $\sigma @ t$ (where strategy σ is being applied to term t) incrementally evaluates to a (possibly nested) *set data structure*, so that the natural concurrency of rewriting logic is directly exploitable in $\mathscr{S}(\mathscr{R})$ by applying different rules in different places of this data structure where solutions are generated. This naturally suggests a distributed implementation of strategy languages.

4 Some applications

We briefly present in this section several research areas where Maude's strategy language has been applied successfully.

Operational semantics Rewriting logic and Maude are a very well-known semantic framework [35, 41]. By using strategies, the semantics representations can be made more simple and powerful, by separating the representation of the semantic rules from the mechanism used to control how they have to be applied. A simple example with Milner's CCS semantics is illustrated in [36].

In the *ambient calculus* [17] an *ambient* is a place limited by a boundary where computations take place. Its contents are a parallel composition of sequential processes and subambients; communication between processes is local, through a blackboard. The operational semantics for the ambient calculus consists of a set of structural congruence rules and a set of reduction rules, which can be represented in Maude by using strategies, as detailed in [42]. It gives us some congruence rules for free (as the commutativity and associativity of some operators), and the rest of the congruence rules are implemented as Maude equations. The reduction rules are then represented as rewrite rules in Maude. However, the reduction relation of the calculus is not a congruence for all the operators. This means that we cannot freely use the rewrite rules, as Maude would apply them anywhere in a term; and we do not want them to be applied after some operators. This is one of the reasons why the definition of a strategy that controls the application of these rules is necessary.

Eden [33] is a parallel extension of the functional language Haskell. On behalf of parallelism, Eden overrides Haskell's pure lazy approach, combining a non-strict functional application with eager process creation and communication. The operational semantics of Eden [26] is defined by means of a two-level transition system: the lower level handles local effects within processes, whereas the upper level describes the effects global to the whole system, like process creation and data communication. Then, the global evolution of the system is described by iteration of the scheduling rules and the sequential composition of other transitions. Thus the definition of the semantics itself imposes an order in the application of the semantic rules and the use of strategies is mandatory. The representation in Maude all of these rules and relations is studied in [28]. Most of the semantic rules are represented as rewrite rules and the transition relations that are defined as the concatenation or repetition of other relations are defined by means of strategies. But there are a few semantic rules that are more abstract in their mathematical formulation so they cannot be directly translated to rewrite rules. They are represented as (usually recursive) strategies that combine other strategies or rewrite rules. All of Eden's operational semantics has been represented at a quite abstract level, independently from factors such as the eagerness degree in the creation of new processes or the speculation degree. It was used to analyze algorithmic skeletons implemented in Eden in [27].

Modular structural operational semantics (MSOS) is a framework that allows structural operational semantics specifications to be made modular in the sense of not imposing the redefinition of transition rules when an extension is made. MSOS can be implemented in Maude in a quite precise way [14, 40]. The Maude MSOS Tool [18], an executable environment for modular structural operational semantics, has been endowed with the possibility of defining strategies over its transition rules, by combining the Maude MSOS Tool with the Maude strategy language in [15]. One advantage of this combination is the possibility of executing Ordered SOS specifications [46], including negative premises.

Membrane systems A membrane consists of a multiset w of objects, a set R of evolution rules (which are ordered by a priority relation), and a control mechanism C describing the way in which the rules are used to modify the multiset w in an evolution step. Control mechanisms can be given by maximally parallel rewriting, maximally parallel rewriting with priorities, and maximally parallel rewriting with promoters and/or inhibitors. In [1] it is shown how Maude can be used to specify membrane systems and how the control mechanisms in membranes can be described by using strategies. The strategy-based rewrite semantics thus defined preserves the maximal concurrency expressed by the maximal parallel application of the evolution rules [34]. This framework has been improved with the notion of *strategy controllers* [2], which allow to reason at the higher level of computation given by the evolution of the membrane systems. The intuition behind a strategy controller is that it decides which strategy is applied

in the current state. A prototype has been developed [2] by extending the implementation at the metalevel of Maude's strategy language [37].

Multi-agent systems   One of the challenges in the design and development of multi-agent systems is to coordinate and control the behavior of individual agents. There are different approaches, from low level ones (e.g., channel-based coordination) to high level ones (e.g., normative artifacts). These normative artifacts observe the actions performed by individual agents, determine their effects in the environment (which is shared by all individual agents), determine the violations caused by performing the actions, and possibly, impose sanctions. In [5, 3], the semantics of norm-based organization artifacts is specified by using Maude and its strategy language. Strategies are used as an alternative way to implement different normative artifacts without changing the semantics of the normative language. Thus, the normative multi-agent system is executed with respect to the transition rules which give the semantics of the normative language. However, how the system changes is described at an upper level by strategically instrumenting the transition rules. By using strategies there is a clear separation between executions and control. Timed choreographies are also considered in [4].

