Parameterized Strategies Specification in Maude

Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo

Parameterized Strategies Specification in Maude*

Rubén Rubio(✉), Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo

Universidad Complutense de Madrid, Spain
{rubenrub,narciso,ipandreu,jalberto}@ucm.es

Abstract. Strategies and parameterization are two convenient tools for building clear and easily configurable specifications of complex computational systems, compositionally. Parameterization is a widely used feature of the Maude rewriting framework, whose strategy language implementation we have recently completed with strategy modules.
This paper describes the Maude strategy language and the associated parameterization techniques. Then, the specification and analysis of some examples of strategy parameterized systems are shown.

Keywords: Rewriting logic · Strategies · Maude · Parameterized specification

1 Introduction

Strategies are ubiquitous in Computer Science. As recipes to tackle search problems and bound nondeterminism, they appear in algorithms, automatic deduction, language semantics, artificial intelligence, ... In *rewriting logic* [13], some of these examples are better specified compositionally, abstracting not only data representation and rules but also the way they are applied. This parametric control of the rewriting process is conveniently expressed using strategies that take other strategies as parameters.

Maude [6,7] is a declarative high-level language based on rewriting logic that allows the description, execution and analysis of concurrent and distributed system models at different levels. First, sorts, symbols, equations and membership axioms are expressed in terms of *membership equational logic* [3]. Then, we add rewrite rules to represent transitions of a concurrent system, which need neither be deterministic, nor confluent, nor terminating. Above this, we can control how rules are applied using a strategy language [8,11]. Its implementation, at the Core Maude level in C++, has been recently completed as an extension of Maude 2.7.1 [7].

The strategy language is based in its authors experience with strategies in Maude and in previous strategy languages like ELAN [2] and Stratego [5]. It

* Research partially supported by MCIU Spanish project *TRACES* (TIN2015-67522-C3-3-R), and by Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731).

has already been exploited in the specification of algorithms, inference systems, and language semantics: Milner's CCS [12], the ambient calculus [12], the semantics of the parallel functional language Eden [9], equational logic completion procedures [16], a proof calculus for membrane systems [1], etc. These examples are likely to be expressed and generalized using control parameterization with strategies, whose implementation was not available at that time. Once expressed in this way, the specified systems can be both executed and tested with different alternative strategies provided as parameters, or analyzed at different levels with specific tools, like a model checker.

The next section introduces Maude, its strategy language, and the rudiments of parameterization. A generic backtracking scheme serves as an introductory example. The following sections describe some other examples of parameterized systems, targeting the simplex algorithm, the λ-calculus, and a functional program interpreter. These and more examples can be downloaded from [15], as well as the current version of Maude with full strategy support.

2 Maude

A Maude program consists of a hierarchy of modules, describing the data representation and behavior of the system specified. There are different module types for different specification levels.

Functional modules define membership equational logic theories, whose signature (K, Σ, S) consists of a set of kinds K, a many-kinded collection of operators $\Sigma = \{\Sigma_{k_1 \cdots k_n, k} : (k_1 \cdots k_n, k) \in K^* \times K\}$, and $S = \{S_k : k \in K\}$ a many-kinded set of partially ordered sorts. Equations and sort membership axioms E are defined on them

$$(\forall X) \qquad \begin{matrix} t = t' \\ t : s \end{matrix} \quad \text{if} \quad \bigwedge_i u_i = u_i' \ \wedge \ \bigwedge_j v_j : s_j$$

In addition, operators can be annotated with structural *axioms*, like commutativity (`comm`), associativity (`assoc`), and identity (`id`). For example, the following functional module specifies sets of integer numbers using both equations and axioms.

```
fmod INT-SET is
  protecting INT .    *** INT module importation
  sort IntSet .
  subsort Int < IntSet .
  *** IntSet contructors (ctor)
  op empty : → IntSet [ctor] .
  op __ : IntSet IntSet → IntSet
          [ctor assoc comm id: empty] . *** union
  var X : Int .
  eq X X  = X .
endfm
```

Functional modules are bound to some executability requirements, like confluence, termination, and sort-decreasingness.

System modules describe rewriting logic theories $\mathcal{R} = (\Sigma, E \cup A, R)$, adding rewriting rules R on top of the equational theory. Rules do not have to be either confluent or terminating, so they are likely to express non-deterministic behavior.

$$(\forall X) \qquad t \Rightarrow t' \;\text{ if }\; \bigwedge_i u_i = u'_i \;\wedge\; \bigwedge_j v_j : s_j \;\wedge\; \bigwedge_k w_k \Rightarrow w'_k$$

Anyhow, rules are required to be coherent with equations and axioms [7, §5.3]. Conditions of the third type are called *rewriting conditions*, and hold true iff the term w_k can be rewritten to match the term w'_k.

```
mod SUM-SET is
  protecting INT-SET .
  vars X Y : Int .
  rl [sum] : X Y ⇒ X + Y . *** matches any two elems
endm
```

The module `SUM-SET` above introduces a rule `sum` that takes two integers in the set and replaces them by their sum.

