A Rewriting Semantics for Maude Strategies*

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

[a] *Facultad de Informática, Universidad Complutense de Madrid, Spain*

[b] *Department of Computer Science, University of Illinois at Urbana-Champaign, IL, USA*

Abstract

Intuitively, a strategy language is a way of taming the nondeterminism of a rewrite theory. We can think of a strategy language $\mathcal{S}$ as a rewrite theory transformation $\mathcal{R} \mapsto \mathcal{S}(\mathcal{R})$ such that $\mathcal{S}(\mathcal{R})$ provides a way of executing $\mathcal{R}$ in a controlled way. One such theory transformation for the Maude strategy language is presented in detail in this paper. Progress in the semantic foundations of Maude's strategy language has led us to study some general requirements for strategy languages. Some of these requirements, like *soundness* and *completeness* with respect to the rewrites in $\mathcal{R}$, 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. We show that the Maude strategy language satisfies all these four requirements.

Keywords: strategies, rewriting semantics, rewriting logic, Maude, ELAN

1 Introduction

This paper has two main goals: one more concrete, and another more general. Its first, more concrete goal is to advance the *semantic foundations* of the Maude strategy language, which has already been prototyped in Full Maude using reflection and the Maude `META-LEVEL` module [8], and whose built-in implementation in Core Maude is partially completed [7]. We have, furthermore, a second, more general goal in mind, namely, to articulate some general *requirements for strategy languages*. We think that such requirements may be useful in comparing different strategy languages, and in making design decisions about such languages. Although we discuss how the requirements apply to our language, we also discuss how ELAN's design [2,1] is related to them; indeed, this second level is completely general.

1.1 Semantic foundations of Maude's strategy language

Regarding our first goal, the specific way in which we advance the semantic foundations of Maude's strategy language in this paper is by giving a detailed *operational*

* Research partially supported by Spanish MEC project DESAFIOS (TIN2006–15660–C02–01), by CAM program PROMESAS (S–0505/TIC–0407), and by ONR grant N00014–02–1–0715.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

semantics by rewriting to Maude's strategy language. This is most fitting, since rewriting logic is a general semantic framework in which strategies themselves should not be thought of as extra-logical constructs, but should instead be expressed *inside the logic*. In our case, this takes the following concrete form. Given a rewrite theory specified as a Maude *system module* M, and given a *strategy module* SM in Maude's strategy language, where an algebra of strategy expressions for M is defined, we specify the generic construction of a rewrite theory $\mathcal{S}(M, SM)$, which defines the *operational semantics* of SM as a strategy module for M. Since this is done for any system module M and any associated strategy module SM, the mapping

$$(M, SM) \mapsto \mathcal{S}(M, SM)$$

provides indeed a general operational semantics by rewriting for Maude's strategy language. The reason for making the operational semantics depend not only on M but also on the strategy module SM is that Maude's strategy language, while providing for each system module M a core strategy language that we could denote as $SM_0(M)$, allows also user-defined *strategy definitions*, in which quite expresive recursive strategies can be defined by the user in *different* strategy modules SM. What all such strategy modules for M have in common, is that they contain the core strategy module $SM_0(M)$ without strategy definitions as a submodule.

The idea of giving an operational, rewriting semantics to a strategy language $\mathcal{S}$ by means of a theory transformation $\mathcal{R} \mapsto \mathcal{S}(\mathcal{R})$ is not new. It was formalized exactly in this way as the notion of an *internal strategy language* in [5], and has also been elegantly articulated in two papers describing in detail the rewriting semantics of ELAN's strategy language [2,1]. Furthermore, all the strategy languages developed experimentally for Maude since [5] (see, e.g., [5,6,8]) have indeed been defined by means of rewrite rules. What is new in this paper is that our rewriting semantics $(M, SM) \mapsto \mathcal{S}(M, SM)$ *is not explicitly reflective*. By contrast, all the semantics in [5,6,8] are reflective and extend Maude's `META-LEVEL` module. This is a conscious tradeoff. On the one hand, since a lot of infrastructure for matching, rewriting, and substitutions is provided *for free* by reflection in the `META-LEVEL` module, a reflective rewriting semantics is typically more succint. On the other hand, however, a reflective semantics relies heavily on a host of reflective constructs, and requires an explicit *change of representation* between terms and theories, and their corresponding metaterms and metatheories. Our choice of giving a not explicitly reflective semantics in this paper is motivated by two main goals: (i) the desire of making the basic ideas more accessible without assuming familiarity with reflective concepts; and (ii) making it easier to discuss general requirements for strategy languages without involving the orthogonal choice of a reflective vs. non-reflective semantics. The notion of a "reflective semantics" should be broadly understood as a semantics given in some (meta-)theory. In this sense, the more recent work in [3], giving semantics to strategies, including ELAN strategies, in the ρ-calculus can be understood as another flavor of a reflective semantics.

An important, second contribution of this work is an explicit comparison between the *operational* semantics of Maude's strategy language defined by the transformation $(M, SM) \mapsto \mathcal{S}(M, SM)$, and the *mathematical*, set-theoretic semantics of the language defined in [7] and recapitulated here in Section 3. If we compare this work to the so-called *functional semantics* of ELAN in [2], the main difference is that in

[2] what we here call the set-theoretic and the operational semantics of the strategy language are in a sense developed *together*. In our case, there are two main reasons for choosing to have two separate levels of semantics. The first is that the set of terms returned by a strategy applied to a term in our strategy language is in general an *infinite* set, which can only be viewed as an infinite *limit* of the operational semantics. The second reason is that several of the constructs in our language require the use of *tasks* and *continuations*, which are not themselves sets, but should instead be thought of as encapsulated, and possibly nested, *set generators*.

1.2 Requirements for strategy languages

At a more general level, we try to articulate in this work several general requirements on a strategy language that, to avoid being too abstract, are directly illustrated by means of the concrete strategy language that we present, but are, nevertheless, of general interest. Some similar requirements were discussed in [5]; but we think that it is worth revisiting this issue with the benefit of the richer experience about strategies for rewriting languages that has been collectively accumulated since that time. We want to emphasize that some of these requirements are *absolute* requirements; while others are very much a matter of *choice*. However, the second, more optional requirements have *important consequences* in a strategy language design and should therefore be a matter of *conscious* choice.

The paper is organized as follows. We begin with a more detailed discussion of such requirements in Section 2. Maude's strategy language and its set-theoretic abstract semantics are summarized in Section 3. We then define its operational, rewriting semantics in Section 4. The agreement of the mathematical and operational semantics and other related results are then treated in Section 5. Section 6 discusses some implementation issues, including the reflective implementation of the rewriting semantics; Section 7 gives a simple example illustrating the use of Maude's strategy language. Finally, we conclude the paper in Section 8.

