A Rewriting Semantics for Maude Strategies

N. Martí-Oliet, J. Meseguer, and A. Verdejo

Universidad Complutense de Madrid

University of Illinois at Urbana-Champaign

IFIP WG 1.3

Urbana, August 1, 2008

In previous works

- we proposed a strategy language for Maude;
- we described a simple set-theoretic semantics for such a language;
- we built a prototype implementation on top of Full Maude;
- we and many other people have used the language and the prototype in several examples:
  - backtracking,
  - semantics,
  - deduction (congruence closure, completion),
  - sudokus,
  - neural networks,
  - membrane computing,
  - ...

`http://maude.sip.ucm.es/strategies`

Goals

- Advance the semantic foundations of Maude's strategy language.
    - 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 [Clavel & Meseguer 96, 97].
    - We describe a detailed operational semantics by rewriting.
    - Rewriting the term $\sigma \mathbin{@} t$ computes the solutions of applying the strategy σ to the term t.
    - Given a *system module* M and a *strategy module* SM, 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.

Goals

- Articulate some general requirements for strategy languages:
    - Absolute requirements: *soundness* and *completeness* with respect to the rewrites in $\mathcal{R}$.
    - More optional requirements, *monotonicity* and *persistence*, represent the fact that no solution is ever lost.
    - The Maude strategy language satisfies all these four requirements.

Strategy language requirements

- The two most basic absolute requirements for any strategy language are:
  - **Soundness**.
    If $\mathcal{S}(\mathcal{R}) \vdash \sigma \,@\, t \longrightarrow^* w$ and $t' \in \mathit{sols}(w)$, 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 $\mathcal{S}(\mathcal{R}) \vdash \sigma \,@\, t \longrightarrow^* w$ and $t' \in \mathit{sols}(w)$.

Strategy language requirements

- An optional requirement is what we call determinism. Intuitively, a strategy language controls and tames the nondeterminism of a theory $\mathcal{R}$.
- At the operational semantics level, this requirement can be made precise as follows:
  - **Monotonicity**.
    If $\mathcal{S}(\mathcal{R}) \vdash \sigma\,@\,t \longrightarrow^* w$ and $\mathcal{S}(\mathcal{R}) \vdash w \longrightarrow^* w'$, then $\mathit{sols}(w) \subseteq \mathit{sols}(w')$.
  - **Persistence**.
    If $\mathcal{S}(\mathcal{R}) \vdash \sigma\,@\,t \longrightarrow^* w$ and there exist terms w' and t' such that $\mathcal{S}(\mathcal{R}) \vdash \sigma\,@\,t \longrightarrow^* w'$ and $t' \in \mathit{sols}(w')$, then there exists a term w'' such that $\mathcal{S}(\mathcal{R}) \vdash w \longrightarrow^* w''$ and $t' \in \mathit{sols}(w'')$.

Strategy language requirements

- A second, optional requirement is what we call the separation between the rewriting language itself and its associated strategy language.
- In the design of Maude's strategy language 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.
- Strategy modules are on a level on top of system modules, which provide the basic rewrite rules.

Maude strategies

Idle and fail are the simplest strategies.

Basic strategies are based on a step of rewriting.

- Application of a rule (identified by the corresponding rule label) to a given term (possibly with variable instantiation).
- For conditional rules, rewrite conditions can be controlled by means of strategy expressions: `L[S]{E1 ... En}`

The rule is applied anywhere in the term where it matches satisfying its condition.

Maude strategies

Top For restricting the application of a rule just to the top of the term.

Tests are strategies that test some property of a given state term, based on matching.

- `amatch T s.t. C` is a test that, when applied to a given state term `T'`, succeeds if there is a subterm of `T'` that matches the pattern `T` and then the condition `C` is satisfied, and fails otherwise.
- `match` works in the same way, but only at the top.

Maude strategies

Regular expressions

```
*** concatenation
op _;_ : Strat Strat -> Strat [assoc] .

*** union
op _|_ : Strat Strat -> Strat [assoc comm] .

*** iteration (0 or more)
op _* : Strat -> Strat .

*** iteration (1 or more)
op _+ : Strat -> Strat .
```

Maude strategies

Conditional strategy is a typical if-then-else, but generalized so that the first argument is also a strategy: `E ? E' : E''`

Using the if-then-else combinator, we can define many other useful strategy combinators as derived operations.

