An Overview of the Maude Strategy Language and its Applications

Rubén Rubio(✉)

Universidad Complutense de Madrid, Madrid, Spain
rubenrub@ucm.es

Abstract. In the Maude specification language, the behavior of systems is modeled by nondeterministic rewrite rules, whose free application may not always be desirable. Hence, a strategy language has been introduced to control the application of rules at a high level, without the intricacies of metaprogramming. In this paper, we give an overview of the Maude strategy language, its applications, related verification tools, and extensions, illustrated with examples.

1 Introduction

Computation in rewriting logic [29,30] is the succession of independent rule applications in any positions within the terms. This flexibility is the cornerstone of its natural representation of nondeterminism and concurrency, but it is sometimes useful to restrict or guide the evolution of rewriting. For example, a theorem prover does not blindly apply its inference rules, and the local reactions of a chemical system may be modulated by the environment. Strategies are the traditional resource to express these concerns, but specifying them in Maude involved the not so easy task of using its reflective capabilities. This has changed in Maude 3 with the inclusion of an object-level strategy language to explicitly control the application of rules [15]. Several operators resembling the usual programming language constructs and regular expressions allow combining the basic instruction of rule application to program arbitrarily complex strategies, which can be compositionally defined in strategy modules. The language was originally designed in the mid-2000s by Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, and Steven Eker [27] based on previous experience with *internal strategies* at the metalevel [12,14] and earlier strategy languages like ELAN [8], Stratego [10], and Tom [6]. Other similar strategy languages appeared later like ρLog [26] and Porgy [18]. The first prototype was available as a Full Maude extension and it was already given several applications [16,21,40–42]. Now, since Maude 3.0, the language is efficiently implemented in C++ as part of the official interpreter [15].

As well as an executable specification language, Maude is also a verification tool and systems modeled with strategies need also be verified. Together with Narciso Martí-Oliet, Isabel Pita, and Alberto Verdejo, we have extended the Maude LTL model checker to work with strategy-controlled specifications [38]

K. Bae (Ed.): WRLA 2022, LNCS 13252, pp. 65–84, 2022.
https://doi.org/10.1007/978-3-031-12441-9_4

and established connections with external model checkers for evaluating CTL, CTL*, and μ-calculus properties [36]. More recently, we have also developed a probabilistic extension of the Maude strategy language whose specifications can be analyzed using probabilistic and statistical model-checking techniques [32].

This paper is based on an invited tutorial on the Maude strategy language given at WRLA 2022 and explains the language and the aforementioned related topics. Section 2 starts with an introduction to the strategy language, Sect. 3 illustrates it with some more examples and includes references for others, Sect. 4 reviews some related tools and extensions, and Sect. 5 concludes with some remarks for future developments. More information about the strategy language, examples, and its related tools is available at maude.ucm.es/strategies.

2 A Brief Introduction to the Maude Strategy Language

In this section, we give an introduction to the Maude strategy language through an example, without claiming to be exhaustive or systematic. For a comprehensive informal reference about the language, we suggest its dedicated chapter in the Maude manual [13, §10]. Formal semantics, both denotational [32] and operational [28,38], are also available [35].

Let us introduce the following system module `WORDS` as running example, where `Word`s are defined as lists of `Letter`s in the latin alphabet. Three rules, `swap`, `remove`, and `append`, are provided to manipulate words.

```
mod WORDS is
  sorts Letter Word .
  subsort Letter < Word .

  ops a b c d ··· z : -> Letter [ctor] .
  op nil : -> Word [ctor] . *** empty word
  op __  : Word Word -> Word [ctor assoc id: nil] .

  rl [swap]   : L W R => R W L .
  rl [remove] : L => nil .
  rl [append] : W => W L [nonexec] .
endm
```

The `swap` rule permutates two letters in a word, `removes` removes one, and `append` attaches a new letter `L` to the end of the word. This latter rule is marked `nonexec`(utable), since it includes an unbound variable in the right-hand side. However, we will be able to execute it with the strategy language.

This rewrite system is nonterminating due to the idempotent `swap` rule. In fact, for every word with at least two letters, Maude's `rewrite` command will loop.

```
Maude> rewrite i t .                    *** does not terminate
```

However, we can obtain something useful from this module by controlling rewriting with the strategy language. The command for executing a strategy expression

α on a term t is `srewrite` t `using` α and its output enumerates all terms that are obtained by this controlled rewriting. Multiple solutions are possible, since strategies are not required to completely remove nondeterminism. The elementary building block of the strategy language is the application of a rule, as cannot be otherwise, whose most basic form is the strategy `all` that applies any rule in the module.

```
Maude> srewrite i t using all .

Solution 1                          Solution 3
rewrites: 1                         rewrites: 3
result Word: t i                    result Letter: i

Solution 2                          No more solutions.
rewrites: 2                         rewrites: 3
result Letter: t
```

The previous fragment evaluates `all` on the term `i t` yielding three different solutions, one for `swap` and two for `remove`. This is equivalent to the command `search` t `=>1 W:Word` that looks for all terms reachable by a single rewrite from t, but the strategy language allows for more flexibility. For instance, if we want to apply only rules with a given label, say `swap`, we can simply write `swap`.

```
Maude> srewrite i t using swap .

Solution 1
rewrites: 1
result Word: t i

No more solutions.
rewrites: 1
```

Rules are applied in any position of the term by default, as seen in the second and third solutions of the first `srewrite` command, or in application of `swap` to the word `w o n` with result $\{\underline{o\,w}\,n, \underline{n}\,o\,\underline{w}, w\,\underline{n\,o}\}$. If this is not desired, the `top` modifier can be used to limit their application to the whole term, like in `top(swap)` or `top(all)`, whose only result is `n o w`. For being more precise when applying rules anywhere, we can also specify an initial substitution to be applied to both sides of the rule and its condition before matching. For example, `swap[L <- w]` would instantiate the rule `L W R => R W L` to `w W R => R W w` and yield `n o w` and `o w n` as solutions. Similarly, `swap[L <- w, W <- nil]` would turn the rule into `w R => R w` and produce the single solution `o w n`.