2 Strategy language requirements

To facilitate our discussion we assume that given a term t in a rewrite theory $\mathcal{R}$, and given a strategy σ in the theory $\mathcal{S}(\mathcal{R})$ associated by the strategy language to $\mathcal{R}$, we can *apply* σ to t, which we denote by the expression $\sigma \mathbin{@} t$. We also assume that for any term w such that $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w$ one can define an abstract function *sols* such that $sols(w)$ denotes the set of *solution terms* already computed by the strategy; that is, terms reachable from the initial term that exactly satisfy the reachability requirements of the given strategy. The two most basic *absolute* requirements for any strategy language are then:

- **Soundness**. If $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w$ and $t' \in sols(w)$, then $\mathcal{R} \vdash t \longrightarrow^* t'$.
- **Completeness**. If $\mathcal{R} \vdash t \longrightarrow^* t'$ then there is a strategy σ in $\mathcal{S}(\mathcal{R})$ and a term w such that $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w$ and $t' \in sols(w)$.

We now discuss two, clearly optional requirements that a strategy language can satisfy. The first is what we call *determinism*. This imposes a natural requirement on the rewrite theory $\mathcal{S}(\mathcal{R})$ that the strategy language associates to $\mathcal{R}$. Intuitively,

one of the motivations for using a strategy language is to *control and tame* the *nondeterminism* of a theory $\mathcal{R}$, since in general its rules may not be confluent and may not terminate, so that many, wildly different executions are possible. If we may get quite different sets of answers when evaluating $\sigma \mathbin{@} t$, then there is still a residual nondeterminism left in the strategy language $\mathcal{S}(\mathcal{R})$. This is indeed a design choice. For example, the ELAN language is nondeterministic in this sense, because of its *don't care* nondeterministic operator *dc* [2,1]. The point is that the semantics of the *dc* operator is essentially [1] given by a conditional rule of the form

$$dc(\sigma_1, \ldots, \sigma_n) \mathbin{@} t \longrightarrow \sigma_i \mathbin{@} t \;\; \textit{if} \;\; \neg(\sigma_i \mathbin{@} t \longrightarrow \textit{fail})$$

where *fail* is a constant denoting failure, so that $\textit{sols}(\textit{fail}) = \emptyset$. This means that, since several of the strategies $\sigma_1, \ldots, \sigma_n$ may be nonfailing on t, when evaluating $dc(\sigma_1, \ldots, \sigma_n) \mathbin{@} t$ we may get completely different *sets* of answers, since the results of evaluating each $\sigma_i \mathbin{@} t$ is itself a set. The deterministic nature of a strategy language can be captured at the mathematical semantics level by the fact that, if we denote by $\textit{Strat}(\mathcal{R})$ the set of strategies defined in the theory $\mathcal{S}(\mathcal{R})$ associated by the given strategy language to a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, then the set-theoretic semantics of the strategy language can be defined by a function

$$[\![\_ \mathbin{@} \_]\!] : \textit{Strat}(\mathcal{R}) \times T_\Sigma(X) \longrightarrow \mathcal{P}(T_\Sigma(X)),$$

extended to a function

$$[\![\_ \mathbin{@} \_]\!] : \textit{Strat}(\mathcal{R}) \times \mathcal{P}(T_\Sigma(X)) \longrightarrow \mathcal{P}(T_\Sigma(X)),$$

in the expected pointwise way, namely, if $\sigma \in \textit{Strat}(\mathcal{R})$ and $U \subseteq T_\Sigma(X)$, we have $[\![\sigma \mathbin{@} U]\!] = \bigcup_{t \in U} [\![\sigma \mathbin{@} t]\!]$. Note that ELAN cannot be given a set-theoretic semantics in this way; instead, because of the *dc* operator, we need a semantic function of the form

$$[\![\_ \mathbin{@} \_]\!] : \textit{Strat}(\mathcal{R}) \times T_\Sigma(X) \longrightarrow \mathcal{P}(\mathcal{P}(T_\Sigma(X))),$$

since in general, the result of evaluating an application of the form $dc(\sigma_1, \ldots, \sigma_n) @ t$ is not a set of terms, but instead a *set of sets of terms.*

At the operational semantics level, determinism is captured by two minimal requirements that we call *monotonicity* and *persistence*. They intuitively mean that *no solution is ever lost.* These requirements can be made precise as follows:

- **Monotonicity**. If $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w$ and $\mathcal{S}(\mathcal{R}) \vdash w \longrightarrow^* w'$, then $\textit{sols}(w) \subseteq \textit{sols}(w')$.
- **Persistence**. If $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w$ and there exist terms w' and t' such that $\mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w'$ and $t' \in \textit{sols}(w')$, then there exists a term w'' such that $\mathcal{S}(\mathcal{R}) \vdash w \longrightarrow^* w''$ and $t' \in \textit{sols}(w'')$.

If these two requirements hold, then we have an alternative definition of the set-theoretic semantics as follows:

$$\{[\sigma \mathbin{@} t]\} = \{t' \in T_\Sigma(X) \mid \exists w. \mathcal{S}(\mathcal{R}) \vdash \sigma \mathbin{@} t \longrightarrow^* w \wedge t' \in \textit{sols}(w)\}.$$

[1] This is not exactly the way the semantics of *dc* is defined in [2,1], since there the results of $\sigma_i \mathbin{@} t$ are assumed to be available in the functional semantics. However, we believe that our formulation is essentially equivalent to that in [2,1].

Note that monotonicity and persistence imply that we can always find a sequence of one-step rewrites

$$\sigma \mathbin{@} t \longrightarrow w_1 \longrightarrow w_2 \longrightarrow \cdots w_n \longrightarrow \cdots$$

such that $\{[\![\sigma \mathbin{@} t]\!]\} = \bigcup_{n \in \mathbb{N}} sols(w_n)$.

A second, optional requirement is what we call the *separation* between the rewriting language itself and its associated strategy language. In Maude's strategy language design this means that a Maude system module M never contains any strategy annotations. Instead, all strategy information is contained in *strategy modules* of the form SM, which linguistically constitute a completely different part of the language, although they of course import the system module M whose execution the strategies in SM control. This strict division of labor between M and SM has some advantages. On the one hand, the strictly declarative nature of M is preserved. On the other, this separation facilitates modularity, since the same system module M can have many different strategy modules SM for different purposes. In ELAN, since it is possible to include strategy expressions within a rewrite rule definition, this separation is not enforced by the language. However, we believe that it is possible to develop ELAN modules following the separation methodology that we advocate here.

