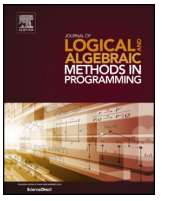
Journal of Logical and Algebraic Methods in Programming 134 (2023) 100887

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



The Maude strategy language

Steven Eker [a], Narciso Martí-Oliet [b], José Meseguer [c], Rubén Rubio [b,*], Alberto Verdejo [b]

[a] SRI International, Menlo Park, CA, USA
[b] Facultad de Informática, Universidad Complutense de Madrid, Spain
[c] University of Illinois at Urbana-Champaign, IL, USA

ARTICLE INFO

Article history:
Received 20 January 2023
Received in revised form 1 June 2023
Accepted 1 June 2023
Available online 7 June 2023

Keywords:
Formal specification
Rewriting logic
Rewriting strategies
Maude

ABSTRACT

Rewriting logic is a natural and expressive framework for the specification of concurrent systems and logics. The Maude specification language provides an implementation of this formalism that allows executing, verifying, and analyzing the represented systems. These specifications declare their objects by means of terms and equations, and provide rewriting rules to represent potentially non-deterministic local transformations on the state. Sometimes a controlled application of these rules is required to reduce non-determinism, to capture global, goal-oriented or efficiency concerns, or to select specific executions for their analysis. That is what we call a strategy. In order to express them, respecting the separation of concerns principle, a Maude strategy language was proposed and developed. The first implementation of the strategy language was done in Maude itself using its reflective features. After ample experimentation, some more features have been added and, for greater efficiency, the strategy language has been implemented in C++ as an integral part of the Maude system. This paper describes the Maude strategy language along with its semantics, its implementation decisions, and several application examples from various fields.

1. Introduction

Rewriting logic specifications describe computational systems as collections of rewriting rules that are applied to the terms of an equational specification. A great generality is achieved by allowing the rules not to be confluent or terminating. However, and specially when the specifications become executable, it may be convenient to control this non-determinism in order to avoid undesired evolutions. This control is realized by *strategies*, whose need has been identified since the first developments of rewriting logic and its implementations [38].

Maude [16] is a specification language based on rewriting logic as well as an interpreter with several formal analysis features and a formal tool environment [21] that allows executing, verifying, and analyzing the specified systems. The non-deterministic rewriting process can be explored in different ways in Maude. In some cases, it is enough to see where a single path of exhaustive rewriting leads to, and this is possible with the `rewrite` and `frewrite` commands that select the next rewrite by some fixed criteria ensuring some fairness properties of the chosen path. Sometimes, all possible

* Corresponding author.
E-mail addresses: eker@csl.sri.com (S. Eker), narciso@ucm.es (N. Martí-Oliet), meseguer@illinois.edu (J. Meseguer), rubenrub@ucm.es (R. Rubio), jalberto@ucm.es (A. Verdejo).

https://doi.org/10.1016/j.jlamp.2023.100887

execution paths need to be explored using the `search` command; for example, to search for a violation of an invariant. However, to observe paths satisfying some user-chosen restrictions, strategies are required. Due to the reflective properties of rewriting logic, and using some Maude *metalevel* functions that reproduce the rule application operations, strategies have been traditionally expressed as metacomputations inside the logic [17]. This mechanism is complete and powerful, but it is sometimes cumbersome for users that are not familiar with the metalevel, because of its conceptual complexity and the verbosity of its notation. Hence, following the influence of previous implementations of strategic rewriting such as ELAN [8], Stratego [12] and TOM [6], a simple but expressive object-level strategy language has been proposed for Maude. Its design is based on the previous Maude experience with strategies at the metalevel and on its predecessors, but it deviates from the treatment of strategies in ELAN in a remarkable aspect. In ELAN, rules and strategies are tightly coupled and strategies can be used in the definition of rules. The Maude language advocates a clear separation between them, making strategies a separate layer above rules. This is enforced and emphasized by banning strategies from *system modules*, i.e., from rewrite theory specifications. Instead, different kinds of strategies can be specified separately for the same system module in different *strategy modules*, so that the same rewrite rules can be controlled with different strategies by means of the chosen strategy module. Of course, for simplicity and ease of use, each strategy language design favors some features and may not directly support others. Nevertheless, besides the fact that Maude's strategy language supports user-definable recursive strategies, which may be used to specify various additional features, Maude itself, including its strategy language, is *user-extensible* thanks to its reflective nature, supported by Maude's `META-LEVEL` module, so that new features can be added by reflective extension.