Substitutions are essential when dealing with nonexecutable rules, like `append` in the `WORDS` module, whose unbound variables can then be instantiated. We can execute `top(append[L <- a]) ; top(append[L <- t])` on the word `g o` to turn it into a `g o a t`. In addition to using `top` for ensuring that the letter is appended at the end of the word, the previous strategy introduced a new combinator `;` that executes a strategy on the results of the previous one, like functional composition or concatenation. Its identity element is the strategy constant `idle` that returns the original term unchanged as only solution. Another

pervasive combinator is the disjunction or nondeterministic choice of strategies α_1 `|` $\cdots$ `|` α_n, whose results are the union of the results of its operands. For example, `remove[L <- w] | remove[L - n]` evaluates on `won` to $\{$`on`, `wo`$\}$. The identity element of the disjunction is `fail`, which does not produce any solution at all. In a broader sense, we say that a strategy *fails* when it does not produce any solution.

Suppose we want to calculate all permutations of a given word. This can be achieved by accumulating the words obtained by successive swaps,

$$\texttt{swap | (swap ; swap) | (swap ; swap ; swap) |} \cdots,$$

until no new words are obtained. The iteration combinator α`*`, which can be inductively described as `idle |` α `;` α`*`, expresses this common pattern. Observe the correspondence between the last strategy combinators and the constructors of regular expressions:

Regular expressions	ε	$\emptyset$	$\alpha \mid \beta$	$\alpha\beta$	α^*
Strategy language	`idle`	`fail`	α `|` β	α `;` β	α`*`

As formalized in [38], the full strategy language is able to describe any recursively enumerable subset of the executions of the original rewrite system, over both finite and infinite words, but regular languages are specially easily expressed with these constructs. Coming back to the example, the expression `swap *` gives all permutations of the original word, so 24 solutions for `goat` after a total of 81 rewrites. If we only need the solutions that start with `g` and finish with a letter other than `a`, we can execute the strategy `swap * ; match g W R s.t. R =/= a` where `match` P `s.t.` C is an operator that filters the terms that match a pattern P and satisfy a condition C. Indeed, it works like an `idle` when the conditions hold and like a `fail` when they do not. Other test variants, `xmatch` and `amatch`, exist for matching with extension for structural axioms (i.e. matching fragments of the flattened associative and/or commutative operators) or inside subterms, respectively.

In Spanish, the letter `h` is not pronounced except when preceded by `c`, so texters and tweeters sometimes obviate it against the criteria of the Royal Spanish Academy. If we do likewise, we would reduce `hola` to its homophone `ola` with `remove[L <- h]`. However, we do not want to transform `brocha` into `broca`, because they are pronounced differently. We need a new tool to restrict the application to a specific context, and this is the subterm rewriting `matchrew` operators. Their syntax is similar to that of tests

$$\texttt{matchrew}\ P\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1\texttt{,}\ \ldots\texttt{,}\ x_n\ \texttt{using}\ \alpha_n$$

but the subterms matched by the variables $x_1, \ldots, x_n$ in its pattern P are rewritten with strategies $\alpha_1, \ldots, \alpha_n$. The solutions of this operator are the combinations of all solutions obtained for every subterm, which are rewritten independently. For example, `matchrew L W by L using remove[L <- h]` will safely remove the first letter of the word if it is an `h`. For removing `h` in the middle of the word, we write

`xmatchrew L R by s.t. L =/= c by R using remove[L <- h]` to ensure that the previous letter is not a `c`. Notice that we have used `xmatchrew` instead of `matchrew`, because we do not want to match the whole term but a fragment of the associative list of letters. These two strategies can be combined with a nondeterminsitic choice $\alpha \mid \beta$ to remove any silent `h` letters in a word. For example, applying this strategy to `h e c h o` yields `e c h o` but not `h e c o`. Unfortunately, the word `h i p o c l o r h i d r i a` is not rewritten to `i p o c l o r i d r i a`, because only one `h` is removed at a time. In order to normalize a term with respect to a strategy, i.e. to apply a strategy until it cannot be executed further, the language includes the α `!` combinator. Putting the previous strategy under this normalization operator we obtain an expression that removes all silent `h` from a word.

Writing strategies as standalone expressions becomes unmanageable as they grow in size. Strategy modules are available to give them name and define them modularly. For example, the following strategy module `WORDS-STRAT` extends the system module `WORDS` with two new strategies, `rmh` and `rmh-one`, declared with the `strats` statement.

```
smod WORDS-STRAT is
  protecting WORDS .

  strats rmh rhm-one @ Word .

  vars L R : Letter .
  var  W   : Word .

  sd rmh := rmh-one ! .
  sd rmh-one := matchrew L W by L using remove[L <- h] .
  sd rmh-one := xmatchrew L R s.t. L =/= c
                  by R using remove[L <- h] .
endsm
```

The sort after the `@` sign indicates which terms are intended to be rewritten by the strategy, although it does not have any practical effect. Each named strategy is assigned zero or more strategy expressions with definitions that start by the `sd` keyword or by `csd` if they are conditional. In the module above, `rmh` is the strategy that removes every silent `h` in a word, while the two definitions of `rmh-one` remove a single `h` at initial and inner position, respectively. When the strategy `rmh-one` is called in `rmh`, the two definitions for `rmh-one` are executed nondeterministically, as if their expressions where joined by the disjunction `|` operator.