Strategy modules allow finer control of the rule rewriting process by means of the strategy language and recursive strategy definitions. They are described in the next section.

2.1 The strategy language

Rules in rewriting logic can be applied in any order, at any position, and with different matches. Maude provides various commands, like `rewrite` and `frewrite`, that execute all available rules against a given term until none can be applied or an optional step bound is reached. They use some internal fixed criteria to select the next rule application. In turn, the `search` command explores all possible rewriting paths to find a target term meeting some conditions. However, the specifier is sometimes interested in imposing constrains on the allowed execution paths, like a particular precedence of rules, or to which subterms they must be applied, etc. This is when strategies get inside the game.

From the point of view of the results, a strategy α is an operation transforming a term t into a set of terms, since the restrictions need not make the process deterministic. Strategies can be executed with the command `srewrite` t `using` α. The most elementary strategy is rule application

$$\mathsf{top}(\mathit{label}\,[x_1 \texttt{<-}\, t_1 \texttt{,} \ldots \texttt{,}\, x_n \texttt{<-}\, t_n]\,\{\alpha_1 \texttt{,} \ldots \texttt{,} \alpha_m\}),$$

that executes any available rules with label *label* on any subterm of the subject term. Variables in the rule and its condition can be instantiated before application with the substitution that maps x_k to t_k, and if the rule contains any rewriting condition, it must be controlled with a substrategy α_k. Moreover, the

optional top modifier restricts the application of the rule to the top of the subject term. A more powerful tool for selecting where to apply a strategy is the matchrew operator

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

It matches the pattern P on top of the subject term, and for each match satisfying the condition C, the subterms corresponding to $x_1,\ldots,x_n$ are rewritten using $\alpha_1,\ldots,\alpha_n$, and reassembled again. The operator name can be prefixed by `a` or `x` to match anywhere within the term or modulo structural axioms. Similar format follow the tests **match** P **s.t.** C, to check if P matches the subject term and satisfies C. Regular expressions are included in the strategy language by means of the alternation $\alpha\,|\,\beta$, the concatenation $\alpha\,;\,\beta$, the Kleene star α^*, and the constants **idle** and **fail**. A conditional strategy $\alpha\,?\,\beta : \gamma$ is also available. It executes α and then β on its results, but if α does not produce any, it applies γ to the initial term.

The last ingredient of the strategy language are potentially recursive named strategies. These strategies are defined in strategy modules, which have the form **smod** `M` **is** `...` **endsm** and contain:

- Strategy declarations **strat** `sname : T1 ... Tn @ T`, which state their parameter types T_1 to T_n, and the type T of the subject terms the strategy will be applied to.
- Strategy definitions **sd** `sname(`$t_1,\ldots,t_n$`) :=` α. The free variables in the right-hand side strategy expression α must be included in the terms t_1 to t_n. Conditional strategy definitions, introduced by **csd**, are available too, and impose conditions as in regular equations. A named strategy can be given any number of strategy definitions, and all definitions whose left-hand side matches the call term will be executed.

Now, we will illustrate all the different strategy constructors described above with a very simple example. Consider the following system module:

```
mod SIMPLE is
  sort Term .

  ops a b c d : → Term [ctor] .
  op f : Term → Term [ctor] .
  op g : Term Term → Term [ctor] .

  vars X Y : Term .

  rl [ab] : a ⇒ b .          rl [ac] : a ⇒ c .
  rl [bc] : b ⇒ c .          rl [pf] : f(X) ⇒ X .
  rl [ad] : f(a) ⇒ d .
endm
```

The SIMPLE module defines some constants and functions to build terms, and various rules that transform them. For example, we can apply the pf rule to the term f(g(f(a), b)) by means of the srewrite command, whose initial keyword can be abbreviated to srew:

```
Maude> srew f(g(f(a), b)) using pf .

Solution 1
result Term: g(f(a), b)

Solution 2
result Term: f(g(a, b))

No more solutions.
Maude> srew f(g(f(a), b)) using pf[T <- a] .

Solution 1
result Term: f(g(a, b))

No more solutions.
```

Notice that all possible rewrites are returned as different solutions. In the first case, the rule can be applied both to the first and to the second occurrence of f. However, in the second case, the initial substitution restricts its application to the subterm f(a). Using the strategy **top**(pf) instead, pf is only applied on top and only g(f(a), b) is obtained. Finer precision on where to apply a strategy can be achieved by the subterm rewriting operator. For example, **matchrew** g(X, Y) **by** X **using** pf restricts the application to the first argument of g. Then g(f(b), f(c)) rewrites only to g(b, f(c)). Moreover, multiple subterms can be rewritten in parallel by strategies like

amatchrew g(X, Y) **by** X **using** pf, Y **using** bc,

which transforms the initial term f(g(f(a), b)) in f(g(a, c)).