Strategies are a useful specification and execution tool [7]. Its origins date back to combinatory logic and the λ-calculus, where choosing the next reduction position by a fixed structural criterion ensures finding a normal form in case it exists. In this vein, strategies can be used to make systems confluent or terminating by imposing additional restrictions. Moreover, they can provide specifications with a notion of global control over the eminently local meaning of rules in various ways: as simple as choosing a rule precedence, or arbitrarily complex by depending on the execution history. The *Rule + Strategies* approach [41,32], as an evolution of the Kowalski's motto *Algorithm = Logic + Control* [31], can be exploited to build specifications with a clear *separation of concerns*. Typically, it is easier to prove that the logical part given by rules ensures correct deductions or computations, while leaving to the strategies the responsibilities on efficiency and goal directedness (as shown in Section 4.4). In some other cases, strategies are actually needed to ensure that the system behaves as intended. In addition, strategies can be used to analyze plain rewriting logic specifications too, by the observation of selected rewriting evolutions. Many examples from different fields have already been specified using the Maude strategy language: it has been applied to express deduction procedures as strategies controlling a declarative inference system [57,23], to specify aspects of the semantics of programming languages [26,11], of the ambient calculus [44], Milner's CCS [37], process scheduling policies [52],[1] membrane systems [3,53], etc.

The first prototype of the strategy language was written at the metalevel as an extension of Full Maude [16]. Now, the complete language is available and efficiently supported at the Maude interpreter level, implemented in C++, from its version 3.0 onwards [15,19]. The most recent version of Maude can be downloaded from `maude.cs.illinois.edu`, and the examples described in this article and many more are available at `maude.ucm.es/strategies` and `github.com/fadoss/strat-examples`.

This article extends, updates, and integrates the information already presented in some workshop and conference talks [35,23,36,48,37], and a PhD thesis [45]. For the first time, this article describes strategy modules and the metarepresentation of the strategy language, and provides a complete and formal denotational semantics of the language with strategy calls. More details are given on the strategy language constructors, and design and implementation decisions are discussed. New and significantly improved examples are included, as well as a comparison with the similar strategy languages of ELAN, Stratego, TOM, and ρLog.

2. Rewriting logic and Maude

Rewriting logic [38,39] is a computational logic for expressing both concurrent computation and logical deduction in a general and natural way. Maude [15,16] is a specification language whose programs are exactly rewrite theories. These can be executed in the language interpreter, and analyzed and verified by means of various formal analysis features in Maude itself, as well as by several formal analysis tools in Maude's formal tool environment [21]. Typically, those formal tools are themselves implemented in Maude through reflection. Maude has already been applied to the specification and verification of many logics, models of concurrency, programming languages, hardware and software modeling languages, distributed algorithms, network protocols, cryptographic protocols, real-time and cyber-physical systems, and biological systems (see [39] for a survey of rewriting logic, Maude, and its applications).

[1] Process scheduling policies, implemented by operating systems to distribute the processor time among multiple simultaneous processes, are an archetypical example of strategies programmed in real software using standard programming languages. Specifying these strategies at a higher level in Maude can be useful for better reasoning about them.

2.1. Abstract reduction

First, some standard notation and terminology will be reviewed. Given a set S of states and a binary relation $(\rightarrow) \in S \times S$, we refer to $s \rightarrow s'$ as a reduction or execution step, and $s_0 \rightarrow \cdots \rightarrow s_n$ as a derivation or execution of length n. Non-terminating executions of the form $s_0 \rightarrow s_1 \rightarrow \cdots$ are also considered. We say that $s \in S$ is *irreducible* if no $s' \in S$ makes $s \rightarrow s'$ hold. We write $s \rightarrow^n s'$ if there is a derivation of length n from s to s', $\rightarrow^+$ for the transitive closure of $\rightarrow$ defined as $\cup_{n \geq 1}(\rightarrow^n)$, and $\rightarrow^*$ for the transitive and reflexive closure of $\rightarrow$. We also write $s \rightarrow^! s'$ if $s \rightarrow^* s'$ and s' is irreducible, and say that s' is a *normal form* of s.