3 Maude strategies and their set-theoretic semantics

In this section we summarize the combinators of our strategy language and their set-theoretic semantics. Although we have benefitted from our own previous experience designing strategy languages in Maude, our language is also influenced by other strategy languages like ELAN [2,1] and Stratego [10]. However, Maude's strategy language is deterministic in the precise sense specified in Section 2. Of course, given a Maude system module M specifying a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, there is not a fixed set $Strat(\mathcal{R})$. Instead, for each strategy module SM for M we have a set of strategies $Strat(M, SM)$. Given then a term $t \in T_\Sigma(X)$ and a strategy $\sigma \in Strat(M, SM)$, the abstract set-theoretic semantics defines a set $[\![\sigma @ t]\!] \in \mathcal{P}(T_\Sigma(X))$.

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, $[\![\texttt{idle} \mathbin{@} t]\!] = \{t\}$, while the second always fails, that is, $[\![\texttt{fail} \mathbin{@} t]\!] = \emptyset$.

3.2 Basic strategies

The basic strategies consist of the application of a rule (identified by the corresponding rule label) to a given term. 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. Therefore, if l is a rule label and t a term, $[\![l \mathbin{@} t]\!]$ is the set of terms to which t rewrites in one step using the rule with label l anywhere where it matches and satisfies the rule's condition.

A slightly more general variant allows variables in a rule to 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. The unconstrained case becomes then the particular case in which the substitution is the identity.

For conditional rules, rewrite conditions can be controlled by means of strategy expressions. As before, the substitution can be omitted if it is empty. A strategy expression of the form `L[S]{E1 ... En}` 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 `E1`, ..., `En` 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(BE)` applies the basic strategy `BE` only at the top of a given state term.

3.4 Tests

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. That is, for T a test and t a term, $[\![T \mathbin{@} t]\!]$ is equal to the singleton $\{t\}$ if T succeeds on t, and to $\emptyset$ if it fails, so that T 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 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 fails otherwise. `match T 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 commutative. If E, E' are strategy expressions and t is a term, then $[\![(E\ ;\ E') \mathbin{@} t]\!] = [\![E' \mathbin{@} [\![E \mathbin{@} t]\!]]\!]$, $[\![(E \mid E') \mathbin{@} t]\!] = [\![E \mathbin{@} t]\!] \cup [\![E' \mathbin{@} t]\!]$, and $[\![E\,\texttt{+} \mathbin{@} t]\!] = \bigcup_{i \geq 1} [\![E^i \mathbin{@} t]\!]$, where $E^1 = E$ and $E^n = (E\ ;\ E^{n-1})$ for $n > 1$. Of course, $E\,\texttt{*} = \texttt{idle} \mid E\,\texttt{+}$.

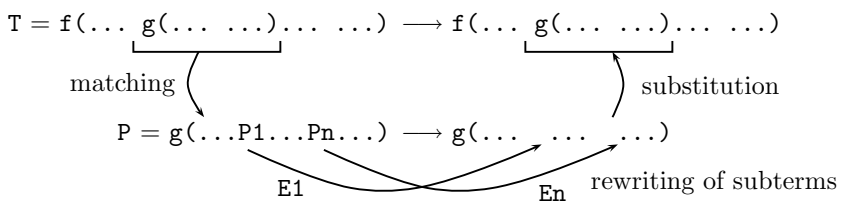

Fig. 1. Behavior of the `amatchrew` combinator.

3.6 Conditional 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 [10], but it also appears in ELAN [2, Example 5.2].

The behavior of the strategy expression `E ? E' : E''` 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. That is, by definition, this combinator satisfies the equation

$$[\![(E \ ? \ E' \ : \ E'') \mathbin{@} t]\!] = \textbf{if } [\![E \mathbin{@} t]\!] \neq \emptyset \textbf{ then } [\![E' \mathbin{@} [\![E \mathbin{@} t]\!]]\!] \textbf{ else } [\![E'' \mathbin{@} t]\!] \textbf{ fi}.$$

As explained in [10,8], using the if-then-else combinator, we can define many other useful strategy combinators as derived operations such as, for example,

```
E orelse E' = E ? idle : E'
not(E) = E ? fail : idle
E ! = E * ; not(E)
try(E) = E ? idle : idle
```

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. In particular, the scope of the substitution in the (a)`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 (a)`matchrew` combinators. When the strategy expression

```
amatchrew P s.t. C by P1 using E1, ..., Pn using En
```

is applied to a state term `T`, first a subterm of `T` that matches `P` and satisfies `C` is selected. Then, the terms `P1`, ..., `Pn` (which must be disjoint subterms of `P`), instantiated appropriately, are rewritten as described by the strategy expressions `E1`, ..., `En`, respectively. The results are combined in `P` and then substituted in `T`, in the way illustrated in Figure 1. The strategy expressions `E1`, ..., `En` can make use of the variables instantiated in the matching, thus taking care of information extracted from the state term.

The version `matchrew` works in the same way, but performing matching only at the top. In all cases, when the condition is `true` it can be omitted. The *congruence operators* used in ELAN and Stratego [2,10] are special cases of the `matchrew` combinator, as shown in [8].

3.8 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 modules, described next.

3.9 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 E1 : T11 ... T1m @ K1 .
  sd E1(P11,...,P1m) := Exp1 .
     ...
  strat En : Tn1 ... Tnp @ Kn .
  sd En(Pn1,...,Pnp) := Expn .
  csd En(Qn1,...,Qnp) := Expn' if C .
     ...
endsd
```

where `M` is the system module whose rewrites are being controlled, `STRAT1`, ..., `STRATj` are imported strategy submodules, `E1`, ..., `En` are identifiers, and `Exp1`, ..., `Expn` are strategy expressions (over the language of labels provided by `M`), where the identifiers can appear, thus allowing (mutually) recursive definitions. The basic idea is that these strategy declarations provide useful abbreviations for strategy expressions `E` that the user can then utilize in a (strategy rewrite) command `srew T using E`, which rewrites a term `T` using a strategy expression `E`.

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`), and there can be several strategy definitions defining different cases of a given strategy E. A concrete example appears in Section 7.

4 Rewriting semantics of strategies

We assume given a rewrite theory in the form of a Maude system module M, and an associated strategy module SM defining strategies for M. The system module M includes both equations and rules. We can assume without loss of generality that the variables used in equations are disjoint from the variables used in rules. In this section we describe how to transform the pair (M, SM) into a rewrite theory $\mathcal{S}(M, SM)$ where we can write strategy expressions and apply them to terms from M, so that rewriting a term of the form $E \mathbin{@} t$ produces the results of rewriting the

term t by means of the strategy E.

4.1 Auxiliary operations

The transformed theory $\mathcal{S}(M, SM)$ is an extension of the equational part of the system module M. Rules in M are replaced by rules that apply appropriate strategies. The terms of the module M become data for the strategy expressions in $\mathcal{S}(M, SM)$. On such terms we will need to perform some typical operations like matching and applying a substitution to a term. For this we need some auxiliary infrastructure:

- All the variables used in rewrite rules in M become constants in $\mathcal{S}(M, SM)$; more specifically, each variable X of sort S becomes a constant X of sort `Var`S, which is a new subsort of S.
- All the labels in rewrite rules in M become constants of a new sort `Label` in $\mathcal{S}(M, SM)$.
- We add syntax for *substitutions*, which are given by sets of assignments defined by means of an overloaded family of operators