Once we are able to apply strategies to particular subterms, we need regular expressions to compose rewriting sequences. If we rewrite g(a, b) by the concatenation ab ; bc, we obtain g(c, b) and g(b, c), since the rule bc has been applied after ab, at the same or at a different position. Instead, when executing the alternation ab | bc, only one of these is applied, and we get g(b, b) and g(a, c) as solutions. Finally, the iteration pf* applies pf zero or more times consecutively, so it rewrites f(f(a)) to f(f(a)), f(a), and a.

Other useful resources are the tests like **match** g(X, Y) **s.t.** X $\neq$ Y. It does not produce any solution for g(a, a), but for g(a, b), the solution is the term itself. Tests easily combine with conditionals. For example, the strategy **match** f(X) ? ab : bc applies ab if the subject term top symbol is an f, and otherwise executes bc. Then, f(a) rewrites to f(b), b to c, and no solution is produced for a. However, conditionals can be used with any strategy as condition, as we will see soon and throughout the paper.

Finally, to define named and recursive strategies, we need to write strategy modules like the following:

```
smod SIMPLE-STRAT is
  protecting SIMPLE .    *** controls SIMPLE

  strat rewrite @ Term .
  sd rewrite := all ? rewrite : idle .
endsm
```

where `all` is a built-in constant that executes any available rule. In this case, it can be seen as equivalent to `ab | ac | bc | pf | ad`. Hence, the strategy `rewrite` applies any rule until no more rules can be applied, as the usual `rewrite` command does. Running this strategy for the term `g(f(a), f(b))` we obtain:

```
Maude> srew g(f(a), f(b)) using rewrite .

Solution 1
result Term: g(d, c)

Solution 2
result Term: g(c, c)

No more solutions.
```

In Figure 1, the strategy language semantics is described in brief. Apart from the initial term, the semantic function $[\![\alpha]\!] : (X \to T_{\Sigma/E}) \times T_{\Sigma/E} \to \mathcal{P}(T_{\Sigma/E})$ receives a substitution by way of variable environment, because function calls and `matchrew`s bind variables. The semantics of the strategy definitions δ is calculated as a fixed point of a continuous operator, and here $\Delta(sl, t_1 \cdots t_n)$ refers to the set of strategy definitions for sl whose left-hand side matches $t_1, \ldots, t_n$, along with the corresponding substitution σ.

2.2 Parameterization

Parameterization is achieved using three basic building blocks: theories, views, and parameterized modules [6,7]. A parameterized module is a usual module taking a set of formal parameters, bound to some theories.

mod `NAME{X1 :: TH1, ..., XN :: THN}` **is** `...` **endm**.

A theory declares the interface, the syntactic and semantic requirements, the actual parameter must respect. Each module type (functional, system, strategy) has its theory counterpart. They are structurally identical, but theories are not required to fulfill the executability properties of a module. Module parameters can only be bound to theories of their type or simpler types. Finally, a view is the way to express how a module honors a theory, mapping each sort, operator, and strategy declared in the theory to the actual value in the module. Then, views are used to instantiate parameterized modules.

As an example, the most basic predefined theory is `TRIV`, which declares a single sort.

$$[\![\textsf{idle}]\!](\theta,t) = \{t\} \qquad [\![\textsf{fail}]\!](\theta,t) = \emptyset$$

$$[\![\alpha\ ;\ \beta]\!](\theta,t) = \bigcup_{t' \in [\![\alpha]\!](\theta,t)} [\![\beta]\!](\theta,t') \qquad [\![\alpha\ \mathsf{|}\ \beta]\!](\theta,t) = [\![\alpha]\!](\theta,t) \cup [\![\beta]\!](\theta,t)$$

$$[\![\alpha *]\!](\theta,t) = \bigcup_{n \geq 0} [\![\alpha]\!]^n(\theta,t)$$

$$[\![\alpha\ ?\ \beta : \gamma]\!](t) = \begin{cases} [\![\alpha ; \beta]\!](\theta,t) & \text{if } [\![\alpha]\!](\theta,t) \neq \emptyset \\ [\![\gamma]\!](\theta,t) & \text{if } [\![\alpha]\!](\theta,t) = \emptyset \end{cases}$$

$$[\![\textsf{match } P \textsf{ s.t. } C\ ]\!](\theta,t) = \begin{cases} \{t\} & \text{if } \mathrm{match}(P,t,C,\theta_{-\mathrm{occur}(P)}) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\textsf{matchrew } P \textsf{ s.t. } C \textsf{ by } x_1 \textsf{ using } \alpha_1\ \textsf{,}\ldots\textsf{,}\ x_n \textsf{ using } \alpha_n]\!](\theta,t) =$$

$$\bigcup_{\sigma \in \mathrm{match}(P,t,C,\theta_{-\mathrm{occcur}(P)})} \bigcup_{t_1 \in [\![\alpha_1]\!](\sigma\circ\theta,\sigma(x_1))} \cdots \bigcup_{t_n \in [\![\alpha_n]\!](\sigma\circ\theta,\sigma(x_n))} P[x_1/t_1,\ldots,x_n/t_n]$$

$$[\![sl(t_1,\ldots,t_n)]\!](\theta,t) = \bigcup_{(\delta,\sigma) \in \Delta(sl,\theta(t_1)\cdots\theta(t_n))} [\![\delta]\!](\sigma,t)$$