The pair $(S, \rightarrow)$ is an *abstract reduction system*. It is *confluent* if for all $s, s_1, s_2 \in S$ such that $s \rightarrow^* s_1$ and $s \rightarrow^* s_2$, there is a s' such that $s_1 \rightarrow^* s'$ and $s_2 \rightarrow^* s'$. It is *terminating* if every execution from any state s is finite. In a confluent and terminating reduction system, also known as *convergent*, every state has a single normal form. Moreover, given another relation $\rightarrow'$ on S, we say $\rightarrow'$ is *coherent* [20] with $\rightarrow$ if for all $s, u, s_1, s_2 \in S$ such that $s \rightarrow^! u$, $s \rightarrow' s_1$, and $u \rightarrow' s_2$, there is a u' such that $s_1 \rightarrow^! u'$ and $s_2 \rightarrow^! u'$. In simpler words, $\rightarrow'$ is coherent with $\rightarrow$ if $\rightarrow'$ can be applied on the canonical forms of $\rightarrow$ without loss of generality.

2.2. Membership equational logic

A *signature* in membership equational logic [9] is given by a set of *sorts* S and a collection Σ of operators $f : s_1 \cdots s_n \rightarrow s$ from which terms are constructed. Sorts are related by a partial order $s_1 < s_2$ representing subsort inclusion. Each connected component in the sort relation is called a *kind*, and the kind of a sort s is denoted by $[s]$. Each operator $f : s_1 \ldots s_n \rightarrow s$ is also lifted to the kind level as $f : [s_1] \ldots [s_n] \rightarrow [s]$. Kinds are interpreted as sets containing all the well-formed terms of the related sorts, as well as some error elements that may arise from partially-defined functions or type errors. Sorts and kinds are uniformly referred to as *types*. The set of terms of a given type s over some variables X is written $T_{\Sigma,s}(X)$ and the full set of terms is written $T_\Sigma(X)$. Terms without variables are called *ground terms*. A *substitution* is a type-preserving function $\sigma : X \rightarrow T_\Sigma(X)$ that assigns terms to variables. It can be extended to a function $\overline{\sigma} : T_\Sigma(X) \rightarrow T_\Sigma(X)$ that replaces the occurrences of the (typed) variables in a term inductively. For any pair of substitutions σ_1, σ_2, we define their composition $\sigma_2 \circ \sigma_1$ by the equality $(\sigma_2 \circ \sigma_1)(x) := \overline{\sigma_2}(\sigma_1(x))$. It satisfies $\overline{\sigma_2 \circ \sigma_1} = \overline{\sigma_2} \circ \overline{\sigma_1}$ in the usual functional sense. The line over the extension is usually omitted.

Membership equational logic for a given theory (Σ, E) has two kinds of atomic sentences, *equations* and sort *membership axioms*, on top of the above signature. They can be conditioned by premises in the form of other equations and sort membership axioms to yield Horn clauses of the form:

$$t = s \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \qquad \text{and} \qquad t : s \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j$$

where $t : s$ is the membership axiom stating that t has sort s. Apart from equations and membership axioms, operators can be annotated with *structural axioms* like associativity, commutativity, and identity. Of course, such structural axioms are a special case of equations. However, in Maude they are specified together with their corresponding operators because they are used in a built-in way to efficiently support deduction *modulo* such structural axioms.

These (possibly conditional) equations and memberships E, and structural axioms B induce an equality relation $=_{E \cup B}$ that identifies different terms up to provable equality with $E \cup B$. The *initial term algebra* $T_{\Sigma/E \cup B}$ is the quotient of the ground terms $T_\Sigma(\emptyset)$ modulo this equality relation. Its elements $[t]$ are equivalence classes modulo $=_{E \cup B}$, but we will usually write simply t when dealing with well-defined characteristics and operations that are independent of the class representatives.