One of the greatest advantages of strategy modules is the possibility of defining recursive strategies. For example, the following strategy module `WORDS-REPEAT` declares a single recursive strategy `remove`(l, n) with two parameters that removes exactly n occurrences of the letter l in the subject term.

```
smod WORDS-REPEAT is
  protecting WORDS .

  strat remove : Letter Nat @ Phrase .
```

```
var N : Letter . var N : Nat .
```

Its semantics is given by two definitions with disjoint matching patterns. For removing zero letters, we simply do nothing with `idle`.

```
sd remove(L, 0) := idle .
```

Otherwise, one occurrence of the letter `L` is deleted with the `remove` rule and the strategy itself is called recursively with a decremented counter.

```
  sd remove(L, s N) := remove[L <- L] ; remove(L, N) .
endsm
```

For example, rewriting `bazaar` with `remove(a, 2)` gives `bzar` and `bazr`. However, `rewrite(a, 4)` would fail because of the attempt to call `remove` for the fourth time. We can make `remove`(l, n) erase as many occurrences of l as possible but no more than n with the following change on the second definition:

```
sd remove(L, s N) := remove[L <- L] ? remove(L, N)
                                    : idle .
```

We have used the conditional operator $\alpha\,?\,\beta : \gamma$ that evaluates β on the results of α or γ on the original term if α yields no solution. This way, we only invoke the recursive strategy if the `remove` rule succeeds, and the execution is finished when it fails. Conditional operators are quite general since its condition is an arbitrary strategy and recurrent conditional patterns are given dedicated syntax. For example, α `or-else` β executes β only if α fails, and it is equivalent to α `? idle :` β.

3 Some Examples

In this section, we further illustrate the language with three examples. At the same time, we cite other published works where applications of the language have been presented.

3.1 Deduction Procedures

In deductive reasoning, inference rules should be carefully applied to reach the desired conclusions in an efficient way. A free or inadequate application of the rules may loop or lead to a poor performance in many examples of inference systems. For instance, the Davis-Putnam-Logemann-Loveland (DPLL) system for deciding the satisfiability of a Boolean formula has a natural brute-force *split* rule that generates two subproblems, where the variable x is respectively assumed true and false.

$$\text{(split)}\quad \frac{\Delta \vdash \Gamma, x \vee C}{\Delta, x \vdash \Gamma \qquad \Delta, \neg x \vdash \Gamma, C} \quad \text{if } x, \neg x \notin \Delta$$

Of course, repeatedly applying this rule will solve the satisfiability problem, but at an exponential cost in the best case. The inference system includes other rules

that are better applied first. For example, *subsume* removes pending clauses with a satisfied atom.

$$\text{(subsume)} \quad \frac{\Delta \vdash \Gamma, x \vee C}{\Delta \vdash \Gamma} \quad \text{if } x \in \Delta$$

Hopefully, this may remove some variables in C that do not appear elsewhere, avoiding some superfluous case distinctions. A first rudimentary strategy for SAT solving with these rules would be `(subsume | ···) or-else split` where the dots are occupied by the other simplification rules. A second one can be more selective and apply `split` to the variable that cancels the most possible clauses. More serious strategies for the DPLL rules are programmed in the Maude strategy language in [20].

Implementations of deduction procedures do not usually individualize the rules in their code, but rule-based systems like Maude can easily separate the basic logic and its control using strategies. In the literature, this has been encouraged by the Kowalski's motto *Algorithm = Logic + Control* [22] or Lescanne's *Rule + Control* approach [24]. This latter work implements in Caml four equational completion procedures on top of the inference rules by Bachmair and Dershowitz [5], decoupling at some extent the rules from their control. These same completion procedures have also been specified using the initial prototype of the Maude strategy language in [42] and an improved redesign of this specification is available in [32]. In this latter version, we have clearly separated the inference rules in a system module `COMPLETION` and the four deduction procedures in four strategy modules being protected extensions of `COMPLETION`, as depicted in Fig. 1. Each procedure is a recursive strategy that maintains the inference state in its call arguments without modifying the term or adding more rules.

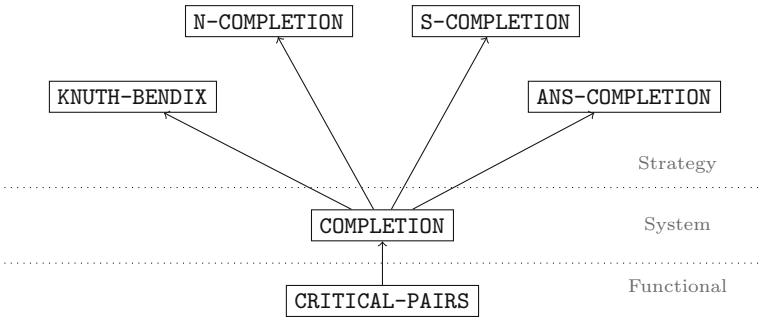

Fig. 1. Equational procedure specification with clear separation of concerns.

Following similar ideas, other examples of deduction procedures are programmed using the Maude strategy language like congruence closure [16], the Martelli-Montanari unification procedure [1], and a Sudoku solver [40].

3.2 Semantics of Programming Languages

Strategies are also meaningful when dealing with semantics of programming languages. Structural operational semantics define the small-step behavior of programs through inference rules whose premises are steps for the constituents of the program. In this sense, they are not much different to the deduction procedures seen in the previous section, and the Maude strategy language can be useful to describe them [9]. Strategies are particularly useful to generalize semantic rules with negative premises or rule precedence, which are not easily captured otherwise. For example, negation in Prolog is described by the following rule

$$\text{(split)} \quad \frac{\Gamma \vdash g \not\rightarrow^* \texttt{nil}}{\Gamma \vdash (\backslash\texttt{+}\, g), gs \rightarrow gs}$$

that removes the negated goal $\texttt{n+}\, g$ from the goal list if g cannot be solved. This premise can be expressed with the strategy combinator $\texttt{not}(\alpha) \equiv \alpha$ `? fail : idle`. Indeed, we have specified an executable Prolog interpreter in [13] where negation and cuts are described with strategies.