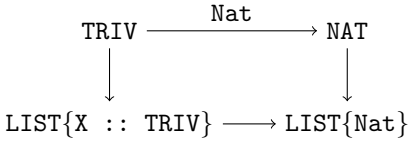
Fig. 1. Strategic set-theoretic semantics definitions

```
fth TRIV is
  sort Elt .
endfth
```

The module `NAT` can be viewed as a `TRIV`:

```
view Nat from TRIV to NAT is
  sort Elt to Nat .
endv
```

Then, the view `Nat` can be used to instantiate modules like lists `LIST{Nat}`, sets `SET{Nat}`, ... Module instantiation is based on the pushout along a view:



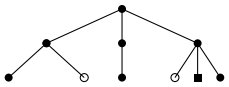
A simplified version of the `LIST` module is the following:

```
fmod LIST{X :: TRIV} is
  sort List .
  subsort X$Elt < List .
  op nil : → List [ctor] .
  op __ : List List → List [ctor assoc id: nil] .
endfm
```

Inside parameterized modules, sorts coming from theories are accessed by prefixing their names by their parameter name followed by a `$` sign. Operators and strategies retain their names as in the parameter theory. The Maude standard prelude includes some general theories like `TRIV`, `TOTAL-PREORDER`, `STRICT-WEAK-ORDER`, ..., and many views from the standard data types to those theories [6,7].

3 An introductory example

A simple example of parameterized control is the generic backtracking scheme.



The abstract backtracking problem is specified in two nested theories, the first proclaims the functional requirements, and the second extends it with the strategic ones. They require a `State` sort for states, and a strategy `expand` to rewrite a state to any of its direct successors, non-deterministically. We admit that the generated successors may not be valid, so we require a predicate `isOk` to test them, and also a predicate `isSolution` to determine whether a given state is already a solution.

```
fth BT-ELEMS-BASE is
  protecting BOOL .
  sort State .
  op isOk : State → Bool .
  op isSolution : State → Bool .
endfth

sth BT-ELEMS is
  including BT-ELEMS-BASE .
  strat expand @ State .
endsth
```

Then, a parameterized module `BT-STRAT`, given the specification of the problem following `BT-ELEMS`, defines a strategy `solve` for executing the backtracking algorithm. This is the following module, where parameters are highlighted in italics.

```
smod BT-STRAT{X :: BT-ELEMS} is
  strat solve @ X$State .
  var S : X$State .
  sd solve := (match S s.t. isSolution(S)) ? idle
       : (expand ; match S s.t. isOk(S) ; solve) .
endsm
```

The `solve` strategy is recursive. It concludes successfully when it finds a solution. Otherwise, it applies the strategy `expand` to obtain a successor, tests

whether it is valid, and iterates the process by a recursive call. Rewriting paths are discarded either when a successor not satisfying `isOk` is reached or when the `expand` strategy does not provide any successor. Indeed, each result of the `expand` strategy opens a new branch of the execution tree.

Although we call it *backtracking*, the strategy is abstract enough to be interpreted or executed otherwise, depending on the order in which the branches are explored. For backtracking, we understand a depth-first exploration, going backwards to try another branch when the current one has been exhausted without finding a solution. However, the default strategy rewriting implementation obeys a fair scheduling, which combines depth and breadth-first search. Here, the fair execution policy will usually require more time and memory usage than a real backtracking.

The generic algorithm is useless without actual instances. They can be specified in separate modules or files, but finally they have to fit in the frame of an abstract backtracking problem by means of a view from `BT-ELEMS`. The following module specifies the search for a Hamiltonian cycle in a graph, i.e. a cycle visiting every vertex of the graph only once.

```
mod HAMILTONIAN is
 protecting LIST{Nat} .
 sorts Edge Adjacency Graph .
 subsort Edge < Adjacency .
 op e : Nat Nat → Edge [ctor comm] .
 op nil : → Adjacency [ctor] .
 op __ : Adjacency Adjacency → Adjacency
                          [ctor assoc comm id: nil] .

 op noCross : List{Nat} → Bool .
 vars K L N V : Nat .
 vars P Q R : List{Nat} .
 var As : Adjacency .
 eq noCross(P K Q K R) = false .
 eq noCross(P) = true [owise] .
 op graph : Nat Adjacency List{Nat} → Graph [ctor] .
 op isOk : Graph → Bool .
 op isSolution : Graph → Bool .
 eq isSolution(graph(N,As,V P V)) = N == size(V P) .
 eq isSolution(G:Graph) = false [owise] .
 eq isOk(graph(N, As, V P V)) = noCross(V P) .
 eq isOk(graph(N, As, P)) = noCross(P) [owise] .
 rl [next] : graph(N, e(V, K) As, P V)
              ⇒ graph(N, As, P V K) .
endm
```

The module defines `Graph`, including both the graph and the current path, which can be extended up to a Hamiltonian cycle using the rule `next` whenever possible. Finally, as we said, the problem has to be presented as a backtracking

instance using a view. Identity mappings do not need to be written, but we have included them to illustrate the syntax.

```
view HamiltonianBT from BT-ELEMS to HAMILTONIAN is
  sort State to Graph .
  op isOk to isOk .
  op isSolution to isSolution .
  strat expand to-expr next .
endv
```

This instance is specially simple, the `expand` strategy has been defined inline as the `next` rule. In general, more elaborated strategies can be defined in a strategy module. In that case, the strategy module must be the *to* part of the view and the strategy is mapped by **strat** `expand` **to** `sname`. This can be seen in other examples available in the webpage [15] like the labyrinth escape problem, the n-queens problem, the graph m-coloring problem, etc.