Other applications   Maude's strategy language has also been applied to solve sudoku puzzles in [44]; to formalize web services composition in [38]; to execute a rewriting logic representation of neural networks and the backpropagation learning algorithm in [45]; to present a rule-based approach for the design of dynamic software architectures in [16]; and to develop a specification of the connection method (a goal-directed proof procedure that requires a careful control over clause copies) for first-order logic in [29]. Of course, there are surely more examples we are not aware of yet.

5   Completion

A *completion procedure* is a method used in equational logic to build from a set of (unordered) identities E an equivalent canonical set of rewrite rules R, i.e. a confluent, noetherian and interreduced set of rules used to compute normal forms. A basic completion procedure can be given by firstly orienting the identities in E (using a provided reduction order on terms) and then iteratively computing all critical pairs of the rewrite system obtained so far and adding to it oriented versions of all the non-joinable ones. In order to avoid the huge number of rules that this basic procedure usually generates, rules can be simplified by reducing them with the help of other rules. Following Bachmair and Dershowitz [6], that method can be described by a set of inference rules (Figure 2) that covers a wide range of different specific completion procedures. A specific completion procedure is obtained from that set of rules by fixing a *strategy* for rule application.

The inference rules work on pairs (E, R) where E is a finite set of identities (input identities or critical pairs that have not yet been transformed into rules) and R is a finite set of terminating rewrite rules.[1] The goal is to transform an initial pair $(E_0, \emptyset)$ into a pair $(\emptyset, R)$ such that R is convergent and equivalent to E_0.

The inference rule DEDUCE derives an identity that is a direct consequence of rules in R, and adds it to E. A special, common case is adding a critical pair of R to E. The rule ORIENT takes an identity that can be ordered with the help of $>$ and adds the corresponding rule to R. The rule DELETE removes a trivial identity, and the rule SIMPLIFY-IDENTITY uses R to reduce identities. Both rules can be used together to remove joinable critical pairs. The rule R-SIMPLIFY-RULE reduces the righthand side of a

---

[1]Termination of R is ensured by a reduction order $>$ that is given as an input to the completion procedure.

$$\text{DEDUCE} \qquad \frac{E,R}{E \cup \{s \approx t\},R} \qquad \text{if } s \leftarrow_R u \rightarrow_R t$$

$$\text{ORIENT} \qquad \frac{E \cup \{s \approx t\},R}{E,R \cup \{s \rightarrow t\}} \qquad \text{if } s > t$$

$$\text{DELETE} \qquad \frac{E \cup \{s \approx s\},R}{E,R}$$

$$\text{SIMPLIFY-IDENTITY} \qquad \frac{E \cup \{s \approx t\},R}{E \cup \{u \approx t\},R} \qquad \text{if } s \rightarrow_R u$$

$$\text{R-SIMPLIFY-RULE} \qquad \frac{E,R \cup \{s \rightarrow t\}}{E,R \cup \{s \rightarrow u\}} \qquad \text{if } t \rightarrow_R u$$

$$\text{L-SIMPLIFY-RULE} \qquad \frac{E,R \cup \{s \rightarrow t\}}{E \cup \{u \approx t\},R} \qquad \text{if } s \xrightarrow{\beth}_R u$$

Figure 2: Inference rules for completion.

rule. Since, by assumption, termination of R can be shown using $>$, we know that $s \rightarrow_R t \rightarrow_R u$ implies $s > t > u$. For this reason, $s \rightarrow u$ can be kept as a rule. However, when reducing the lefthand side of a rule $s \rightarrow t$ to u, it is not clear whether $u > t$ is satisfied. Thus, the rule L-SIMPLIFY-RULE adds $u \approx t$ as an identity. The notation $s \xrightarrow{\beth}_R u$ is used to express that s is reduced by a rule $l \rightarrow r \in R$ such that l cannot be reduced by $s \rightarrow t$.

As mentioned above, specific completion procedures can be obtained from the inference rules in Figure 2 by fixing a strategy for rule application. In the following sections we show how to use Maude's strategy language to represent several completion procedures described by Lescanne in [32]. ELAN [8] has also been used to prototype completion algorithms in [31] by using strategies and constraints. Even Maude has already been used to represent Huet's completion algorithm [30], by using strategies defined at the metalevel in [21].