2.3. Rewriting logic

A *rewrite theory* $\mathcal{R} = (\Sigma, E \cup B, R)$ is a membership equational theory $(\Sigma, E \cup B)$ together with a set R of rewriting rules, which are interpreted as non-equational transitions. A possibly conditional rewriting rule has the form:

$$l \Rightarrow r \quad \text{if} \bigwedge_i u_i = u_i' \wedge \bigwedge_j v_j : s_j \wedge \bigwedge_k w_k \Rightarrow w_k'$$

The application of a rule to a term t is the replacement of an instance (modulo B) of l in some position p of t by r at that same position p instantiated accordingly if the condition holds. Conditions of the third type are named *rewriting conditions*. They are satisfied if the instance of each w_k can be rewritten in zero or more steps to match an instance (modulo $E \cup B$) of w_k'. Formally, a rule as above rewrites a term t to t', $t \rightarrow_{\mathcal{R}} t'$, if there is a position p in t and a substitution $\sigma : X \rightarrow T_\Sigma(X)$ such that $t|_p =_{E \cup B} \sigma(l)$ and $t' =_{E \cup B} t[\sigma(r)]_p$, and $\sigma(u_i) =_{E \cup B} \sigma(u_i')$ for all i, $\sigma(v_j) \in T_{\Sigma,s_j}(\emptyset)$ for all j, and $\sigma(w_k) \rightarrow^*_{\mathcal{R}} \sigma(w_k')$ for all k.

For execution purposes, since the equality relation $=_{E \cup B}$ can be undecidable, equations are handled as oriented rewrite rules $\rightarrow_{E/B}$ modulo structural axioms. Moreover, some additional executability requirements are given on $\mathcal{R}$, namely (1) $(\Sigma, \rightarrow_{E/B})$ is assumed *convergent* modulo B; and (2) the rules R are assumed *coherent* modulo B with respect to the (oriented) equations E. This second requirement means that the relation $\rightarrow_{R/B}$, consisting in $\rightarrow_{\mathrm{R}}$ with $=_{E \cup B}$ replaced by

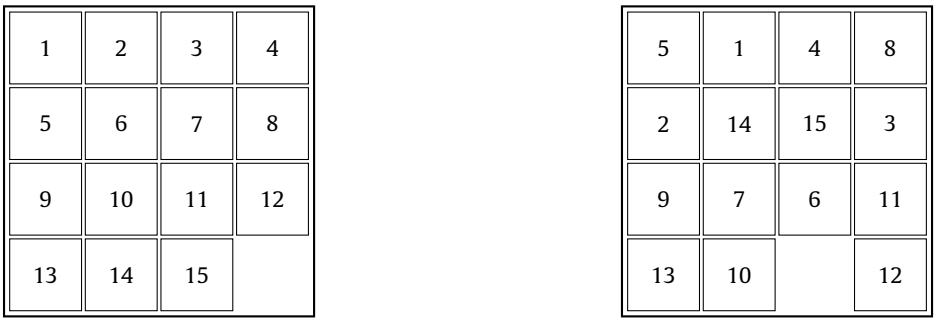


Fig. 1. The 15-puzzle in its goal position and in a shuffled one.

$=_B$, is coherent with $\rightarrow_{E/B}$ as defined in Section 2.1. The effect of assumptions (1)-(2) is that $\rightarrow_{\mathcal{R}}$ can be computed in terms of the much simpler and decidable equality relation $=_B$, instead of requiring the complex relation $=_{E\cup B}$, as $\rightarrow_{R/B} \circ \rightarrow^{!}_{E/B}$. These executability requirements are expressed at the Maude level by the notion of *admissible* Maude modules, as explained in Section 2.4.

2.4. Maude

Maude is a specification language [15] whose programs are a straight translation to ASCII of the previous mathematical notation for equations, membership axioms, and rules. Specifications are organized in modules that can include or extend other modules. Some useful modules are included in the Maude *prelude* specifying natural and floating-point numbers, lists, strings, sets, reflective operations, and so on. Pure equational theories are specified in *functional modules*, introduced by the `fmod` keyword.