A refinement of backtracking is *branch and bound*. This algorithmic technique was also programmed with parameterized strategies. Its problem specification, its theories, include richer functional and strategic requirements, so we do not describe them here. In this case, we cannot use an `expand` strategy that non-deterministically evolves to a successor, we need to examine them all to decide which to explore first according to the rank function. Hence, we have experimented with multiple approaches using different problem signatures. These can also be downloaded from [15].

4 The simplex algorithm

The simplex method [14] is a well-known algorithm for solving linear programming problems, i.e. for finding solutions that maximize (or minimize) a linear functional subject to some linear constraints.

$$\begin{aligned} \max / \min \quad & c_1 \ x_1 + \cdots + c_n \ x_n \\ & a_{11}x_1 + \cdots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + \cdots + a_{2n}x_n = b_2 \\ & a_{31}x_1 + \cdots + a_{3n}x_n \geq b_3 \\ & x_1, \ldots, x_n \geq 0 \end{aligned}$$

Formulated by George Dantzig in the late 1940s, it has had many industrial applications. Here we present an executable linear programming solver written in Maude using this method. The various non-deterministic steps are controlled by strategies, which are configurable through parameterization.

Figure 2 shows all the specification modules. `LINPROG` includes some basic definitions of linear algebra (polynomials, inequalities, ..., and operations on them), as well as linear programming problems. Then, `SIMPLEX-TABLE` defines the simplex tables (the state of the algorithm) and some operations to obtain information and modify them. At the rule level, `SIMPLEX-CONSTR` defines how

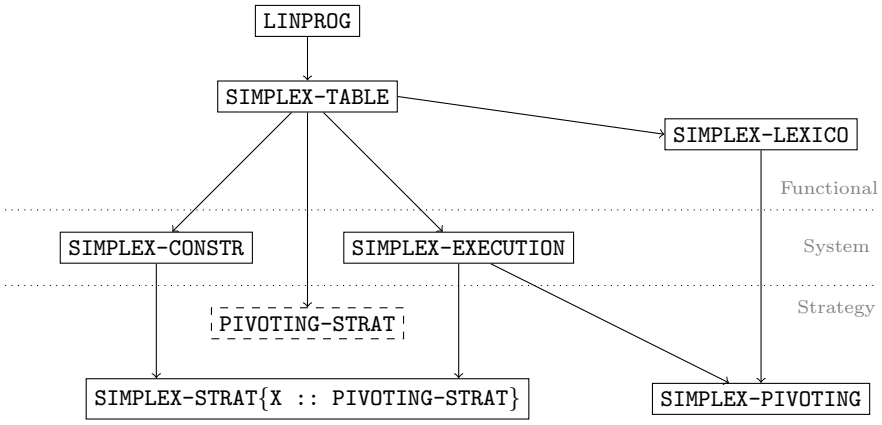


Fig. 2. Module structure for the simplex algorithm specification

to build a table from a linear programming problem, and `SIMPLEX-EXECUTION` provides the rules for the different actions of the simplex method. The most important is `pivot`, which changes the algorithm basis

```
crl [pivot] : Table ⇒ pivot(Table, Ve, Vl) if
                Ve, R := enterVars(Table) ∧
                Vl, S := leaveVars(Table, Ve) .
```

This rule is non-deterministic because `Ve` and `Vl` can be chosen among different alternatives. Pivoting carelessly may even lead to cycles, so that the algorithm may not terminate. Fortunately, there are different cycle prevention techniques to avoid non-termination. The more common ones are the *Bland rule* and the *lexicographic rule*. These rules (or any other) can be switched by means of parameterization.