  ```
  op _<-_ : VarS S -> Substitution .
  ```

  for each sort S in M, as well as a union operator

  ```
  op _,_ : Substitution Substitution -> Substitution [assoc id: none] .
  ```

  Applying a substitution is done by means of an overloaded application operator

  ```
  op _._ : S Substitution -> S .
  ```

  which is equationally defined in the standard way.
- We also add a special "hole" constant for each sort S in M,

  ```
  op []S : -> S .
  ```

  so that terms built with this constant are *contexts*. The following partial operation is used to replace the hole in a context with a term of the appropriate sort:

  ```
  op replace : S S' ~> S .
  ```
- Finally, we add a couple of overloaded *matching* operators that given a pair of terms return the set of matches, either at the top or anywhere, that satisfy a given condition. A match consists of a pair formed by a substitution and a context; when matching is at the top, the context is simply the hole.

  ```
  sorts Match MatchSet .
  subsort Match < MatchSet .
  op <_,_> : Substitution S -> Match .
  op __ : MatchSet MatchSet -> MatchSet [assoc comm id: none] .
  op getMatch : S S Condition -> MatchSet .
  op getAmatch : S S Condition -> MatchSet .
  ```

  Notice that the equations defining the semantics of substitutions and matching depend on the signature of the module M. For example, if M includes operators `f : S -> S` and `g : S S -> S`, the definition of the matching operations would include equations such as

  ```
  eq getMatch(f(T:S), g(T':S, T'':S)) = none .
  eq getAmatch(f(T:S), g(T':S, T'':S)) =
       expand-context(getAmatch(T:S, g(T':S, T'':S)), f([]S)) .
  ```

  where `expand-context` is an auxiliary operation that adapts all contexts obtained

in the recursive call to the bigger initial pattern.

4.2 Strategy syntax

The abstract syntax of the strategy language summarized in Section 3 can be given by a signature represented in Maude as follows:

```
sorts BasicStrat Strat Test StratCall .
subsorts BasicStrat Test StratCall < Strat .
sorts TermStrat TermStratList .
subsort TermStrat < TermStratList .

op _[_] : Label Substitution -> BasicStrat .
op _[_]{_} : Label Substitution StratList -> BasicStrat .
op match : S Condition -> Test .
op amatch : S Condition -> Test .
ops idle fail : -> Strat .
op top : BasicStrat -> Strat .
op _|_ : Strat Strat -> Strat [assoc comm id: fail] .
op _;_ : Strat Strat -> Strat [assoc] .
op if : Strat Strat Strat -> Strat .
op _* : Strat -> Strat .
op _+ : Strat -> Strat .
op _! : Strat -> Strat .
op matchrew : S Condition TermStratList -> Strat .
op amatchrew : S Condition TermStratList -> Strat .

op _using_ : S Strat -> TermStrat .
op nilTSL : -> TermStratList .
op _,_ : TermStratList TermStratList -> TermStratList [assoc id: nilTSL] .

sort StratList .
subsort Strat < StratList .
op nil : -> StratList .
op __ : StratList StratList -> StratList [assoc id: nil] .
```

Notice that the operators `match`, `amatch`, `matchrew`, and `amatchrew` are provided as overloaded families of operators for all sorts S in M.

All this syntax is added to the transformed theory $\mathcal{S}(M, SM)$. Moreover, the operator $\_\cdot\_$ that applies a substitution to a term is extended to strategy expressions in the standard way.

4.3 Applying strategies

The following sorts and operators provide the main infrastructure defined in the transformed theory $\mathcal{S}(M, SM)$ in order to define the application of a strategy to a term. Again several operators are provided as overloaded families of operators for all sorts S in M.

```
sorts Task Tasks Cont .
subsort Task < Tasks .
op none : -> Tasks .
op __ : Tasks Tasks -> Tasks [assoc comm id: none] .
eq T:Task T:Task = T:Task .

op <_@_> : Strat S -> Task .
op sol :  S -> Task .
op <_;_> : Tasks Cont -> Task .
```

```
op chkrw : RewriteList Condition StratList S S' ->  Cont .
op seq : Strat -> Cont .
op ifc : Strat Strat Term -> Cont .
op mrew : S Condition TermStratList S' -> Cont .

sort RewriteList .
op nilRL : -> RewriteList .
op _=>_  : S S -> RewriteList .
op _/\_ : RewriteList RewriteList -> RewriteList [assoc id:nilRL] .
```

Applying a strategy to a term is a *task* that, in the process of rewriting, can give rise to more tasks. Therefore, the sort `Tasks` represents *sets of tasks* by means of an associative, commutative, and idempotent operator written as juxtaposition, with identity the empty set `none`. Solved tasks are of the form `sol`(t), meaning that the term t is a *solution*. Sometimes, these solutions are only the intermediate results in a task that must be continued with another process; therefore, there is also some syntax to represent *continuations* as terms of sort `Cont`.

4.4 Strategy rewrite rules

4.4.1 Idle and fail

For each sort S in the system module M, we add rules

```
rl < idle @ T:S > => sol(T:S) .
rl < fail @ T:S > => none .
```

We recall that the notation `T` : S means that `T` is a Maude variable of sort S. Since all the operators and variables must be appropriately typed, we need to repeat operator declarations and rules for each sort.

4.4.2 Basic strategies

For each nonconditional rule `[l] : t1 => t2` and each sort S in M, we add the following rule that collects by means of the `getAmatch` operation all the possible matches for the lefthand side pattern in the given state term and then returns the set of all appropriate instantiations of the righthand side. The `gen-sols` operation is used to traverse the set of matches and for each pair of substitution and context, it builds the appropriate instantiation by first applying the substitution to the righthand side and then replacing the hole in the context with the resulting term.