Let us illustrate the relation between strategies and programming with two simple strategies for the untyped λ-calculus. We specify the basics of this formalism in the following module `LAMBDA`.

```
mod LAMBDA is
  sorts Var LambdaTerm .
  subsort Var < LambdaTerm .

  op \_._ : Var LambdaTerm -> LambdaTerm [ctor...] .
  op __ : LambdaTerm LambdaTerm -> LambdaTerm [ctor...] .

  op subst : LambdaTerm Var LambdaTerm -> LambdaTerm .
  *** the equational definition of subst is ommitted

  var x : Var . vars M N : LambdaTerm .

  rl [beta] : (\ x . M) N => subst(M, x, N) .
endm
```

As usual, there are only two constructors of λ-terms, abstraction $\lambda x.M$ and application $M\,N$, and we consider a single reduction rule `beta` that transforms $(\lambda x.M)N$ into $M[x/N]$ where every occurrence of x is replaced by N in M. There may be multiple positions where to apply the β rule in a λ-term, called *β-redexes*, but the Church-Rosser theorem tells that the calculus is confluent, i.e. if we can reduce $t \rightarrow^* t_1$ and $t \rightarrow^* t_2$, there exists a t' such that $t_1 \rightarrow^* t'$ and $t_2 \rightarrow^* t'$. Nevertheless, how rules are applied still matters, since some reductions may lead to a normal form while others may diverge for the same term (see Fig. 2). Another classical result of the λ-calculus tells that repeatedly reducing the outer leftmost redex always leads to a normal form in case it exists, and this can be expressed as a strategy.

Fig. 2. Two reduction paths from the λ-term (**KI**) Ω.

In the strategy module `LAMBDA-STRAT`, we extend `LAMBDA` with two strategies `normal` and `applicative` for reducing λ-terms, but more variants can be defined like call-by-value and call-by-name. These are covered in an extended specification of this example [32,34].

```
smod LAMBDA-STRATS is
  protecting LAMBDA .
  strats normal applicative ··· @ LambdaTerm .
  vars x y z t : Var .          vars M N : LambdaTerm .
```

The definition of `normal` describes a single reduction step of the normalization strategy mentioned in the previous paragraph, i.e. applying `beta` on the outer leftmost redex.

```
sd normal := matchrew \ x . M by M using normal
  | top(beta) or-else matchrew M N by M using normal
              or-else matchrew M N by N using normal .
```

For completely reducing a term, we can simply write `normal !` with the normalization operator. Alternatively, λ-terms can be reduced in the usual applicative order, by selecting the inner rightmost redex.

```
sd applicative := matchrew \ x . M by M using applicative
  | matchrew M N by N using applicative
    or-else matchrew M N by M using applicative
    or-else top(beta) .
```

However, this new strategy does not ensure that a normal form is reached if it exists. We can see it by running the `K I Omega` term of Fig. 2 under both strategies.

```
Maude> srew K I Omega using normal !  .

Solution 1
rewrites: 17
result LambdaTerm: \ x . x

No more solutions.
rewrites: 17
```

The normal form $\mathbf{I} \equiv \lambda x.x$ is reached with `normal`, but it is not with `applicative`. Notice that the `srewrite` command finishes even though the strategy does not terminate.

```
Maude> srew K I Omega using applicative !  .
```

```
No solution.
rewrites: 11
```

This is because the `srewrite` infrastructure is able to detect execution cycles and interrupt the evaluation of the strategy, but the absence of solutions is the evidence that applicative reduction does not terminate for this term.

The semantics of other programming languages have been addressed with the Maude strategy language like Eden [21], the REC language of Glynn Winskel's textbook [34], the ambient calculus [31], CCS [27], and the Maude strategy language itself [38].

3.3 Games

Strategies are pervasive in games, most usually for specifying how players can solve or win them. Besides the Sudoku solver [40], already mentioned, the strategy language has been used to work out the 15-puzzle [32], the Hanoi tower's puzzle [15], to compare different player strategies for Tic-Tac-Toe by model checking [37], and to solve other smaller games [1].

In addition to expressing procedures for solving a game, strategies can also specify intrinsic restrictions that are rather difficult to express with rules. For example, in the river-crossing problem formalization in [36], we use strategies to enforce a precedence that is part of the rules of the game. Here, we briefly describe this example without going into details about the data representation, which are available in the referenced article and in the repository of examples [1]. In the classical river-crossing puzzle, a *shepherd* needs to cross a river with a `wolf`, a `goat`, and a `cabbage` using a boat with room for two passengers, the shepherd included. The problem is that the wolf would eat the goat and the goat would eat the cabbage as soon as they are left alone without the shepherd in any side of the river. Our representation of the river is `left` L | `right` R where L and R are sets of characters and `left shepherd wolf goat cabbage` | `right` is the initial state. Four rewrite rules, `alone`, `wolf`, `goat`, and `cabbage`, let the shepherd cross alone or with the corresponding passenger to the other side. Two more rules, `wolf-eats` and `goat-eats`, carry out the threat of the mentioned animal over its "prey". Moreover, a key restriction is that the wolf and the goat will never miss the opportunity to eat, so eating must happen eagerly before moving. For instance, the `wolf` rule rewrites the initial state to `left goat cabbage | right shepherd wolf`, where the goat and the cabbage are left alone. In this situation, the `goat-eats` rule must be applied to yield `left goat| right shepherd wolf`, but moving `alone` is also allowed in the uncontrolled rewrite system. Indeed, we can try to use the `search` command to solve the problem, by looking for the final state.

```
Maude> search initial
          =>* left | right shepherd wolf goat cabbage .

Solution 1 (state 31)
states: 32  rewrites: 60
```

```
empty substitution

No more solutions.
states: 36  rewrites: 89
```

This answer would make us think that the problem is solvable. It is indeed, but this command is not an evidence, since recovering the path to this solution gives an invalid sequence of moves.

```
Maude> show path 31 .
state 0, River: right | shepherd wolf goat cabbage left
===[ wolf ]===>
state 2, River: goat cabbage left | shepherd wolf right
===[ alone ]===>
state 7, River: shepherd goat cabbage left | wolf right
===[ goat ]===>
state 15, River: cabbage left | shepherd wolf goat right
===[ alone ]===>
state 23, River: shepherd cabbage left | wolf goat right
===[ cabbage ]===>
state 31, River: left | shepherd wolf goat cabbage right
```

In the second state the `goat-eats` rule should be applied, but `alone` is applied instead.