The global solving process starts with a linear programming problem. The first stage is the generation of the simplex table, a non-deterministic process which produces equivalent tables up to renaming. So, we will concentrate on the second stage, the simplex method itself.

```
sd solve := makeTable ; simplex .
sd simplex := step ? simplex : idle .
sd step := (unbounded | finish | phase2 | unfeas)
           or-else pivotingStrat .
```

The `simplex` strategy executes a simplex step until no more can be applied. Each step first applies some rules that transform the table in some specific situations: `unbounded` detects when the problem is unbounded and transforms the table in a description of the beam of infinite improvement, and `finish` detects when the problem is already solved and presents the solution in a more readable form. The other rules are related to the *two phases method*, which may be sometimes needed to find an initial feasible solution of the linear system: `unfeas` signals

that the problem is unfeasible; and `phase2` transforms the table to a usual table whenever possible. Finally, if none of these (cheaper and concluding) rules can be applied, the pivoting strategy `pivotingStrat` is applied. This is the parameter of the `SIMPLEX-STRAT` module, provided by the theory `PIVOTING-STRAT`, which declares a single strategy without parameters applicable to the simplex table sort. For instance, the Bland rule is specified in `SIMPLEX-PIVOTING` with the following strategy

```
sd bland := matchrew T s.t.
              Ve := minVar(enterVars(T)) ∧
              Vl := minVar(leaveVars(T, Ve))
            by T using pivot[Ve <- Ve, Vl <- Vl] .
```

where `minVar` equationally computes the minimum variable within a set (a total order `<` is defined on them). A view `Bland` from `PIVOTING-STRAT` to `SIMPLEX-PIVOTING`, mapping `pivotingStrat` to `bland`, is then used to instantiate `SIMPLEX-STRAT`.

4.1 Parameterized analysis

A natural analysis of parameterized systems is the comparison of the behavior of their multiple instances, which can be executed or simulated, looking at the execution time, the number of rewriting steps, or the memory requirements. More essential properties can be inspected by tracing the state of the system while running the simulation. Strategies are a useful tool for that purpose. Apart from this, parameterized modules can be analyzed with specific tools like the Maude model checker.

For the simplex method, we have compared the performance of two pivoting strategies and free rewriting in terms of time and number of rewrites, against a linear programming exercises set. Since these properties depend on the actual Maude implementation, we also consider an intrinsic property of the algorithm: the number of iterations or pivoting steps until a solution is found. To obtain this attribute, we count the number of `pivot` rule executions with the aid of a parameterized analyzer module that maintains a pair `watch(T, N)` of a simplex table and a counter, applies the rules to the table using `matchrew`, and updates the counter accordingly. This can be compared with the unrestricted better case and worst case number of iterations, which can be calculated using strategies too.

	Free	Bland	Lexicographic
Iterations above better case	2.05	2.05	1.47
Number of rewrites	4246	5195	5191
Time (ms)	1.84	2.29	2.2

We observe that the lexicographic rule reduces the mean number of iterations, but since its decisions require more computations, its performance is similar to Bland's. The free strategy, executing the rules at the discretion of the Maude rewriting engine, performs quite well.

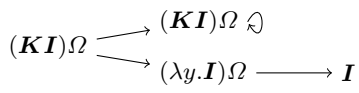
Regarding auxiliary tools, we have applied an experimental prototype of a strategy-aware model checker. Thanks to it, we can check whether the algorithm following a given strategy, for a fixed example, may cycle or not. Predictably, no example cycles with the Bland and lexicographic rules, but some do with free rewriting.

5 The λ-calculus

The λ-calculus can be easily expressed and executed in Maude [10]. β-reduction is the single rule of the rewriting system, so a term can be reduced by simply using the `rewrite` and `search` commands.

```
subsort Var < LambdaTerm .
op \_._ : Var LambdaTerm → LambdaTerm [ctor] .
op __ : LambdaTerm LambdaTerm → LambdaTerm [ctor] .
rl [beta] : (\ x . M) N ⇒ subst(M, x, N) .
```

However, which β-redex is reduced first is an important choice. Some reduction paths may not terminate while others reach an irreducible term, even though this is unique by the Church-Rosser property. The normalization theorem says that reducing the leftmost outermost redex first guarantees finding a normal form in case it exists.



$\boldsymbol{K} = \lambda x.(\lambda y.x)$
$\boldsymbol{I} = \lambda x.x$
$\Omega = (\lambda x.xx)(\lambda x.xx)$

We allow selecting which strategy `step` to use for a single reduction step and, using a parameterized module, we define a strategy that reduces the term to an irreducible form, if any.

```
strat reduce @ LambdaTerm .
sd reduce := step ! .
```

The operator $\alpha!$ executes α until it cannot be further applied. The different implemented strategies for a single reduction are:

- Applicative order (inner rightmost redex first)

```
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) .
```

- Normalizing strategy (outer leftmost redex first)

```
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 .
```

– By name (normalizing but no reduction inside abstraction)

```
sd byname := top(beta)
  or-else matchrew M N by M using byname
  or-else matchrew M N by N using byname .
```

– By value (only outermost redex and when argument is value)

```
sd byvalue := (match (\ x . M) z
  | match (\ x . M) (\ y . N)) ; top(beta) .
```

For example, **srew** `(K I) Omega` **using** `applicative` does not finish, but **srew** `(K I) Omega` **using** `normal` produces `\ x . x` as a result. Moreover, depending on the reduction strategy, we obtain different canonical forms. From `(K z) t` we get `z` using all strategies but `byvalue`, whose canonical form is the term unchanged.