```
crl < l[Sb] @ T:S > => gen-sols(MAT, t2·Sb)
 if MAT := getAmatch(t1·Sb, T:S, trueC) .

eq gen-sols(none, T:S') = none .
eq gen-sols(< Sb, Cx:S > MAT, T:S') =
     sol(replace(Cx:S, T:S'·Sb)) gen-sols(MAT, T:S') .
```

Notice that `Sb` is a variable of sort `Substitution`. In particular, `Sb` can be instantiated to the identity substitution.

The treatment of conditional rules is much more complex, because we need to make sure that each solution satisfies the rewrites in the condition using the appropriate strategies. Moreover, we need to guarantee that no solution exists when returning the empty set. For these reasons, we make use of a continuation

that handles this situation. Let us consider a conditional rule of the form

```
crl [l] : t1 => t2 if u1 => v1 /\.../\ un => vn /\ C .
```

Notice that we have put together all the fragments of the condition that are not rewrites at the end, in order to simplify the presentation. The first rule is used to get all the matches with the lefthand side and for each one we instantiate the rule conditions and call the appropriate continuation to check that the condition is really satisfied. The `gen-rw-tks` operation is used to traverse the set of matches and generate for each match the appropriate task, as shown in the second equation below.

```
crl < l[Sb]{E1...En} @ T:S >
 => gen-rw-tks(MAT, E1, u1·Sb, (u1 => v1 /\.../\ un => vn)·Sb, C·Sb, E1...En, t2·Sb)
 if MAT := getAmatch(t1·Sb, T:S, trueC) .

eq gen-rw-tks(none, E, T:S', RL, C, EL, Rhs:S'') = none .
eq gen-rw-tks(< Sb, Cx:S > MAT, E, T:S', RL, C, EL, Rhs:S'') =
     < < E @ T:S'·Sb > ; chkrw(RL·Sb, C·Sb, EL, Rhs:S''·Sb, Cx:S) >
     gen-rw-tks(MAT, E, T:S', RL, C, EL, Rhs:S'') .
```

The `chkrw` continuation uses the strategy list to try to satisfy the rewrites in the conditions. For each partial solution obtained up to the moment the continuation is called, it checks whether the first rewrite in the first argument is satisfied and then calls the next strategy with the next rewrite adequately instantiated by the previous substitution. However, each rewrite condition may also be satisfied in different ways, giving rise to different substitutions and hence different solutions. Therefore, we also need to handle the set of possible matches (at the top) by means of another operation `gen-rw-tks2` that traverses this set and generates all the corresponding continuation tasks.

```
var TS : Tasks .  var RL : RewriteList .  var EL : StrategyList .
crl < sol(R:S) TS ;
      chkrw(U:S => V:S /\ U':S' => V':S' /\ RL, C, (E E' EL), Rhs:S'', Cx:S''') >
 => < TS ; chkrw(U:S => V:S /\ U':S' => V':S'  /\ RL, C, (E E' EL), Rhs:S'', Cx:S''') >
    gen-rw-tks2(MAT, E', U':S', (U':S' => V':S' /\ RL), C, (E' EL), Rhs:S'', Cx:S''')
 if MAT := getMatch(V:S, R:S, trueC) .

eq gen-rw-tks2(none, E, T:S', RL, C, EL, Rhs:S'', Cx:S''') = none .
eq gen-rw-tks2(< Sb, Cx:S > MAT, E, T:S', RL, C, EL, Rhs:S'', Cx:S''') =
     < < E @ T:S'·Sb > ; chkrw(RL·Sb, C·Sb, EL, Rhs:S''·Sb, Cx:S''') >
     gen-rw-tks2(MAT, E, T:S', RL, C, EL, Rhs:S'', Cx:S''') .
```

When the last rewrite is reached, the continuation also checks that the non-rewrite condition `C` is satisfied as part of the matching. Again, different matches are possible, which are collected in a set traversed by the operation `gen-sols2`, in charge of generating all the solutions by instantiating the righthand side of the conditional rule and putting the result in the original context.

```
crl < sol(R:S) TS ; chkrw(U:S => V:S, C, E, Rhs:S', Cx:S'') >
 => < TS ; chkrw(U:S => V:S, C, E, Rhs:S', Cx:S'') >
    gen-sols2(MAT, Rhs:S', Cx:S'')
 if MAT := getMatch(V:S, R:S, C) .

eq gen-sols2(none, Rhs:S, Cx:S') = none .
eq gen-sols2(< Sb, Cx':S'' > MAT, Rhs:S, Cx:S')
 = sol(replace(Cx:S', Rhs:S·Sb)) gen-sols2(MAT, Rhs:S, Cx:S') .
```

```
rl < none ; chkrw(RL, C, EL, Rhs:S')) > => none .
```

4.4.3 Top

For a basic strategy `BE`, the strategy `top(BE)` is defined using the same rules as for `BE`, except that matching takes place only at the top instead of anywhere. Therefore, the corresponding rules are obtained from the rules for basic strategies in the previous section by using the `getMatch` operation instead of `getAmatch`.

4.4.4 Tests

Application of a matching test is directly based on the `getMatch` and `getAmatch` operations, as expected, depending on whether the match is only at the top or everywhere. When the set of matches is empty (the constant `none`), the test fails and hence returns the empty set of solutions; when the set of matches is not empty, the result consists of only a solution which coincides with the initial state term.

```
crl < match(P:S, C) @ T:S' > => sol(T:S')
 if < Sb, Cx:S' > MAT := getMatch(P:S, T:S', C) .

crl < match(P:S, C) @ T:S' > => none
 if getMatch(P:S, T:S', C) = none .

crl < amatch(P:S, C) @ T:S' > => sol(T:S')
 if < Sb, Cx:S' > MAT := getAmatch(P:S, T:S', C) .

crl < amatch(P:S, C) @ T:S' > => none
 if getAmatch(P:S, T:S', C) = none .
```

4.4.5 Regular expressions

The rewriting semantics of the union combinator on strategies is based on the union operator on sets of solutions. Concatenation uses another continuation, while iteration is reduced to concatenation. In this version, we consider E `*` as a primitive, while E `+` is defined by means of a simple equation.

```
rl < E | E' @ T:S > => < E @ T:S > < E' @ T:S > .
rl < E ; E' @ T:S > => < < E @ T:S > ; seq(E') > .
rl < sol(R:S) TS ; seq(E') > => < E' @ R:S > < TS ; seq(E') > .
rl < none ; seq(E') > => none .
rl < E * @ T:S > => sol(T:S) < E ; (E *) @ T:S > .
eq E + = E ; E *  .
```

4.4.6 If-then-else

The conditional strategy combinator is easily defined by means of a continuation that "remembers" the remaining arguments while the strategy given as first argument is executed. If solutions to this strategy are found, then the continuation can discard the third argument, but the second argument must be applied to all solutions; otherwise, the second argument is discarded and the third is applied only to the original state term.