In [32] several algorithms for completion are presented using the functional programming language CAML. The main differences among the algorithms are the data structures on which the transition rules operate, and the control that describes the way the transitions rules are invoked. Here, we redo this work in Maude by using the proposed strategy language: the data structure is the term being rewritten, inference rules (transition rules in Lescanne's terminology) are represented as rewrite rules, and the control is represented as strategies applying the rules in a directed way. We consider three algorithms, as described by Lescanne: N-Completion, S-Completion, and ANS-Completion. Each algorithm is a refinement of the previous one, obtained by adding more components to the data structure, adapting the transition rules, and changing the control with the idea of making the algorithm more efficient. In this presentation we do not deal with the two unfailing completion algorithms also presented by Lescanne.

5.1 N-completion

N-completion is a first improvement of the overly abstract inference rules in order to take the computation of critical pairs into account. For this algorithm, the data structure has three components:

- *E* is a set of identities, either given identities or computed critical pairs,[2]

- *T* is a set of rules whose critical pairs have not been computed yet, and

- *R* is another set of rules whose critical pairs have already been computed (marked rules in Huet's terminology [30]).

The transition rules correspond to the inference rules but adapted to these three components. They are represented in Maude as follows:

```
mod N-COMPLETION is
  pr CRITICAL-PAIRS .

  sort System .
  op <_,_,_> : RlS RlS EqS -> System .  *** < R, T, E >

  var E : EqS .  var r : Rl .  vars R T : RlS .  vars s t u : Term .

  rl [Deduce] : < R, T, E > => < R, T, E s =. t > . *** if  s <-- u --> t

  crl [Orient] : < R, T, E s =. t > => < R, T s -> t, E >  if s > t .

  rl [Delete] : < R, T, E s =. s > => < R, T, E > .

  crl [Simplify] : < R, T, E s =. t > => < R, T, E u =. t >
   if u := reduce(s, R T) .

  crl [R-Simplify] : < R s -> t, T, E > => < R s -> u, T, E >
   if u := reduce(t, R T) .

  crl [R-Simplify] : < R, T s -> t, E > => < R, T s -> u, E >
   if u := reduce(t, R T) .

  crl [L-Simplify] : < R s -> t, T, E > => < R, T, E u =. t >
   if u := reduce>(s -> t, R T) .

  crl [L-Simplify] : < R, T s -> t, E > => < R, T, E u =. t >
   if u := reduce>(s -> t, R T) .

  rl [move] : < R, r T, E > => < r R, T, E > .
endm
```

The included module `CRITICAL-PAIRS` contains functions, defined at the metalevel, that compute critical pairs of a set of rules, or the reductions $\to_R$ (`reduce`) and $\overset{\exists}{\to}_R$ (`reduce>`). Observe that the inference rule R-SIMPLIFY-RULE gives rise to two rewrite rules: one where the righthand side of a rule in the set *R* is simplified and another one where the righthand side of a rule in *T* is simplified. The same

---

[2]The fact that an identity is an unordered pair is represented in Maude by using an operator `_=._` with the commutative attribute.

happens with L-SIMPLIFY-RULE. The rewrite rule `move` will be used by the strategy to move a rule from set *T* to set *R*.[3]

The algorithm N-completion has essentially three steps, namely *success*, when *T* and *E* are empty, *computing critical pairs* after simplification of the rules, when *E* is empty, and *orienting* an identity into a rule after simplification of the identities, when *E* is not empty. In the orientation part, it could happen that by simplification all the identities disappear. In this case, one does nothing, that is just translated by a recursive call. This algorithm can be succinctly expressed by the following strategy module (the strategy identifier declarations have been omitted):

```
smod N-COMPLETION-STRAT is
  protecting N-COMPLETION .

  sd N-COMP :=  success orelse deduce orelse orient .

  sd success := match < R, mtRlS, mtEqS > .

  sd deduce := match < R, r T, mtEqS > ;
               deduction ;
               simplify-rules ;
               N-COMP .

  sd deduction := matchrew < R, r' T, E > by
                      < R, r' T, E > using (add-crit-pairs(CP(r', R r')) ;
                                            move[r <- r']) .

  sd add-crit-pairs(mtEqS) := idle .
  sd add-crit-pairs(s1 =. t1 E) := Deduce[s <- s1 ; t <- t1] ; add-crit-pairs(E) .

  sd simplify-rules := (L-Simplify | R-Simplify) ! .

  sd orient := match < R, T, e E >  ;
               simplify-eqs ;
               ( (match < R, T, mtEqS > ; N-COMP)
                 orelse (Orient ; N-COMP) ) .

  sd simplify-eqs := (Delete | Simplify) ! .
endsm
```