In order to enforce the precedence of eating over moving we can use the Maude strategy language. The following recursive strategy `eagerEating` applies rules under this restriction until the final state is reached.

```
sd eagerEating :=
  (match left | right shepherd wolf goat cabbage) ? idle
  : ((eating or-else oneCrossing) ; eagerEating) .
sd eating      := wolf-eats | goat-eats .
sd oneCrossing := shepherd | wolf | goat | cabbage .
```

Notice that nonterminating executions are also admitted by the strategy, but they are not a problem for the strategy execution engine because of its cycle detector. We can use the experimental `search` command controlled by a strategy[1] to find a valid solution for the problem.

```
Maude> search initial =>* left | right shepherd wolf
          goat cabbage using eagerEating .
Solution 1 (state 30)
states: 36  rewrites: 72
empty substitution

No more solutions.
states: 36  rewrites: 75
Maude> show path 30 .
state 0, River: shepherd wolf goat cabbage left | right
```

[1] The `search-using` command is not currently available in the official version of Maude, but in an extended version with the strategy-aware model checker [38].

```
===[ goat ]===>
state 23, River: wolf cabbage left | shepherd goat right
===[ alone ]===>
state 24, River: shepherd wolf cabbage left | goat right
===[ wolf ]===>
state 25, River: cabbage left | shepherd wolf goat right
===[ goat ]===>
state 27, River: shepherd goat cabbage left | wolf right
===[ cabbage ]===>
state 28, River: goat left | shepherd wolf cabbage right
===[ alone ]===>
state 29, River: shepherd goat left | wolf cabbage right
===[ goat ]===>
state 30, River: left | shepherd wolf goat cabbage right
```

3.4 Other Examples

Beyond the examples already cited in the previous sections, other applications of the Maude strategy language have been published like specifications of the Routing Information Protocol [38], membrane systems with several extensions and model checking [39], the simplex algorithm and a parameterized backtracking scheme with instances for finding solutions to the labyrinth, 8-queens, graph m-coloring, and Hamiltonian cycle problems [34], semaphores and processor scheduling policies [38], a branch and bound scheme [1], Bitcoin smart contracts [4], neural networks [41], and more [27].

4 Related Tools and Extensions

In this section, we briefly describe three extensions and related tools for the strategy language: an extended model checker for strategy-controlled systems, the support for reflective manipulation of strategies with some applications, and a probabilistic extension of the language.

4.1 Model Checking

Model checking [11] is an automated verification technique based on the exhaustive exploration of the execution space of the model. The properties to be checked are usually expressed in temporal logics like Linear-Time Temporal Logic (LTL) or Computation Tree Logic (CTL). Its integrated model checker for LTL is one of the most widely used features of Maude [17]. However, it cannot be applied to strategy-controlled specifications, since it does not know anything about strategies. In order to solve this, we implemented a strategy-aware extension [33,38] of this LTL model checker, which has been extended for branching-time temporal logics in subsequent works [36].

Intuitively, a strategy describes a subset of the executions of the original model or a subtree of the original computation tree, so the satisfaction of a linear-time or branching-time temporal property in a strategy-controlled system should be evaluated on these representations of its restricted behavior. For example, in the river-crossing puzzle of Sect. 3.3, the LTL formula $\Box\,(risky \rightarrow \Box \neg\, goal)$ (once a risky state –where an animal is able to eat– is visited, the goal is no longer reachable) does not hold in the uncontrolled system

```
Maude> red modelCheck(initial, [] (risky -> [] ~ goal)) .
rewrites: 43
result ModelCheckResult: counterexample(..., ...)
```

but it does hold when the system is controlled by the `eagerEating` strategy.

```
Maude> red modelCheck(initial, [] (risky -> [] ~ goal),
                              'eagerEating) .
rewrites: 178
result Bool: true
```

However, the property $\Diamond\, goal$ (the goal is eventually reached) does not hold in any case, since the shepherd may keep moving in cycles, for example.

```
Maude> red modelCheck(initial, <> goal, 'eagerEating) .
rewrites: 24
result ModelCheckResult: counterexample(..., ...)
```

Counterexamples returned by the strategy-aware model checker are executions allowed by the strategy, which are often shorter or easier to understand. The usage of the strategy-aware model checker is documented in [38] and it can be downloaded from maude.ucm.es/strategies, along with examples and documentation. Branching-time properties in CTL, CTL*, and the μ-calculus can also be checked with the `umaudemc` tool [36], also available at this website. For example, we can check the CTL* property $\mathbf{A}(\Box \neg\, risky \rightarrow \mathbf{E} \Diamond\, goal)$, saying that we can eventually reach the goal by avoiding risky states, which holds both with and without strategy.

```
$ umaudemc check river.maude initial
    'A ([] ~ risky -> E <> goal)'
The property is satisfied in the initial state
(36 system states, 197 rewrites, holds in 18/36 states)
$ umaudemc check river.maude initial
    'A ([] ~ risky -> E <> goal)' eagerEating
The property is satisfied in the initial state
(35 system states, 176 rewrites, holds in 17/35 states)
```

4.2 Reflective Manipulation of Strategies

Even though the strategy language was introduced to avoid the complications of the metalevel when controlling rewriting, reflection is still useful in the context of strategies. Like any other Maude feature, the strategy language, strategy

modules, and the associated operations are reflected at the metalevel [13, §17.3]. First, every combinator of the language is declared as a term of sort `Strategy` or its subsorts in the `META-STRATEGY` module of the Maude prelude.

```
ops fail idle : -> Strategy [ctor] .
op _[_]{_} : Qid Substitution StrategyList
             -> RuleApplication .
op match_s.t._ : Term EqCondition -> Strategy .
op _?_:_ : Strategy Strategy Strategy -> Strategy [...] .
op _[[_]] : Qid TermList -> CallStrategy [ctor prec 21] .
*** and more
```