6 Semantics of programming languages

The semantics of programming languages is an interesting field to apply both strategies [4] and parameterization. The Maude implementation of Eden [9], a concurrent language based on Haskell, can be cited as example. The general idea is that strategies control the execution process and can be used to tweak some semantic choices, and see how alternatives perform. Here we describe a simpler example, the *Recursion equations* (REC) language [17, Chapter 9].

A REC program consists of a closed integer expression to evaluate and a set of integer function definitions of the form $f(x_1, \ldots, x_{a_f}) = \langle expr \rangle$. Expressions contain integer constants, sums, products, subtractions, conditionals, and calls to any defined function. Hence, functions can be recursive and mutually recursive. An organized set of modules (see Figure 3) specifies the representation of REC programs and its basic rules:

```
rl [apply] : Q(Args) ⇒ apply(find(Q, Defs), Args)
                                          [nonexec] .
crl [cond] : if C then E else F ⇒
       if C == 0 then E else F fi if C : Int .
```

The rule `apply` replaces a function call `Q(Args)` by its definition according to the list of definitions `Defs`, substituting its variables by the call arguments using the equational function `apply`. This rule is not directly executable since `Defs`, absent in the left-hand side, must be provided from the context in the application strategy substitution, as we will see. Rule `cond` resolves a conditional when its condition expression has already been reduced to an integer. In any other case, integer expressions are reduced equationally, since the REC expressions sort `RecExpr` has been defined as an extension of the built-in sort `Int`. REC programs are executed following a `reduce` strategy that receives a list of function definitions as an argument. It is parameterized by a strategy `st` that is intended to expand function calls, and which we will later instantiate with `byvalue` and `byname` alternatives.

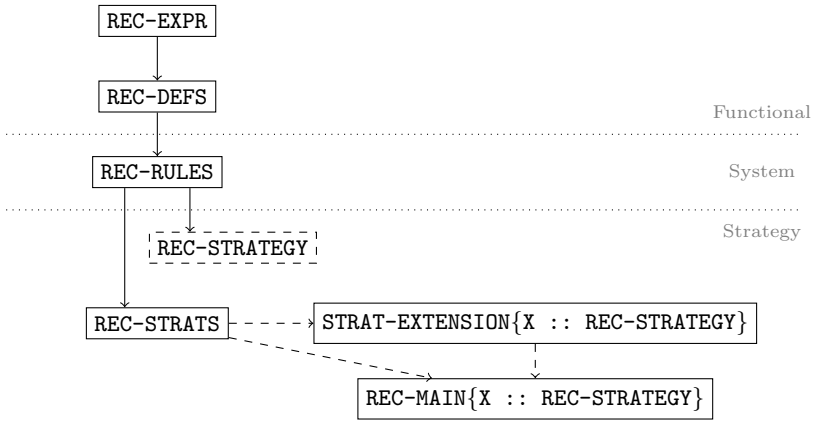


Fig. 3. Module structure for the REC language specification

```
smod REC-MAIN{X :: REC-STRATEGY} is
  strat reduce : List{FunctionDef} @ RecExpr .
  var FL : List{FunctionDef} .
  sd reduce(FL) := (cond or-else st(FL)) ! .
endsm
```

According to the `reduce` definition, conditionals and calls are reduced as long as possible, but conditionals are reduced first. This is convenient, since function calls may appear anywhere inside the integer expression, and a simple recursive example like the factorial

```
eq factorial = 'f('n) := if 'n then 1
                         else 'n * 'f('n-1) .
```

shows that expanding calls anywhere is problematic. In effect, the precedence of `cond` avoids the non-terminating reduction

$$\begin{aligned}\texttt{'f(0)} &\rightarrow_{\texttt{st}} \texttt{if 0 then 1 else 0 * 'f(-1)}\\ &\rightarrow_{\texttt{st}} \texttt{if 0 then 1 else 0 * (if -1 then 0 else 'f(-2))} \ldots\end{aligned}$$

Still, this is not enough when evaluating terms like `'f('f(0))`. So, in general, we must ensure not to reduce conditional branches until the condition is solved, no matter if calling by value or by name. To make strategies aware of these precautions while saving code, we will take advantage of strategy parameterization too: we define an additional parameterized module `STRAT-EXTENSION` that extends a strategy facing function calls `st` to a strategy `xst` that applies the reduction step to any program term, but reducing conditions first.

```
smod STRAT-EXTENSION{X :: REC-STRATEGY} is
 strat xst : List{FunctionDef} @ RecExpr .
 vars E F G : RecExpr .
```

```
  var  FL    : List{FunctionDef} .

  sd xst(FL) := st(FL)
   | matchrew E + F by E using xst(FL)
       or-else matchrew E + F by F using xst(FL)
   | matchrew E * F by E using xst(FL)
       or-else matchrew E * F by F using xst(FL)
   | (matchrew E - F by E using xst(FL))
       or-else matchrew E - F by F using xst(FL)
   | matchrew if E then F else G by E using xst(FL) .
endsm
```