5.1.1 Example

Let us consider the following set of equations

$$\{ \, g(x,y) \approx a, g(x,y) \approx h(x,y), h(x,y) \approx f(x), h(x,y) \approx f(y) \, \}$$

and the lexicographic path order induced by the precedence $g > h > f > a$. The basic completion procedure uses ORIENT to generate the rules

$$\{ \, g(x,y) \rightarrow a, g(x,y) \rightarrow h(x,y), h(x,y) \rightarrow f(x), h(x,y) \rightarrow f(y) \, \}$$

and then DEDUCE to compute the critical pairs $a \approx h(x,y)$ and $f(x) \approx f(y)$. We can reproduce this behavior by using some of the previous strategies

---

[3] That is because the only way a strategy can modify the term being rewritten is by means of rewrite rules.

```
Maude> (srew < mtRlS, mtRlS, eqs > using Orient ! .)
result System :
    < mtRlS, 'g['x:S,'y:S] -> 'a.S      'g['x:S,'y:S] -> 'h['x:S,'y:S]
            'h['x:S,'y:S] -> 'f['x:S] 'h['x:S,'y:S] -> 'f['y:S], mtEqS >

Maude> (cont using deduction ! ; Delete ! .)
result System :
    < 'g['x:S,'y:S] -> 'a.S       'g['x:S,'y:S] -> 'h['x:S,'y:S]
      'h['x:S,'y:S] -> 'f['x:S]  'h['x:S,'y:S] -> 'f['y:S],
    mtRlS, 'a.S =. 'h['x1:S,'y1:S]  'f['x1:S] =. 'f['y1:S] >
```

where `srew` is the command for applying a strategy to a given term; the constant `eqs` represents the above set of equations; `cont` is the command used to apply a strategy to the term returned by the previous application; and all the terms are metarepresented.

The basic completion procedure continues by trying to simplify and orient these critical pairs, but the terms in $f(x) \approx f(y)$ are irreducible and cannot be compared with any reduction order. So, the procedure fails. However, $a \approx h(x, y)$ can be oriented and used to compute new critical pairs that allow to reduce $f(x) \approx f(y)$ to a trivial identity. The N-completion procedure is able to find this solution.

```
Maude> (srew < mtRlS, mtRlS, eqs > using N-COMP .)
result System :
    < 'f['x:S] -> 'a.S  'g['x:S,'y:S] -> 'a.S  'h['x:S,'y:S] -> 'a.S, mtRlS, mtEqS >
```

5.2   S-completion

The main aim of orienting identities is to use them to simplify whenever it is possible. However, N-completion makes a bad use of simplification. S-completion is an improvement of N-completion where a rule is used for simplification as soon as it has been generated. When an identity is oriented into a rule, it enters a set *S* where it is used to simplify all the other identities and rules. Thus, the data structure has now four components:

- *E* is a set of identities, like in N-completion,

- *S* is a (singleton or empty) set of oriented identities (rules) that are used to simplify other rules,

- *T* is a set of rules already used for simplifying, but whose critical pairs have not been computed yet, and

- *R* is another set of rules whose critical pairs have already been computed, like in N-completion.

The only difference with the N-completion is the set *S* through which a rule has to go before entering *T*. The rewrite rules are modified to express this fact. We only show the rule that really changes (`Orient`) and a new rule that is used by the strategy to join the sets *T* and *S*. The rest of the rules are only modified by including the new set *S*.

```
mod S-COMPLETION is
  pr CRITICAL-PAIRS .
  sort System .
  op <_,_,_,_> : RlS RlS RlS EqS -> System .   *** < R, T, S, E >

  crl [Orient] : < R, T, S, E s =. t > => < R, T, S s -> t, E >  if s > t .

  rl [concatT&S] : < R, T, S, E > => < R, T S, mtRlS, E > .
```

```
  [...]
endm
```

The simplification step is clearly distinguished from the three others. It is performed when *S* is not empty. The completion process ends when there are no more identities or rules in *E*, *S* or *T*. Again, we only show the strategies that really change.

```
smod S-COMPLETION-STRAT is

  sd S-COMP :=  success orelse simplify-rules orelse deduce orelse orient .

  sd success := match < R, mtRlS, mtRlS, mtEqS > .

  sd simplify-rules := match < R, T, r S, E > ;
                       (L-Simplify | R-Simplify) ! ;
                       concatT&S ;
                       S-COMP .

  sd deduce := match < R, r T, mtRlS, mtEqS > ;
               deduction ;
               S-COMP .
  [...]
endsm
```