Then, strategy modules and their statements are defined as data in `META-MODULE`.

```
op sd_:=_[_]. : CallStrategy Strategy AttrSet
                -> StratDefinition .
op csd_:=_if_[_]. : CallStrategy Strategy EqCondition
                    AttrSet -> StratDefinition .
op smod_is_sorts_._______endsm : ··· -> StratModule .
```

Finally, the `srewrite` and `dsrewrite`[2] commands are metarepresented in the `META-LEVEL` module.

```
sort SrewriteOption .
ops breadthFirst depthFirst : -> SrewriteOption [ctor] .
op metaSrewrite : Module Term Strategy SrewriteOption
                  Nat ~> ResultPair? [...] .
```

Strategies can be reflectively generated and transformed using these tools with interesting applications. In [37], we explain metaprogramming of strategies with several examples, from a theory-dependent normalization strategy for context-sensitive rewriting [25] to extensions of the strategy language itself. For instance, the similar strategy languages ELAN [8] and Stratego [10] include some constructs that are not available in Maude, like congruence operators $f(\alpha_1, \ldots, \alpha_n)$ for applying strategies to every argument of a symbol f. However, these absences are not substantial, since most can be easily expressed using the combinators of the Maude strategy language, for which an automated translation can be programmed at the metalevel.

Multistrategies is another more complex extension that allows distributing the control of the system in multiple strategies $\alpha_1, \ldots, \alpha_n$ orchestrated by another one γ. Typically, each strategy α_k describes the behavior of a component, agent, or player of the system, while γ specifies how their executions are interleaved. Namely, γ can make them execute concurrently at almost rule-application granularity, by turns, or in other arbitrary ways. Systems controlled by multistrategies can be executed and model checked with an implementation that relies on the metarepresentation of the strategy language.

Yet another example is an extensible small-step operational semantics of the Maude strategy language, already mentioned in Sect. 3.2. It is specified with

[2] `dsrewrite` is the depth-first search variant of `srewrite`, which does a fair breadth-first-like search.

rules and strategies that manipulate terms and strategies at the metalevel [38]. Of course, running strategies or model checking under this semantics is not useful in practice, since the builtin implementation of the language is much more efficient. However, experimentation is easier with this specification. For instance, a synchronized rewrite or intersection operator $\alpha \wedge \beta$ denotes the rewriting paths allowed by both α and β, which cannot be expressed in terms of the original combinators. Nevertheless, $\alpha \wedge \beta$ can be implemented with a pair of two execution states of the semantics that are advanced in parallel as long as they represent the same term.

4.3 A Probabilistic Extension

In addition to qualitative properties, quantitative aspects like time, cost, and probabilities are relevant when analyzing the behavior of systems. Statistical methods are often used to estimate them by simulation, that is, by evaluating the measures on many executions generated at random. However, for this analysis to be sound, all sources of nondeterminism must be quantified. We argued before that strategies are a useful resource to restrict nondeterminism, but they are also suitable for quantifying it. Indeed, probabilistic choice operators have been proposed for ELAN [7] and are available in Porgy [18]. In the context of Maude, PSMaude [7] proposes a restricted strategy language for quantifying the choice of positions, rules, and substitutions. These latter specifications can be simulated and model checked against PCTL properties.

For the specification of probabilities in the Maude strategy language, new combinators have been added. The first one is equivalent to those of ELAN and Porgy.

- A quantified version of non-deterministic choice $\alpha_1 \mid \cdots \mid \alpha_n$ where each alternative is associated a weight

  $$\texttt{choice}(w_1 : \alpha_1, \ldots, w_n : \alpha_n)$$

  Weights w_k are terms of sort `Nat` or `Float` that are evaluated in the context where the strategy is executed. The probability of choosing the alternative α_k is $\sigma(w_k) / \sum_{i=1}^{n} \sigma(w_i)$ where σ is the current variable context.
- A sampling operator from a probabilistic distribution π to a variable `X` that can be used in a nested strategy α

  $$\texttt{sample}\ X\ \texttt{:=}\ \pi(t_1, \ldots, t_n)\ \texttt{in}\ \alpha$$

  The repertory of available distributions includes `bernoulli(`p`)`, `uniform(`a`,` b`)`, `norm(`μ`,` σ`)`, `exp(`λ`)` (for the exponential distribution), and `gamma(`α`,` λ`)`. Their parameters are also evaluated in the current variable context.

These operators are not currently available in the official version of Maude, but in the extended version including the strategy-aware model checker in Sect. 4.1. They can be used in the usual `srewrite` and `dsrewrite` commands, and in

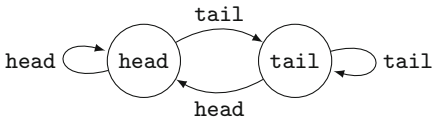

Fig. 3. Coin toss module.

the `metaSrewrite` function. When a `sample` operator is evaluated, a variable is sampled at random and the nested α is executed with this random value. When a `choice` is executed, one of the strategies is chosen at random according to their probabilities.

We can apply both statistical and probabilistic model-checking methods on these specifications enhanced with probabilities. For instance, suppose we model tossing a coin like in Fig. 3, with two constants `head` and `tail`, two homonym rules, and two homonym atomic propositions. A fair coin can then be modeled with the strategy `choice(1 : head, 1 : tail)`. The expected number of steps until the first tail is obtained can be estimated with the `scheck` subcommand of `umaudemc`.

```
$ umaudemc scheck coin head firstTail.quatex
  --assign strategy 'choice(1 : head, 1 : tail) !'
Number of simulations = 46530
μ = 6.00143993122 σ = 5.50060692624 r = 0.0499808311155
```

The simulation is driven by an expression in the QuaTEx language of PMaude [2] specified in the `firstTail.quatex` file, where `#` means *in the next step*.

```
FirstTail() = if (s.rval("C == tail") == 1) then
              s.rval("steps") else # FirstTail() fi ;
eval E[FirstTail()] ;
```

However, statistical model checking is more useful when continuous-time aspects are involved, i.e., when using the `sample` operator. For discrete models like this one, we can also use probabilistic model-checking techniques. This is available through the `pcheck` subcommand of `umaudemc` and relies on either the PRISM [23] or Storm [19] model checkers. The following command is equivalent to the previous one but using probabilistic methods.

```
$ umaudemc pcheck coin head '<> tail' --assign strategy
                  'choice(1 : head, 1 : tail) !' --steps
Result: 6.0
```

As well as obtaining expected values, `pcheck` allows calculating the probabilities that a temporal formula in LTL, CTL, PCTL, and other logics holds.

```
$ umaudemc pcheck coin head '<> <= 5 tail'
  --assign strategy 'choice(1 : head, 1 : tail) !'
Result: 0.96875
```

For complementing this haphazard appetizer, more information on the probabilistic extensions can be found in [32] and the strategy language website.