The following functional module specifies the board of the 15-puzzle game, which consists of 15 sliding numbered square tiles lying in a 4×4 frame, as depicted in Fig. 1. The puzzle has been represented as a semicolon-separated list of rows, each a list of tiles, which are in turn either a natural number or a blank `b`. Declarations and statements can be annotated with attributes between brackets. The `ctor` attribute for operator declarations states that the operator is intended to be a data constructor of the range sort. Structural axioms of commutativity, associativity, and identity element e are respectively written as operator attributes `comm`, `assoc`, and `id: e`. The syntax for conditions and membership axioms follows directly from the mathematical notation of the previous section, except that a special form of equality condition `l := r`, known as *matching condition*, allows free variables in its lefthand side to be bound by matching. Apart from this exception, all variables in the condition and the righthand side of an equation should appear in the lefthand side of that equation.

```
fmod 15PUZZLE-BOARD is
  protecting NAT .                                              *** module inclusion

  sorts Tile Row Puzzle .                                       *** sort declarations
  subsorts Nat < Tile < Row < Puzzle .                          *** subsorts

  op b : -> Tile [ctor] .                                     *** operator declarations
  op nil : -> Row [ctor] .
  op __ : Row Row -> Row [ctor assoc id: nil prec 25] .
  op _;_ : Puzzle Puzzle -> Puzzle [ctor assoc] .

  var T : Tile .                                              *** variables
  var R : Row .                                               *** (local to the module)

  op size : Row -> Nat .
  eq size(nil) = 0 .                                                    *** equations
  eq size(T R) = size(R) + 1 .
endfm
```

Not all functional modules are *admissible*, as explained in Section 2.3. Maude uses equations as left-to-right simplification rules, which are applied exhaustively to obtain canonical forms of the terms modulo the equations and axioms. For this procedure to be sound, the simplification relation $\rightarrow_{E/B}$ should be confluent and terminating modulo the axioms B (in this example, the specified associativity and identity axioms). Maude does not check this automatically, since it is generally undecidable, but the Maude Formal Environment [21] includes tools for checking these properties.

System modules specify rewrite theories by adding rules. In this case, the rule application relation $\rightarrow_{\mathcal{R}}$ need not be confluent nor terminating. However, Maude works efficiently by reducing a term to normal form before applying a rule, so that an $\mathcal{R}$-rewriting step is achieved by the much simpler relation $\rightarrow_{R/B} \circ \rightarrow^{!}_{E/B}$. This assumes that the rules R are coherent

with the equations E modulo structural axioms B, i.e., the second requirement in the last paragraph of Section 2.3 must be satisfied.

For example, rules can be added to the functional module `15PUZZLE-BOARD` above. The rules `left`, `right`, `down`, and `up` displace the tile at each side of the blank towards it. In other words, they move the blank towards such direction. Using these operations, the goal of the game is to arrive at the configuration of the first board of Fig. 1 from any shuffled board like the second board.

```
mod 15PUZZLE is
  protecting 15PUZZLE-BOARD .

  var T : Tile . vars LU RU LD RD : Row . var P : Puzzle .

   rl [left]  : T b => b T .
   rl [right] : b T => T b .
  crl [down]  : (LU b RU) ; (LD T RD)
             => (LU T RU) ; (LD b RD) if size(LU) = size(LD) .
  crl [up]    : (LU T RU) ; (LD b RD)
             => (LU b RU) ; (LD T RD) if size(LU) = size(LD) .
endm
```

Rules may be named by means of an optional *label* between brackets. This will be very helpful for the specification of strategies in the strategy language. Conditions for rules are equational conditions plus possible rewriting condition fragments `l => r`, which may contain free variables in its righthand side `r` to be instantiated by matching.