5.3 ANS-completion

S-completion computes all the critical pairs between all rules in *R* and one rule in *T*. In order to apply simplification as soon as possible, it is better to compute the critical pairs between one rule in *R* and one rule in *T* at a time. A new set *C* is created to contain one rule extracted from *T* with which critical pairs with rules in *R* are computed. To keep track of the rules whose critical pairs are computed with the rule in *C*, *R* is split into two sets, *A* (for already computed) and *N* (for not yet computed). The data structure has now six components:

- *E* is a set of identities, like in S-completion,

- *S* is a set of simplifying rules, like in S-completion,

- *T* is a set of rules coming from *S* and waiting to enter *C*,

- *C* is a set that contains at most one rule and whose critical pairs are computed with one in *N*,

- *N* is the part of *R* whose critical pairs have not been computed with *C* but whose critical pairs with $A \cup N$ have been computed, and

- *A* is a set whose critical pairs with $A \cup N \cup C$ have been computed.

The transition rules are trivially adapted to work with this new data structure, and two new rules are added. Rule AC2N joins the sets *A* and *C* with *N* and rule `fillC` extracts the smallest rule in *T* and puts it in *C*.

```
mod ANS-COMPLETION is
  pr CRITICAL-PAIRS .
```

```
  sort System .
  op <_,_,_,_,_,_> : RlS RlS RlS RlS RlS EqS -> System .   *** < A, N, C, T, S, E >

  [...]

  rl [AC2N] : < A, N, C, T, S, E > => < mtRlS, A N C, mtRlS, T, S, E > .

  crl [fillC] : < A, N, C, T, S, E > => < mtRlS, A N, r, T', S, E >
            if  r := least-rule(T) /\
                r T' := T .
endm
```

The procedure has now six parts, namely *success*, *simplification*, *orientation*, *deduction*, *internal deduction*, and *beginning of a new loop* of computation of critical pairs. *Deduction* computes the critical pairs between the smallest rule in *N* and the rule in *C*, whereas *internal deduction* computes the critical pairs obtained by superposing the rule in *C* on itself.

```
smod ANS-COMPLETION-STRAT is

  sd ANS-COMP :=  success             orelse
                  simplify-rules      orelse
                  orient              orelse
                  deduce              orelse
                  internal-deduction  orelse
                  new-loop .

  sd success := match ( < A, N, mtRlS, mtRlS, mtRlS, mtEqS > ) .

  sd deduce := match (< A, r N, r', T, mtRlS, mtEqS >) ;
               deduction ;
               ANS-COMP .

  sd deduction := matchrew < A, r' N, r'', T, S, E > s.t. r' := least-rule(r' N) by
                      < A, r' N, r'', T, S, E > using (add-crit-pairs(CP(r'', r')) ;
                                                     move[r <- r']) .

  sd internal-deduction := ( matchrew < A, mtRlS, r', T, mtRlS, mtEqS > by
                  < A, mtRlS, r', T, mtRlS, mtEqS > using (add-crit-pairs(CP(r', r')) ;
                                                        AC2N) ) ;
                           ANS-COMP .

  sd new-loop := fillC ; ANS-COMP .
  [...]
endsm
```

6   Concluding remarks

As another application of the Maude strategy language, we have described completion procedures as transition rules + control, following Lescanne's proposal [32]. Our version, using rewrite rules and declarative strategies instead of CAML programs, is more abstract and emphasizes the fact that inference rules do not change at all in the different algorithms.

This new case study on top of a growing list of applications throughout the world, has confirmed that the design of the Maude strategy language is good enough to handle a variety of algorithms based on the transition rules + control paradigm, using Lescanne's words, or simply data + actions + strategies, using the well-known three-level approach to problem solving.

As part of ongoing work, we are studying the integration of the strategies and model-checking features of Maude. Properties of rewriting systems expressed in linear temporal logic can be studied in Maude with the help of its integrated model checker [24, 20]. This tool checks the temporal properties fulfilled by a transition system by considering all the possible executions from a given state; however, as described in this paper, we can be interested in using the Maude strategy language to control those executions and possibly to restrict them, thus modifying the transition system. Thus, we plan to study how to model check temporal formulas satisfied by systems controlled by strategies, thus combining the advantages of two quite useful Maude features for the specification of systems: model checking at the property level and strategies at the execution level.

Acknowledgments

References

[1] Oana Andrei, Gabriel Ciobanu & Dorel Lucanu (2006): *Expressing Control Mechanisms of Membranes by Rewriting Strategies*. In Hendrik Jan Hoogeboom, Gheorghe Paun, Grzegorz Rozenberg & Arto Salomaa, editors: *Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers, Lecture Notes in Computer Science* 4361, Springer, pp. 154–169. Available at `http://dx.doi.org/10.1007/11963516_10`.