```
rl < if(E, E', E'') @ T:S > => < < E @ T:S > ; ifc(E', E'', T:S) > .
rl < sol(R:S) TS ; ifc(E', E'', T:S') > => < E' @ R:S > < TS ; seq(E') > .
rl < none ; ifc(E', E'', T:S) > => < E'' @ T:S > .
```

4.4.7 Rewriting of subterms

Matching and rewriting subterms is again a complex process that is best described using another continuation. In order to simplify the presentation of the following rules, we assume that in a strategy expression of the form

```
(a)matchrew P s.t. C by P1 using E1, ..., Pn using En
```

the pattern fragments `P1`, ..., `Pn` (besides being disjoint) appear once in pattern `P`.

The first rule generates all the matches of the given pattern in the state term and for each one we call the appropriate continuation to rewrite the corresponding fragments using the given strategies. The `gen-mrew` operation is used to traverse the set of matches and generate for each match the appropriate task, as shown in the second equation below. In this second equation we call again the `getAmatch` operation to locate the subpattern `P1` (already instantiated) inside the pattern `P` (also instantiated with the same substitution), but the result of this call must necessarily be the identity substitution with a unique context, because of the simplifying assumption we have mentioned before.

```
crl < amatchrew(P:S, C, TSL) @ T:S' > => gen-mrew(MAT, P:S, TSL)
 if MAT := getAmatch(P:S, T:S', C) .

eq gen-mrew(none, P:S, TSL) = none .
ceq gen-mrew(< Sb', Cx':S' > MAT, P:S, (P1:S1 using E1, TSL)) =
    < < E1·Sb' @ P1:S1·Sb' > ; mrew(Cx:S, TSL·Sb', Cx':S') >
    gen-mrew(MAT, P:S, (P1:S1 using E1, TSL))
 if < none, Cx:S > := getAmatch(P1:S1·Sb', P:S·Sb', trueC) .
```

The following rules describe the behavior of the `mrew` continuation. For each solution obtained from the previous task, this continuation finds the appropriate fragment to be rewritten using the next strategy in the strategy list. When the strategy list is empty, the continuation simply takes care of rebuilding the whole term by putting together the solution inside the appropriate contexts.

```
crl < sol(R:S'') TS ; mrew(Cx:S, (P1:S1 using E1, TSL), Cx':S') >
 => < < E1 @ P1:S1 > ; mrew(Cx'':S, TSL, Cx':S') >
    < TS ; mrew(Cx:S, (P1:S1 using E1, TSL), Cx':S') >
 if < none, Cx'':S > := getAmatch(P1:S1, replace(Cx:S, R:S''), trueC) .

rl < sol(R:S'') TS ; mrew(Cx:S, nilTSL, Cx':S') >
=> sol(replace(Cx':S', replace(Cx:S, R:S''))) < TS ; mrew(Cx:S, nilTSL, Cx':S') > .

rl < none ; mrew(Cx:S, TSL, Cx':S') > => none .
```

4.4.8 Strategy definitions

For each strategy definition declaration

```
strat sid : S1 ... Sn @ S .
```

in the strategy module SM, we have a new operator in the transformed theory:

```
op sid : S1 ... Sn -> StratCall .
```

Such strategy identifiers are defined in SM by means of (possibly conditional) equations of the form

```
csd sid(P1, ..., Pn) := E if C .
```

In the transformed theory, all such strategy definitions are put together as a set of

definitions which is the value of a constant of appropriate sort:

```
eq DEFS = (sid(P1, ..., Pn), E, C) , ... .
```

Then the rule for applying strategies defined in this way is the following