The Maude interpreter provides various commands to execute its programs. The `reduce` command simplifies a given term to its normal form with the equations and memberships E modulo the structural axioms B.

```
Maude> reduce size(1 b 2 3) .
rewrites: 14
result NzNat: 4
```

The `rewrite` and `frewrite` commands [15, §5.4] rewrite a term using all the rewriting rules in the module (as well as the equations and memberships E, all applied modulo the axioms B, as explained above). An optional bound on the number of rewriting steps can be given between brackets.

```
Maude> rewrite [21] 1 b 2 3 .
rewrites: 21
result Row: b 1 2 3
```

Moreover, the `search` command lets the user find all terms reachable by rewriting that match a pattern and satisfy a specified condition. The rewriting paths that lead to the found terms can also be inspected.

```
Maude> search 1 b 2 3 =>* 1 2 R .

Solution 1 (state 2)
states: 3  rewrites: 2
R:Row --> b 3

Solution 2 (state 3)
states: 4  rewrites: 5
R:Row --> 3 b

No more solutions.
states: 4  rewrites: 6
```

Further details about the language and the interpreter can be found in the Maude manual [15].

2.5. Strategies

In a broad sense, strategies are seen as recipes to control a process in which multiple decisions are possible. Several formal definitions of strategy have been given for reduction and rewriting systems [10,28,58]. In the λ-calculus [7], a strategy is usually formalized by a function from λ-terms to λ-terms, whose output can be derived from the input by some reductions. They typically apply a rule in a position determined by a fixed criterion, like a β-reduction on the leftmost-innermost redex. More general notions of strategy let the selection of the next rewrite be non-deterministic and may depend on the derivation history [56].

Considering rule labels, a rewriting system can be seen as a labeled graph (T_Σ, A) whose states are terms and its transitions $(t, l, t') \in A$ are rule applications $t \to_R t'$ with a rule of label l. Thus, an execution is a path within the graph, either finite or infinite. In this context, we can see strategies from different points of view:

- Seeing them as a choice of allowed executions, an *abstract strategy* is formally defined as a subset $\mathcal{S}$ of the paths in the graph.
- Focusing on the results of these computations, strategies are seen as functions from an initial term to the set of its results

$$\mathcal{S}(t) := \left\{ t' \in T_\Sigma \text{ s.t. there is a path } t \xrightarrow{l_1}_R t_1 \xrightarrow{l_2}_R \cdots \xrightarrow{l_n}_R t' \text{ in } \mathcal{S} \right\}$$

  Notice that abstract strategies may also select infinite paths, which are useful in the specification of reactive systems, as described in [47,52,50].
- Looking at the allowed next steps, *intensional strategies* are defined as partial functions that map an execution in progress to the admitted next steps. This formalization is often used in games and concurrent agent systems.
- Regarding syntax, we can contemplate strategies as expressions in some strategy specification language, whose semantics can be given in any of the more abstract forms above.

Although the expressive power of abstract strategies is greater than that of intensional ones [10], and sets of results lack information about the intermediate states, all these descriptions may be useful in different situations. In practice, our strategies will be specified explicitly with a specific syntax: the Maude strategy language that we describe in this paper.

3. The Maude strategy language

The design of the Maude strategy language is highly influenced by the traditional reflective specification of strategies as metalevel programs in Maude [16,14, §14], while avoiding their conceptual difficulties and verbosity, and by other strategy languages like ELAN [8], Stratego [12], and Tom [6], as already discussed in Section 1. In this section, we describe the syntax and informal semantics of the language, step by step and with examples.

The basic instruction of the Maude strategy language is the selective rule application, and other operators are available to combine them and build more complex strategic programs. These *combinators* include the typical programming constructs (sequential composition, tests, conditions, loops, recursive definitions), nondeterministic choices, and others that specifically exploit the structure of terms. For executing a strategy expression α on a term t, the Maude interpreter provides a command `srewrite` *t* `using` α, thoroughly described in Section 3.9. At the moment, it is enough to know that it enumerates all terms that can be obtained by rewriting with the strategy from the given term, each called a *result* or *solution* of the strategy. The term to which the strategy is applied will be called the *subject term*. Since strategies may be non-deterministic, these results can be zero, one, or even an infinite number. Looking at the results as a whole, strategies can be seen as transformations from a term to a set of terms. *Strategy modules*, the modules where strategies can be named and recursive strategies can be defined, are described in Section 3.10. Formal semantics of what is here informally described can be found in Section 5.