[2] Oana Andrei & Dorel Lucanu (2009): *Strategy-Based Proof Calculus for Membrane Systems*. In Roşu [43], pp. 23–43. Available at `http://dx.doi.org/10.1016/j.entcs.2009.05.011`.

[3] Lacramioara Astefanoaei, Frank S. de Boer & Mehdi Dastani (2009): *Rewriting Agent Societies Strategically*. In: *Proceedings of the 2009 IEEE/WIC/ACM International Conference on Web Intelligence and International Conference on Intelligent Agent Technology - Workshops, Milan, Italy, September 15-18, 2009*, IEEE, pp. 441–444. Available at `http://dx.doi.org/10.1109/WI-IAT.2009.321`.

[4] Lacramioara Astefanoaei, Frank S. de Boer & Mehdi Dastani (2010): *Strategic executions of choreographed timed normative multi-agent systems*. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lespérance, Michael Luck & Sandip Sen, editors: *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2010, Toronto, Canada, May 10-14, 2010, Volume 1-3*, IFAAMAS, pp. 965–972. Available at `http://doi.acm.org/10.1145/1838206.1838336`.

[5] Lacramioara Astefanoaei, Mehdi Dastani, John-Jules Ch. Meyer & Frank S. de Boer (2009): *On the Semantics and Verification of Normative Multi-Agent Systems*. Journal of Universal Computer Science 15(13), pp. 2629–2652. Available at `http://www.jucs.org/jucs_15_13/on_the_semantics_and`.

[6] Leo Bachmair & Nachum Dershowitz (1994): *Equational Inference, Canonical Proofs, and Proof Orderings*. Journal of the ACM 41(2), pp. 236–276. Available at `http://doi.acm.org/10.1145/174652.174655`.

[7] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau & Antoine Reilles (2006): *TOM Manual*. Available at `http://tom.loria.fr`.

[8] Peter Borovanský, Claude Kirchner, Hélène Kirchner & Pierre-Etienne Moreau (2002): *ELAN from a rewriting logic point of view*. Theoretical Computer Science 285(2), pp. 155–185.

[9] Peter Borovanský, Claude Kirchner, Hélène Kirchner & Christophe Ringeissen (2001): *Rewriting with Strategies in ELAN: A Functional Semantics*. International Journal of Foundations of Computer Science 12, pp. 69–95.

[10] Peter Borovanský, Claude Kirchner, Hélène Kirchner & Christophe Ringeissen (2001): *Rewriting with Strategies in ELAN: A Functional Semantics*. International Journal of Foundations of Computer Science 12(1), pp. 69–95. Available at `http://dx.doi.org/10.1142/S0129054101000412`.

[11] Adel Bouhoula, Jean-Pierre Jouannaud & José Meseguer (2000): *Specification and proof in membership equational logic*. Theoretical Computer Science 236(1-2), pp. 35–132. Available at `http://dx.doi.org/10.1016/S0304-3975(99)00206-6`.

[12] BPEL (2007): *Web Services Business Process Execution Language (WS-BPEL). Version 2.0*. OASIS Standard.

[13] BPMN (2011): *Business Process Model and Notation (BPMN). Version 2.0*. OMG Specification.

[14] Christiano Braga, Edward Hermann Haeusler, José Meseguer & Peter D. Mosses (2000): *Maude Action Tool: Using Reflection to Map Action Semantics to Rewriting Logic*. In Teodor Rus, editor: *Algebraic Methodology and Software Technology. 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings, Lecture Notes in Computer Science* 1816, Springer, pp. 407–421. Available at `http://dx.doi.org/10.1007/3-540-45499-3_29`.

[15] Christiano Braga & Alberto Verdejo (2007): *Modular Structural Operational Semantics with Strategies*. In Rob van Glabbeek & Peter D. Mosses, editors: *Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, August 26, 2006, Electronic Notes in Theoretical Computer Science* 175(1), Elsevier, pp. 3–17. Available at `http://dx.doi.org/10.1016/j.entcs.2006.10.024`.

[16] Roberto Bruni, Alberto Lluch-Lafuente & Ugo Montanari (2009): *Hierarchical Design Rewriting with Maude*. In Roşu [43], pp. 45–62. Available at `http://dx.doi.org/10.1016/j.entcs.2009.05.012`.

[17] Luca Cardelli & Andrew D. Gordon (1998): *Mobile Ambients*. In Maurice Nivat, editor: *Foundations of Software Science and Computation Structures, First International Conference, FoSSaCS'98 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'98 Lisbon, Portugal, March 28–April 4, 1998 Proceedings, Lecture Notes in Computer Science* 1378, Springer, pp. 140–155.