```
crl < SC:StratCall @ T:S > => < E' @ T:S >
 if E' := find-def(SC:StratCall, DEFS) .
```

where the auxiliary operation `find-def` is used to find in the set `DEFS` of strategy definitions the one corresponding to the strategy call `SC`.

5 Correctness and determinism

Rewriting logic is based on rewriting equivalence classes of terms modulo some equational axioms; however, to simplify the presentation of the following results we simply consider terms t instead of equivalence classes $[t]$. First, we state a couple of basic properties of the set-theoretic semantics summarized in Section 3 that can be thought of as the soundness and completeness of such semantics.

Proposition 5.1 *For terms t and t' and strategy expression E in $\mathcal{S}(M, SM)$, if $t' \in [\![E \mathbin{@} t]\!]$ then $M \vdash t \longrightarrow^* t'$.*

Proposition 5.2 *If $M \vdash t \longrightarrow^* t'$ then there is a strategy expression E in $\mathcal{S}(M, SM)$ such that $t' \in [\![E \mathbin{@} t]\!]$.*

Proof. It is a consequence of the Sequentality Lemma 3.6 in [9]. Any rewrite $t \longrightarrow^* t'$ can be decomposed as a sequence of rewrites $t \xrightarrow{l_1} t_1 \xrightarrow{l_2} t_2 \cdots \xrightarrow{l_n} t'$, where each step consists of the application of a single rule l_i to a single subterm. Then the desired strategy expression is $E = l_1 \,;\, l_2 \,;\, \ldots \,;\, l_n$. □

In our rewriting semantics, the initial terms denoting the application of a strategy to a state term have the form `<` E `@` t `>` and are of sort `Task`. When this term is rewritten using the rules described in the previous section, new tasks are created, so that rewriting takes place at the level of the sort `Tasks` representing sets of tasks. Those tasks (at the top level) of the form `sol`(t) represent the solutions obtained up to the moment. Then, the abstract function *sols* mentioned in Section 2 is defined in our case as follows: for a term w of sort `Tasks`, $sols(w)$ is the set of terms t such that `sol`(t) is a subterm *at the top* of w.

Now we show that the Maude strategy language satisfies the four requirements put forward in Section 2.

Theorem 5.3 (Soundness)
If $\mathcal{S}(M, SM) \vdash$ `<`E`@`t`>` $\longrightarrow^* w$ and $t' \in sols(w)$, then $M \vdash t \longrightarrow^* t'$.

Theorem 5.4 (Monotonicity)
If $\mathcal{S}(M, SM) \vdash$ `<`E`@`t`>` $\longrightarrow^* w$ and $\mathcal{S}(M, SM) \vdash w \longrightarrow^* w'$, then $sols(w) \subseteq sols(w')$.

Proof. In the rewrite theory $\mathcal{S}(M, SM)$ described in Section 4 there are no rules or equations that can affect a subterm *at the top* of w of the form `sol`(t). Thus, further rewrites take place in disjoint subterms of the ones representing $sols(w)$. □

Theorem 5.5 (Persistence)
If $\mathcal{S}(M, SM) \vdash$ `<E@t>` $\longrightarrow^* w$ *and there exist terms* w' *and* t' *such that* $\mathcal{S}(M, SM) \vdash$ `<E@t>` $\longrightarrow^* w'$ *and* $t' \in sols(w')$*, then there exists a term* w'' *such that* $\mathcal{S}(M, SM) \vdash w \longrightarrow^* w''$ *and* $t' \in sols(w'')$*.*

The rewriting semantics defined in the previous section and the set-theoretic semantics summarized in Section 3 are related by the following results:

Proposition 5.6 *For terms* t *and* t' *and strategy expression* E *in* $\mathcal{S}(M, SM)$*, if there is a term* w *of sort* `Tasks` *such that* $\mathcal{S}(M, SM) \vdash$ `<E@t>` $\longrightarrow^* w$ *with* $t' \in sols(w)$*, then* $t' \in [\![E \mathbin{@} t]\!]$*.*

Proposition 5.7 *If* $t' \in [\![E \mathbin{@} t]\!]$*, then there exists a term* w *such that* $\mathcal{S}(M, SM) \vdash$ `<E@t>` $\longrightarrow^* w$ *and* $t' \in sols(w)$*.*

Putting together Propositions 5.2 and 5.7, we get the following

Theorem 5.8 (Completeness)
If $M \vdash t \longrightarrow^* t'$ *then there is a strategy expression* E *in* $\mathcal{S}(M, SM)$ *and a term* w *such that* $\mathcal{S}(M, SM) \vdash$ `<E@t>` $\longrightarrow^* w$ *and* $t' \in sols(w)$*.*

Moreover, Propositions 5.6 and 5.7 together say that both set-theoretic semantics coincide.

Theorem 5.9 $[\![E \mathbin{@} t]\!] = \{[E \mathbin{@} t]\}$*.*

6 Implementation issues

By taking advantage of the reflective properties of rewriting logic, which allow to consider metalevel entities such as theories as usual data, the transformation described in Section 4 could be implemented as an operation from rewrite theories to rewrite theories, specified itself in rewriting logic. More specifically, we could write this kind of transformation as an extension of Full Maude, as other theory transformations described in [4, Chapter 15]. Full Maude is an extension of Maude, written in Maude itself using the features of the predefined `META-LEVEL` module, which provides an efficient implementation of the reflective features of rewriting logic. In particular, by going to the metalevel, the `getMatch` and `getAmatch` could be implemented by means of the descent functions `metaMatch` and `metaXmatch` provided in `META-LEVEL` as generic operations for matching, either at the top or at all possible positions in a term, together with the operations `up` and `down` that relate the object level and the metalevel.

Instead of doing this, we have written a *parameterized* module (see [4, Section 8.3] for information on Maude parameterization features) that extends the metalevel with the rewriting semantics of the strategy language in a generic and quite efficient way.

In addition to the inherently complex processes of checking the satisfaction of rewrite conditions in applying a conditional rule, and matching and rewriting with strategies at subterms in the `matchrew` combinator (which have been handled by means of continuations), the transformed rewrite theory described in Section 4 is

more complex than expected because of the need for handling substitutions, contexts and matching, and the overloading of operators and repetition of rules for several sorts when a process makes sense for different sorts. Both things get considerably simplified if we are willing to use the predefined `META-LEVEL` module. There are two main reasons for this simplification:

- The `META-LEVEL` module includes a sort `Term` representing by means of appropriate operations all terms for all sorts in a fully generic way, together with sorts `Substitution`, `Context`, `TermList`, etc. to handle typical operations on terms.
- As already mentioned, among others, the `META-LEVEL` module also includes very powerful descent functions `metaMatch`, `metaXmatch`, `metaApply`, and `metaXapply` representing respectively the processes of matching and applying a rule, either at the top or at all possible positions in a term.

Using these `META-LEVEL` features, we can simplify entire families of overloaded operator declarations to a single declaration, like, for example, the following ones:

```
op match : Term Condition -> Test .
op amatch : Term Condition -> Test .
op matchrew : Term Condition TermStratList -> Strat .
op amatchrew : Term Condition TermStratList -> Strat .
op <_@_> : Strat Term -> Task .
op sol : Term -> Task .
```

In the same way, a family of rules for all sorts in a given module can be reduced to a single rule such as, for example, the rules corresponding to the idle strategy, and the sequential and conditional combinators for strategies:

```
rl < idle @ T > => sol(T) .
rl < E ; E' @ T > => < < E @ T > ; seq(E') > .
rl < sol(T) TS ; seq(E') > => < E' @ T > < TS ; seq(E') > .
rl < none ; seq(E') > => none .
rl < if(E, E', E'') @ T > => < < E @ T > ; if(E', E'', T) > .
rl < sol(T') TS ; if(E', E'', T) > => < E' @ T' > < TS ; seq(E') > .
rl < none ; if(E', E'', T) > => < E'' @ T > .
```

where `T` is now a variable of sort `Term`.

The matching tests are also reduced to a single rule in each case. Moreover, they are simply based on the `metaMatch` (for matching only at the top) and `metaXmatch` (for matching anywhere) descent functions available in `META-LEVEL`:

```
rl < match(P, C) @ T > => if metaMatch(MOD, P, T, C, 0) == noMatch
                          then none else sol(T) fi .

rl < amatch(P, C) @ T > => if metaXmatch(MOD, P, T, C, 0, unbounded, 0) == noMatch
                           then none else sol(T) fi .
```

where `P` and `T` are both variables of sort `Term`, and `MOD` is a constant (received as a parameter of the semantics) naming the given sytem module M.

The `matchrew` combinator is treated combining both the continuation idea described in Section 4 and the `meta(X)match` descent functions.

Application of a nonconditional rule is now based on the `metaXapply` descent function. In addition, the `apply-everywhere` and `apply-top` auxiliary operations collect in a single term all the possible results for the different ways of matching, either due to finding the same pattern in different positions of a given state term,

or to structural axioms in such a term (such as commutativity, for example). Instead of returning sets of matches, as the operations `getMatch` and `getAmatch` we have previously considered in our transformation, the descent functions `metaMatch`, `metaXmatch`, `metaApply`, and `metaXapply` have a natural number argument which is used to enumerate all the possible solutions. The same technique appears as the fourth argument of the operations `apply-everywhere` and `apply-top`.