To illustrate each strategy constructor at work, let us go back to the example specified in the modules `15PUZZLE-BOARD` and `15PUZZLE` of Section 2.4. The data representation of the square is not intended to be efficient, as shown by the conditional rules for `down` and `up`, but just to be easily readable and illustrative for the strategy examples in this section. The game will be discussed in more detail in Section 4.2.

3.1. Idle and fail

The simplest strategies are the constants **`idle`** and **`fail`**.

⟨*Strat*⟩ ::= `idle` | `fail`

The **`fail`** strategy always *fails* for any given state, i.e., it does not produce any result.

```
Maude> srewrite 1 b 2 using fail .

No solution.
rewrites: 0
```

In a broader sense, we say that a strategy α *fails* in a state t if it does not produce any result. On the contrary, the **`idle`** constant always succeeds, and returns the given state unaltered as its only result.

```
Maude> srewrite 1 b 2 using idle .

Solution 1
rewrites: 0
result Row: 1 b 2
```

```
No more solutions.
rewrites: 0
```

These strategy constants are useful in combination with the conditional operator (see Section 3.5), the rule application operator with rewriting conditions (see Section 3.2), and strategy definitions (see Sections 3.11, 4.2 and 4.3 for examples where `idle` is used to fill the base case of an inductive strategy).

3.2. Rule application

Since strategies are used to control rewriting, the cornerstone of the language is the application of rules. Rules are usually selected by their labels, but a finer control on how they are applied is possible using an initial ground substitution, which can optionally be provided between brackets. Moreover, in order to apply a conditional rule with rewriting conditions, strategies must be provided to control them.

⟨*Strat*⟩ ::= ⟨*RuleApp*⟩
 | `top(` ⟨*RuleApp*⟩ `)`

⟨*RuleApp*⟩ ::= ⟨*Label*⟩ `[` `[` ⟨*Substitution*⟩ `]` `]` `[` `{` ⟨*StratList*⟩ `}` `]`
 | `all`

⟨*Substitution*⟩ ::= ⟨*Variable*⟩ `<-` ⟨*Term*⟩
 | ⟨*Substitution*⟩ `,` ⟨*Substitution*⟩

⟨*StratList*⟩ ::= ⟨*Strat*⟩
 | ⟨*Strat*⟩ `,` ⟨*StratList*⟩

The strategy expression $label[x_1 <- t_1, \ldots, x_n <- t_n]\{\alpha_1, \ldots, \alpha_m\}$ denotes the application anywhere within the subject term of any rule labeled *label* in the current module having exactly m rewriting condition fragments.

```
Maude> srewrite 1 b 2 ; 3 b 4 using right .

Solution 1
rewrites: 1
result Puzzle: 1 2 b ; 3 b 4

Solution 2
rewrites: 2
result Puzzle: 1 b 2 ; 3 4 b

No more solutions
rewrites: 2
```

Moreover, the variables in both rule sides and in its condition are previously instantiated by the substitution that maps x_i to t_i for all $1 \leq i \leq n$.

```
Maude> srewrite 1 b 2 ; 3 b 4 using left[T <- 1] .

Solution 1
rewrites: 1
result Puzzle: b 1 2 ; 3 b 4

No more solutions
rewrites: 1
```

When strategies are specified within curly brackets, each α_i controls the rewriting of the i-th rewriting condition fragment l_i `=>` r_i of the selected rule. As usual when evaluating rule conditions, both sides l_i and r_i are instantiated with the substitution determined by the matching of the lefthand side and the previous condition fragments. However, the instance of l_i is rewritten according to the strategy α_i, and only its solutions are matched against r_i. As usual again, the evaluation of the next rewriting condition fragments continues with the partial substitution obtained by each result of the match modulo B, yielding potentially different one-step rewrites. We now present a simple module illustrating the application of rules with rewriting fragments. The `PuzzleLog` pair maintains a history of applied moves in addition to the puzzle being solved, and both are updated simultaneously with the `move` rule. This rule is marked with the `nonexec`(utable) attribute, which causes the Maude rewrite commands to ignore it, because it contains a free variable `M` in the right-hand side. However, it can be executed with a rule application strategy that instantiates such a variable.

```
mod 15PUZZLE-LOG is
  protecting 15PUZZLE .
  protecting LIST{Qid} .

  sort PuzzleLog .
  op <_|_> : List{Qid} Puzzle -> PuzzleLog [ctor] .

  var M : Qid . var L : List{Qid} . vars P P' : Puzzle .

  crl [move] : < L | P > => < L M | P' > if P => P' [nonexec] .
endm
```