[18] Fabricio Chalub & Christiano Braga (2007): *Maude MSOS Tool*. In Denker & Talcott [22], pp. 133–146. Available at `http://dx.doi.org/10.1016/j.entcs.2007.06.012`.

[19] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn Talcott (2011): *Maude Manual (Version 2.6)*. Available at `http://maude.cs.uiuc.edu/maude2-manual`.

[20] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer & Carolyn L. Talcott (2007): *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*. Lecture Notes in Computer Science 4350, Springer. Available at `http://dx.doi.org/10.1007/978-3-540-71999-1`.

[21] Manuel Clavel & José Meseguer (1997): *Internal Strategies in a Reflective Logic*. In Bernhard Gramlich & Hélène Kirchner, editors: *Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction*, Townsville, Australia, pp. 1–12.

[22] Grit Denker & Carolyn Talcott, editors (2007): *Proceedings of the Sixth International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006. Electronic Notes in Theoretical Computer Science* 176(4), Elsevier.

[23] Steven Eker, Narciso Martí-Oliet, José Meseguer & Alberto Verdejo (2007): *Deduction, Strategies, and Rewriting*. In Myla Archer, Thierry Boy de la Tour & César Muñoz, editors: *Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006, Electronic Notes in Theoretical Computer Science* 174(11), Elsevier, pp. 3–25. Available at `http://dx.doi.org/10.1016/j.entcs.2006.03.017`.

[24] Steven Eker, José Meseguer & Ambarish Sridharanarayanan (2003): *The Maude LTL Model Checker and Its Implementation*. In Thomas Ball & Sriram K. Rajamani, editors: *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings, Lecture Notes in Computer Science* 2648, Springer, pp. 230–234. Available at `http://dx.doi.org/10.1007/3-540-44829-2_16`.

[25] Laura Henche (2009): *Introducción a la notación BPMN y su relación con las estrategias del lenguaje Maude*. Master's thesis, Facultad de Informática, Universidad Complutense de Madrid, Spain.

[26] Mercedes Hidalgo-Herrero & Yolanda Ortega-Mallén (2002): *An Operational Semantics for the Parallel Language Eden*. *Parallel Processing Letters* 12(2), pp. 211–228.

[27] Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén & Fernando Rubio (2005): *Towards Improving Skeletons in Eden*. In Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado & Emilio L. Zapata, editors: *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain, John von Neumann Institute for Computing Series* 33, Central Institute for Applied Mathematics, Jülich, Germany, pp. 843–850.

[28] Mercedes Hidalgo-Herrero, Alberto Verdejo & Yolanda Ortega-Mallén (2007): *Using Maude and Its Strategies for Defining a Framework for Analyzing Eden Semantics*. In Sergio Antoy, editor: *Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006, Electronic Notes in Theoretical Computer Science* 174(10), Elsevier, pp. 119–137. Available at `http://dx.doi.org/10.1016/j.entcs.2007.02.051`.

[29] Bjarne Holen, Einar Broch Johnsen & Arild Waaler (2009): *Proof Search for the First-Order Connection Calculus in Maude*. In Roşu [43], pp. 173–188. Available at `http://dx.doi.org/10.1016/j.entcs.2009.05.019`.

[30] Gérard P. Huet (1981): *A Complete Proof of Correctness of the Knuth-Bendix Completion Algorithm*. *Journal of Computer and System Sciences* 23(1), pp. 11–21. Available at `http://dx.doi.org/10.1016/0022-0000(81)90002-7`.

[31] Hélène Kirchner & Pierre-Etienne Moreau (1995): *Prototyping Completion with Constraints Using Computational Systems*. In Jieh Hsiang, editor: *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5-7, 1995, Proceedings, Lecture Notes in Computer Science* 914, Springer, pp. 438–443. Available at `http://dx.doi.org/10.1007/3-540-59200-8_79`.

[32] Pierre Lescanne (1989): *Completion Procedures as Transition Rules + Control*. In J. Díaz & F. Orejas, editors: *TAPSOFT'89 Proceedings of the International Joint Conference on Theory and Practice of Software Development, Barcelona, Spain, March 13-17, 1989, Lecture Notes in Computer Science* 351, Springer, pp. 28–41.

[33] Rita Loogen, Yolanda Ortega-Mallén & Ricardo Peña (2005): *Parallel Functional Programming in Eden*. *Journal of Functional Programming* 15(1), pp. 431–475.