Now, the strategy extension is a reusable component that can be used to define the alternative strategies succinctly, avoiding boilerplate code. First, we instantiate `STRAT-EXTENSION` with views for `byname` and `byvalue`, defined in `REC-STRATS`, to obtain the extended strategies. Then, we instantiate `REC-MAIN` with new views from `REC-STRATEGY` to the instantiated modules, mapping `st` to the extended strategies. On the contrary, the strategy `free` applies reduction carelessly and anywhere, so it instantiates `REC-MAIN` directly.

```
sd free(FL)    := apply[Defs <- FL] .
sd byname(FL)  := top(apply[Defs <- FL]) .
sd byvalue(FL) := (matchrew E, NeArgs
                     by E using byvalue(FL))
          or-else matchrew E, NeArgs
                     by NeArgs using byvalue(FL)
      | (matchrew Q(Args) by Args using byvalue(FL))
        or-else top(apply[Defs <- FL]) .
```

The `FL` parameter must be filled with the program function definitions. For example, with the Ackermann function defined below we execute

```
srew 1 + 'A(2, 3) using reduce(ackermann) .
```

and obtain `10`.

```
eq ackermann = 'A('m, 'n) := if 'm then 'n + 1 else
         (if 'n then 'A('m - 1, 1)
                 else 'A('m - 1, 'A('m, 'n - 1))) .

eq nameonly = ('f('x) := 'f('x) + 1) ('g('x) := 7) .
```

The differences between call by value and call by name are appreciated with the `nameonly` example, since $g(f(0))$ will be evaluated to 7 by name while it will never finish by value.

7 Conclusions

Rewriting strategies are a useful tool in the description of concurrent and logical systems, in accordance with the *separation of concerns* principle. A stratified

specification of rules and its global control allows discharging the functional and rule levels of improper complexity, and makes the specified systems more configurable and adaptable. On the other hand, parameterized modules are a basic feature for building complex systems, reusing abstract specifications, and switching between alternative components. Such specifications can be compared in their multiple instances or analyzed parametrically.

The Maude strategy language implementation is now complete, including parameterized strategy modules. In this paper we have seen some examples of parameterized specification with strategies related to algorithms and programming languages. In the future, the combination of strategies and parameterization can be used as a fundamental approach to specify more complex and interesting systems. Also, more elaborated analyses can be done on these, including model checking, whose strategy-aware implementation is currently under development.

References

1. Andrei, O., Lucanu, D.: Strategy-based proof calculus for membrane systems. In: Roşu, G. (ed.) Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008. ENTCS, vol. 238(3), pp. 23–43. Elsevier (2009). https://doi.org/10.1016/j.entcs.2009.05.011
2. 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
3. Bouhoula, A., Jouannaud, J.P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science **236**(1), 35–132 (2000). https://doi.org/10.1016/S0304-3975(99)00206-6
4. 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, August 26, 2006. ENTCS, vol. 175(1), pp. 3–17. Elsevier (2007). https://doi.org/10.1016/j.entcs.2006.10.024
5. Bravenboer, M., Kalleberg, K.T., Vermaas, R., Visser, E.: Stratego/XT 0.17. A language and toolset for program transformation. Science of Computer Programming **72**(1-2), 52–70 (2008). https://doi.org/10.1016/j.scico.2007.11.003
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude-A High-Performance Logical Framework, LNCS, vol. 4350. Springer (2007). https://doi.org/10.1007/978-3-540-71999-1
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (v2.7.1) (July 2016), http://maude.cs.uiuc.edu/
8. 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, August 16, 2006. ENTCS, vol. 174(11), pp. 3–25. Elsevier (2007). https://doi.org/10.1016/j.entcs.2006.03.017
9. 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, August 11, 2006. ENTCS, vol. 174(10), pp. 119–137. Elsevier (2007). https://doi.org/10.1016/j.entcs.2007.02.051
10. Martí-Oliet, N., Meseguer, J.: Rewriting Logic as a Logical and Semantic Framework, pp. 1–87. Springer Netherlands (2002). https://doi.org/10.1007/978-94-017-0464-9_1
11. 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, March 27-April 4, 2004. ENTCS, vol. 117, pp. 417–441. Elsevier (2004). https://doi.org/10.1016/j.entcs.2004.06.020
12. Martí-Oliet, N., Palomino, M., Verdejo, A.: Strategies and simulations in a semantic framework. Journal of Algorithms **62**(3), 95 – 116 (2007). https://doi.org/10.1016/j.jalgor.2007.04.002
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96**(1), 73 – 155 (1992). https://doi.org/10.1016/0304-3975(92)90182-F
14. Murty, K.G.: Linear programming. John Wiley & Sons, New York (1983)
15. Rubio, R., Martí-Oliet, N., Pita, I., Verdejo, A.: Strategy language for Maude web page, `http://maude.sip.ucm.es/strategies`
16. Verdejo, A., Martí-Oliet, N.: Basic completion strategies as another application of the Maude strategy language. In: Escobar, S. (ed.) Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011. EPTCS, vol. 82, pp. 17–36 (2011). https://doi.org/10.4204/EPTCS.82.2
17. Winskel, G.: The Formal Semantics of Programming Languages. Foundations of Computing, The MIT Press (1993)