5 Conclusions

In this tutorial, we have provided an overview of the Maude strategy language, illustrated with several examples, and explained some extensions and associated tools. We refer the interested reader to the works cited in the paper and to the maude.ucm.es/strategies website to complete the information about the strategy language and those tools.

As future work, we plan extending the probabilistic strategy language in Sect. 4.3 with an operator to quantify the choice of matches

$$\texttt{matchrew}\ P\ \texttt{s.t.}\ C\ \texttt{with weight}\ w\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1, \ldots, x_n\ \texttt{using}\ \alpha_n$$

and new verification features. Another natural and interesting extension of the strategy language is its application to narrowing [3].

Acknowledgments. I would like to thank Kyungmin Bae for the invitation to give this tutorial at WRLA, as well as the coauthors of previous works on the strategy language, Steven Eker, Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, Isabel Pita, and the other members of the Maude Team. Special thanks are due to Narciso Martí-Oliet for their helpful comments when revising this manuscript. This work was partially supported by the Spanish Ministry of Science and Innovation through projects TRACES (TIN2015-67522-C3-3-R) and ProCode (PID2019-108528RB-C22), and by the Spanish Ministry of Universities through grant FPU17/02319.

References

1. Examples of the Maude strategy language (2022). https://fadoss.github.io/strat-examples
2. Agha, G.A., Meseguer, J., Sen, K.: PMaude: rewrite-based specification language for probabilistic object systems. In: Cerone, A., Wiklicky, H. (eds.) Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages, QAPL 2005, Edinburgh, UK, 2–3 April 2005. Electronic Notes in Theoretical Computer Science, vol. 153(2), pp. 213–239. Elsevier (2006). https://doi.org/10.1016/j.entcs.2005.10.040
3. Aguirre, L., Martí-Oliet, N., Palomino, M., Pita, I.: Strategies in conditional narrowing modulo SMT plus axioms. Technical report 2/21, Departamento de Sistemas Informáticos y Computación. Universidad Complutense de Madrid (2021). https://eprints.ucm.es/68621/
4. Atzei, N., Bartoletti, M., Lande, S., Yoshida, N., Zunino, R.: Developing secure bitcoin contracts with BitML. In: Dumas, M., Pfahl, D., Apel, S., Russo, A. (eds.) Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, 26–30 August 2019, pp. 1124–1128. ACM (2019). https://doi.org/10.1145/3338906.3341173

5. Bachmair, L., Dershowitz, N.: Equational inference, canonical proofs, and proof orderings. J. ACM **41**(2), 236–276 (1994). https://doi.org/10.1145/174652.174655
6. Balland, E., Brauner, P., Kopetz, R., Moreau, P.-E., Reilles, A.: Tom: piggybacking rewriting on Java. In: Baader, F. (ed.) RTA 2007. LNCS, vol. 4533, pp. 36–47. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73449-9_5
7. Bentea, L., Ölveczky, P.C.: A probabilistic strategy language for probabilistic rewrite theories and its application to cloud computing. In: Martí-Oliet, N., Palomino, M. (eds.) WADT 2012. LNCS, vol. 7841, pp. 77–94. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37635-1_5
8. Borovanský, P., Kirchner, C., Kirchner, H., Ringeissen, C.: Rewriting with strategies in ELAN: a functional semantics. Int. J. Found. Comput. Sci. **12**(1), 69–95 (2001). https://doi.org/10.1142/S0129054101000412
9. Braga, C., Verdejo, A.: Modular structural operational semantics with strategies. In: van Glabbeek, R., Mosses, P.D. (eds.) Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, 26 August 2006. Electronic Notes in Theoretical Computer Science, vol. 175, no. 1, pp. 3–17. Elsevier (2007). https://doi.org/10.1016/j.entcs.2006.10.024
10. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Sci. Comput. Program. **72**(1–2), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003
11. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R. (eds.): Handbook of Model Checking. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8
12. Clavel, M.: Strategies and user interfaces in Maude at work. In: Gramlich, B., Lucas, S. (eds.) Proceedings of the 3rd International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2003, Valencia, Spain, 8 June 2003. Electronic Notes in Theoretical Computer Science, vol. 86, no. 4, pp. 570–592. Elsevier (2003). https://doi.org/10.1016/S1571-0661(05)82612-X
13. Clavel, M., et al.: Maude manual v3.2.1 (2022-02). http://maude.cs.illinois.edu
14. Clavel, M., Meseguer, J.: Reflection and strategies in rewriting logic. In: Meseguer, J. (ed.) Proceedings of the First International Workshop on Rewriting Logic and its Applications, WRLA 1996, Asilomar, California, 3–6 September 1996. Electronic Notes in Theoretical Computer Science, vol. 4, pp. 126–148. Elsevier (1996). https://doi.org/10.1016/S1571-0661(04)00037-4
15. Durán, F., et al.: Programming and symbolic computation in Maude. J. Log. Algebraic Methods Program. **110** (2020). https://doi.org/10.1016/j.jlamp.2019.100497
16. Eker, S., Martí-Oliet, N., Meseguer, J., Verdejo, A.: Deduction, strategies, and rewriting. In: Archer, M., de la Tour, T.B., Muñoz, C. (eds.) Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, 16 August 2006. Electronic Notes in Theoretical Computer Science, vol. 174, no. 11, pp. 3–25. Elsevier (2007). https://doi.org/10.1016/j.entcs.2006.03.017
17. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Gadducci, F., Montanari, U. (eds.) Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, 19–21 September 2002. Electronic Notes in Theoretical Computer Science, vol. 71, pp. 162–187. Elsevier (2004). https://doi.org/10.1016/S1571-0661(05)82534-4
18. Fernández, M., Kirchner, H., Pinaud, B.: Strategic port graph rewriting: an interactive modelling framework. Math. Struct. Comput. Sci. **29**(5), 615–662 (2019). https://doi.org/10.1017/S0960129518000270