```
op apply-everywhere : Label Substitution Term Nat -> Tasks .
op apply-top : Label Substitution Term Nat -> Tasks .

rl < L[Sb] @ T > => apply-everywhere(L, Sb, T, 0) .

ceq apply-everywhere(L, Sb, T, N) = sol(T') apply-everywhere(L, Sb, T, N + 1)
 if { T', Ty, Sb', CX } := metaXapply(MOD, T, L, Sb, 0, unbounded, N) .
eq apply-everywhere(L, Sb, T, N) = none [owise] .

rl < top(L[Sb]) @ T > => apply-top(L, Sb, T, 0) .

ceq apply-top(L, Sb, T, N) = sol(T') apply-top(L, Sb, T, N + 1)
 if { T', Ty, Sb' } := metaApply(MOD, T, L, Sb, N) .
eq apply-top(L, Sb, T, N) = none [owise] .
```

Application of a conditional rule combines both the continuation idea described in Section 4 for checking the rewrite conditions with strategies, and the `meta(X)apply` descent functions.

The complete implementation can be found at the web page `http://maude.sip.ucm.es/strategies`. Finally, we would like to mention the ongoing implementation of our strategy language at the C++ level, at which the Maude system itself is implemented, to make the language a stable new feature of Maude, and to allow a more efficient execution [7].

7 Example

We show here how to solve by means of strategies the "Crossing the river" problem shown in [4, Section 7.8]. In this problem 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 constant `initial` denotes the initial situation, where we assume that all the objects are located on the left riverbank. 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 .
  op __ : Group Group -> Group [assoc comm] .
  op init : -> Group .
```

```
  vars S S' : Side .
  eq change(left) = right .  eq change(right) = left .
  eq initial = s(left) w(left) g(left) c(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
```

In [4] 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 `oneCross` strategy applies one of the other rules once; finally, the `allCE` strategy returns all the possible reachable states where eating has had the higher priority.

```
smod RIVER-CROSSING-STRAT is
  protecting RIVER-CROSSING .
  strat eating : @ Group .
  sd eating := (wolf-eats | goat-eats) ! .
  strat oneCross : @ Group .
  sd oneCross := shepherd-alone | wolf | goat | cabbage .
  strat allCE : @ Group .
  sd allCE := (eating ; oneCross) * .
endsm
```

If the strategy `allCE ; match (s(right) w(right) g(right) c(right))` applied to the initial state returns a solution, it means that there is a way in which the shepherd can transport all his belongings to the other side of the river.

By instantiating the parameterized module implementing the semantics of the strategy language at the metalevel summarized in Section 6 with the modules `RIVER-CROSSING` and `RIVER-CROSSING-STRAT`, we can execute the above strategy. Since strategy `allCE` is not terminating, we use a bound on the number of rewrites.

```
frew [5000] < call('allCE.Strat) ;
              match('__['s['right.Side],'w['right.Side],
                        'g['right.Side],'c['right.Side]], nil) @
            '__['s['left.Side],'w['left.Side],'g['left.Side],'c['left.Side]] > .
result (sort not calculated):
  sol('__['c['right.Side],'g['right.Side],'s['right.Side],'w['right.Side]]) ...
```

The result contains a solution together with some pending tasks not shown.

8 Concluding remarks

We have given general requirements for strategy languages that control the execution of a rewriting-based language. We have also discussed the mathematical and operational semantics of Maude's strategy language, and have shown how the general requirements are met in its case.

Much work remains ahead. At the implementation level, the C++ implementation of Maude's strategy language still needs to be completed, although a subset of it has been available in alpha versions for some time. At the level of requirements,

we would like to emphasize that the ones given in Section 2 are very basic, but in a sense they are still too weak. For example, although we believe that the deterministic nature of a strategy language is succintly captured by the requirement that the evaluation of a strategy applied to a term should be a set of terms, instead of a set of sets of terms, at the operational semantics level, the monotonicity and persistence requirements are certainly necessary conditions for determinism, but they are still insufficient. The question can be put as follows: how should the nondeterminism of a theory $\mathcal{R}$ be *eliminated as much as possible* in the strategy theory $\mathcal{S}(\mathcal{R})$? We believe that the right answer resides in the notion of *fairness*. That is, any *fair* execution (in an appropriate sense still to be made precise) of the form

$$\sigma \mathbin{@} t \longrightarrow w_1 \longrightarrow w_2 \longrightarrow \cdots w_n \longrightarrow \cdots$$

should be such that $[\![\sigma \mathbin{@} t]\!] = \bigcup_{n \in \mathbb{N}} \mathit{sols}(w_n)$. We leave the investigation of this topic for future reseach. It has, however, a direct bearing on yet another future research direction, namely, the *increased performance of strategy evaluations through parallelism*. The point is that in $\mathcal{S}(\mathcal{R})$ a term $\sigma \mathbin{@} t$ incrementally evaluates to a (possibly nested) *set data structure*, so that the natural concurrency of rewriting logic is directly exploitable in $\mathcal{S}(\mathcal{R})$ by applying different rules in different places of this data structure where solutions are generated. This naturally suggests a distributed implementation of strategy languages, so that a fair implementation where all subexpressions are eventually evaluated should guarantee that all solutions are eventually reached if the strategy language is deterministic, in the stronger sense of also satisfying the fairness requirement sketched above.

References

[1] P. Borovanský, C. Kirchner, H. Kirchner, and P.-E. Moreau. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285:155–185, 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] H. Cirstea, C. Kirchner, L. Liquori, and B. Wack. Rewrite strategies in the rewriting calculus. In B. Gramlich and S. Lucas (eds.), *WRS 2003, 3rd International Workshop on Reduction Strategies in Rewriting and Programming*, ENTCS 86(4), pages 593-624. Elsevier, 2003.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, LNCS 4350. Springer, 2007.

[5] M. Clavel and J. Meseguer. Reflection and strategies in rewriting logic. In J. Meseguer (ed.), *Proc. First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, ENTCS 4, pages 126–148. Elsevier, 1996.

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

[7] S. Eker, N. Martí-Oliet, J. Meseguer, and A. Verdejo. Deduction, strategies, and rewriting. In M. Archer, T. B. de la Tour, and C. A. Muñoz (eds.), *Proc. of the 6th International Workshop on Strategies in Automated Deduction (STRATEGIES 2006)*, ENTCS 174(11), pages 3–25. Elsevier, 2007.

[8] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet (ed.), *Proc. Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, ENTCS 117, pages 417–441. Elsevier, 2005.

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

[10] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer (ed.), *Domain-Specific Program Generation*, LNCS 3016, pages 216–238. Springer, 2004.