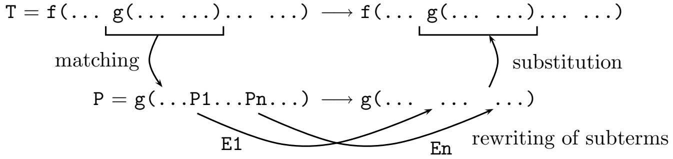
```
E orelse E' = E ? idle : E'
not(E) = E ? fail : idle
E ! = E * ; not(E)
try(E) = E ? idle : idle
```

Maude strategies

Strategies applied to subterms with the (a)`matchrew` combinator control the way different subterms of a given state are rewritten.

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

applied to a state term `T`:


T = f(... g(... ...)... ...) ⟶ f(... g(... ...)... ...)
matching
substitution
P = g(...P1...Pn...) ⟶ g(... ... ...)
E1
En
rewriting of subterms

Maude strategies

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.

Modules The user can write one or more strategy modules to define strategies for a system module `M`.

```
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
```

Rewriting semantics of strategies

- 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.
- Rewriting a term of the form <E@t> produces the results of rewriting the term t by means of the strategy E.

```
sort Task .

op <_@_> : Strat S -> Task .

op sol :  S -> Task .
```

Rewriting semantics of strategies

- A set of tasks constitutes the main infrastructure for defining the application of a strategy to a term:

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

- We use continuations to control the computation of applied strategies:

```
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 .
```

Rewriting semantics of strategies

- We also need several auxiliary operations for representing
  - variables,
  - labels,
  - substitutions,
  - contexts,
  - replacement, and
  - matching.
- In particular, we use 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.

```
op getMatch : S S Condition -> MatchSet .
op getAmatch : S S Condition -> MatchSet .
```

Some strategy rewrite rules

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 .
```

Basic strategies For each nonconditional rule `[l] : t1 => t2` and each sort S in M, we add the rule

```
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′) .
```

The treatment of conditional rules is much more complex.

Some strategy rewrite rules

Regular expressions

```
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 *  .
```

Some strategy rewrite rules

Conditional strategies

```
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 > .
```

Some strategy rewrite rules

Rewriting of subterms

```
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) .
```

Some strategy rewrite rules

Rewriting of subterms

```
crl < sol(R:$S''$) TS ; mrew(Cx:$S$, (P1:$S_1$ using E1, TSL), Cx':$S'$) >
 => < < E1 @ P1:$S_1$ > ; mrew(Cx'':$S$, TSL, Cx':$S'$) >
    < TS ; mrew(Cx:$S$, (P1:$S_1$ using E1, TSL), Cx':$S'$) >
 if < none, Cx'':$S$ > :=
       getAmatch(P1:$S_1$, 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 .
```

Correctness and determinism

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.

Theorem (Soundness)

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

Theorem (Monotonicity)

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

Correctness and determinism

Theorem (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* $t' \in sols(w'')$*.*

Theorem (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* $t' \in sols(w)$*.*

Relation with set-theoretic semantics

Proposition

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

Proposition

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

Relation with set-theoretic semantics

Proposition

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* $\mathcal{S}(M, SM) \vdash \texttt{<}E\texttt{@}t\texttt{>} \longrightarrow^* w$ *with* $t' \in sols(w)$*, then* $t' \in [\![E\,@\,t]\!]$*.*

Proposition

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

Implementation issues

- By taking advantage of the reflective properties of rewriting logic, the transformation described before could be implemented as an operation from rewrite theories to rewrite theories, specified itself in rewriting logic.
- The transformed rewrite theory 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.
- Instead of doing this, we have written a parameterized module that extends the predefined `META-LEVEL` module with the rewriting semantics of the strategy language in a generic and quite efficient way.

Example

```
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] .

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

Example

```
  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
```

Example

```
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
```

Example

```
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]]) ...
```

Conclusions and future work

- We have given general requirements for strategy languages that control the execution of a rewriting-based language.
- We have also discussed a rewriting-based operational semantics of Maude's strategy language.
- We have shown that the general requirements are indeed met by our proposal for Maude's strategy language.

Conclusions and future work

- The C++ implementation of Maude's strategy language still needs to be completed.
- The given requirements are very basic, but in a sense they are still too weak. 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.
- Distributed implementation of strategy languages: the natural concurrency of rewriting logic is directly exploitable in $\mathcal{S}(\mathcal{R})$ by applying different rules to different tasks.

`http://maude.sip.ucm.es/strategies`