[34] Dorel Lucanu (2009): *Strategy-Based Rewrite Semantics for Membrane Systems Preserves Maximal Concurrency of Evolution Rule Actions*. In Aart Middeldorp, editor: *Proceedings of the Eighth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2008, Castle of Hagenberg, Austria, July 14, 2008, Electronic Notes in Theoretical Computer Science* 237, Elsevier, pp. 107–125. Available at `http://dx.doi.org/10.1016/j.entcs.2009.03.038`.

[35] Narciso Martí-Oliet & José Meseguer (2002): *Rewriting logic as a logical and semantic framework*. In Dov M. Gabbay & Franz Guenthner, editors: *Handbook of Philosophical Logic, Second Edition, Volume 9*, Kluwer Academic Publishers, pp. 1–87.

[36] Narciso Martí-Oliet, José Meseguer & Alberto Verdejo (2004): *Towards a Strategy Language for Maude*. In Narciso Martí-Oliet, editor: *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004, Electronic Notes in Theoretical Computer Science* 117, Elsevier, pp. 417–441. Available at `http://dx.doi.org/10.1016/j.entcs.2004.06.020`.

[37] Narciso Martí-Oliet, José Meseguer & Alberto Verdejo (2009): *A Rewriting Semantics for Maude Strategies*. In Roşu [43], pp. 227–247. Available at `http://dx.doi.org/10.1016/j.entcs.2009.05.022`.

[38] Hamza Merouani, Farid Mokhati & Hassina Seridi-Bouchelaghem (2010): *Towards formalizing web service composition in Maude's strategy language*. In Ayman Alnsour & Shadi Aljawarneh, editors: *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications, ISWSA 2010, Amman, Jordan, June 14-16, 2010*, ACM, p. 15. Available at `http://doi.acm.org/10.1145/1874590.1874605`.

[39] José Meseguer (1992): *Conditional Rewriting Logic as a Unified Model of Concurrency*. Theoretical Computer Science 96(1), pp. 73–155. Available at `http://dx.doi.org/10.1016/0304-3975(92)90182-F`.

[40] José Meseguer & Christiano Braga (2004): *Modular Rewriting Semantics of Programming Languages*. In Charles Rattray, Savi Maharaj & Carron Shankland, editors: *Algebraic Methodology and Software Technology, 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, Lecture Notes in Computer Science 3116, Springer, pp. 364–378. Available at `http://dx.doi.org/10.1007/978-3-540-27815-3_29`.

[41] José Meseguer & Grigore Roşu (2007): *The rewriting logic semantics project*. Theoretical Computer Science 373(3), pp. 213–237. Available at `http://dx.doi.org/10.1016/j.tcs.2006.12.018`.

[42] Fernando Rosa-Velardo, Clara Segura & Alberto Verdejo (2006): *Typed Mobile Ambients in Maude*. In Horatiu Cirstea & Narciso Martí-Oliet, editors: *Proceedings of the 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan, April 23, 2005*, Electronic Notes in Theoretical Computer Science 147(1), Elsevier, pp. 135–161. Available at `http://dx.doi.org/10.1016/j.entcs.2005.06.041`.

[43] Grigore Roşu, editor (2009): *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*. Electronic Notes in Theoretical Computer Science 238(3), Elsevier.

[44] Gustavo Santos-García & Miguel Palomino (2007): *Solving Sudoku Puzzles with Rewriting Rules*. In Denker & Talcott [22], pp. 79–93. Available at `http://dx.doi.org/10.1016/j.entcs.2007.06.009`.

[45] Gustavo Santos-García, Miguel Palomino & Alberto Verdejo (2009): *Rewriting Logic Using Strategies for Neural Networks: An Implementation in Maude*. In Juan M. Corchado, Sara Rodríguez, James Llinas & José M. Molina, editors: *Proceedings of the International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008, University of Salamanca, Spain, October 22-24, 2008*, Advances in Soft Computing 50, Springer, pp. 424–433. Available at `http://dx.doi.org/10.1007/978-3-540-85863-8_50`.

[46] Irek Ulidowski & Iain Phillips (2002): *Ordered SOS Process Languages for Branching and Eager Bisimulations*. Information and Computation 178, pp. 180–213.

[47] Eelco Visser (2001): *Stratego: A Language for Program Transformation Based on Rewriting Strategies*. In Aart Middeldorp, editor: *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22-24, 2001, Proceedings*, Lecture Notes in Computer Science 2051, Springer, pp. 357–362.

[48] Eelco Visser (2004): *Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9*. In C. Lengauer, editor: *Domain-Specific Program Generation*, Lecture Notes in Computer Science 3016, Springer, pp. 216–238.