19. Hensel, C., Junges, S., Katoen, J.P., Quatmann, T., Volk, M.: The probabilistic model checker Storm. Int. J. Softw. Tools Technol. Transf. **23**(4), 1–22 (2021). https://doi.org/10.1007/s10009-021-00633-z
20. Hernández Cerezo, A.: Strategies for implementing SAT algorithms in rewriting logic. Bachelor's thesis, Universidad Complutense de Madrid (2020). https://eprints.ucm.es/63693
21. Hidalgo-Herrero, M., Verdejo, A., Ortega-Mallén, Y.: Using Maude and its strategies for defining a framework for analyzing Eden semantics. In: Antoy, S. (ed.) Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, 11 August 2006. Electronic Notes in Theoretical Computer Science, vol. 174, no. 10, pp. 119–137. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.02.051
22. Kowalski, R.A.: Algorithm = logic + control. Commun. ACM **22**(7), 424–436 (1979). https://doi.org/10.1145/359131.359136
23. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_47
24. Lescanne, P.: Implementation of completion by transition rules + control: ORME. In: Kirchner, H., Wechler, W. (eds.) ALP 1990. LNCS, vol. 463, pp. 262–269. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53162-9_44
25. Lucas, S.: Context-sensitive rewriting. ACM Comput. Surv. **53**(4), 78:1–78:36 (2020). https://doi.org/10.1145/3397677
26. Marin, M., Kutsia, T.: Foundations of the rule-based system ρLog. J. Appl. Non Class. Log. **16**(1–2), 151–168 (2006). https://doi.org/10.3166/jancl.16.151-168
27. Martí-Oliet, N., Meseguer, J., Verdejo, A.: Towards a strategy language for Maude. In: Martí-Oliet, N. (ed.) Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, 27 March–4 April 2004. Electronic Notes in Theoretical Computer Science, vol. 117, pp. 417–441. Elsevier (2004). https://doi.org/10.1016/j.entcs.2004.06.020
28. Martí-Oliet, N., Meseguer, J., Verdejo, A.: A rewriting semantics for Maude strategies. In: Roşu, G. (ed.) Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, 29–30 March 2008. Electronic Notes in Theoretical Computer Science, vol. 238, no. 3, pp. 227–247. Elsevier (2009). https://doi.org/10.1016/j.entcs.2009.05.022
29. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theor. Comput. Sci. **96**(1), 73–155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
30. Meseguer, J.: Twenty years of rewriting logic. J. Log. Algebr. Program. **81**(7–8), 721–781 (2012). https://doi.org/10.1016/j.jlap.2012.06.003
31. Rosa-Velardo, F., Segura, C., Verdejo, A.: Typed mobile ambients in Maude. In: Cirstea, H., Martí-Oliet, N. (eds.) Proceedings of the 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan, 23 April 2005. Electronic Notes in Theoretical Computer Science, vol. 147(1), pp. 135–161. Elsevier (2006). https://doi.org/10.1016/j.entcs.2005.06.041
32. Rubio, R.: Model checking of strategy-controlled systems in rewriting logic. Ph.D. thesis, Universidad Complutense de Madrid (2022). https://eprints.ucm.es/71531
33. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Model checking strategy-controlled rewriting systems. In: FSCD 2019 (2019). https://doi.org/10.4230/LIPIcs.FSCD.2019.31

34. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Parameterized strategies specification in Maude. In: Fiadeiro, J.L., Tutu, I. (eds.) WADT 2018. LNCS, vol. 11563, pp. 27–44. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23220-7_2
35. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: The semantics of the Maude strategy language. Technical report 01/21, Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2021). https://eprints.ucm.es/67449
36. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Strategies, model checking and branching-time properties in Maude. J. Log. Algebr. Methods Program. **123**, 100700 (2021). https://doi.org/10.1016/j.jlamp.2021.100700
37. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Metalevel transformation of strategies. J. Log. Algebr. Methods Program. **124**, 100728 (2022). https://doi.org/10.1016/j.jlamp.2021.100728
38. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Model checking strategy-controlled systems in rewriting logic. Autom. Softw. Eng. **29**(1), 1–62 (2021). https://doi.org/10.1007/s10515-021-00307-9
39. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Simulating and model checking membrane systems using strategies in Maude. J. Log. Algebr. Methods Program. **124**, 100727 (2022). https://doi.org/10.1016/j.jlamp.2021.100727
40. Santos-García, G., Palomino, M.: Solving Sudoku puzzles with rewriting rules. In: Denker, G., Talcott, C. (eds.) Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, 1–2 April 2006. Electronic Notes in Theoretical Computer Science, vol. 176, no. 4, pp. 79–93. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.06.009
41. Santos-García, G., Palomino, M., Verdejo, A.: Rewriting logic using strategies for neural networks: An implementation in Maude. In: Corchado, J.M., Rodríguez, S., Llinas, J., Molina, J.M. (eds.) International Symposium on Distributed Computing and Artificial Intelligence (DCAI 2008). Advances in Soft Computing, vol. 50, pp. 424–433. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-85863-8_50
42. Verdejo, A., Martí-Oliet, N.: Basic completion strategies as another application of the Maude strategy language. In: Escobar, S. (ed.) WRS 2011 (2012). https://doi.org/10.4204/EPTCS.82.2