The imported module `LIST{Qid}` is an instance of the parameterized module `LIST` (described in Section 3.11) with quoted identifiers as elements. These identifiers are arbitrary words prefixed by a single quote, like `'left`, and they are appended to the list by juxtaposition.

```
Maude> srewrite < nil | 1 b 2 > using move[M <- 'left]{left} .

Solution 1
rewrites: 3
result PuzzleLog: < 'left | b 1 2 >

No more solutions.
rewrites: 3
```

A special strategy for rule application is the constant `all` that executes any rule in the module, labeled or not. Rewriting conditions are evaluated without restrictions, as in the usual `rewrite` command. However, rules marked with `nonexec` and the implicit rules that handle external objects [15, §9] are excluded.

```
Maude> srewrite < nil | 1 b 2 > using all .

Solution 1
rewrites: 3
result PuzzleLog: < nil | b 1 2 >

Solution 2
rewrites: 4
result PuzzleLog: < nil | 1 2 b >

No more solutions.
rewrites: 4
```

Rules are usually applied anywhere in the subject term by default; but rule application expressions can be prefixed by the `top` modifier to restrict matching to the top of the subject term. In combination with the subterm rewriting operator, to be explained in Section 3.6, this allows restricting rewriting to specific positions within the term structure. Nevertheless, `top(`α`)` is not necessarily deterministic, because matching of the rule lefthand side takes place modulo structural axioms B such as associativity and commutativity. Additionally, rules can contain matching and rewriting condition fragments that may produce multiple substitutions. For example, if we had a rule `multimv` as `LU b RU => LU RU b` that jumps multiple positions at once, we would obtain:

```
Maude> srewrite 1 b 2 b 3 using top(multimv) .

Solution 1
rewrites: 1
result Row: 1 2 b 3 b

Solution 2
rewrites: 2
result Row: 1 b 2 3 b

No more solutions.
rewrites: 2
```

3.3. Tests

Strategies often need to check some property of the subject term to decide whether to continue rewriting it or not, or whether it can be given as a solution. Test can be used to do these checks and abandon those rewriting paths in which they are not satisfied. Since matching is one of the most basic features of rewriting, tests are based on matching and on evaluation of equational conditions.

⟨*EqCondition*⟩ ::= ⟨*BoolTerm*⟩
| ⟨*Term*⟩ = ⟨*Term*⟩
| ⟨*Term*⟩ := ⟨*Term*⟩
| ⟨*EqCondition*⟩ /\ ⟨*EqCondition*⟩

⟨*Test*⟩ ::= `amatch` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]
| `match` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]
| `xmatch` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩]

Three variants of tests are available regarding where matching takes place: **`match`** tries to match at the top of the subject term, **`amatch`** matches anywhere, and **`xmatch`** matches only at the top, but *with extension* modulo the structural axioms of the top symbol.[2] Their behavior is equivalent to **`idle`** when the test passes, and to **`fail`** when it does not. In other words, the test strategy applied to a term t will evaluate to the set of results $\{t\}$ if the matching and condition evaluation succeeds, and to $\emptyset$ if they fail.

```
Maude> srewrite 1 b 2 using xmatch b N s.t. N =/= 1 .

Solution 1
rewrites: 1
result Row: 1 b 2

No more solutions.
rewrites: 1
```

If the pattern and the condition contain variables that are bound in the current scope, they will be instantiated before matching. The condition is then evaluated like an equational condition [15, §4.3].

3.4. Regular expressions

The first strategy combinators to describe execution paths are the typical regular expression constructors.

⟨*Strat*⟩ ::= ⟨*Strat*⟩ `;` ⟨*Strat*⟩
| ⟨*Strat*⟩ `|` ⟨*Strat*⟩
| ⟨*Strat*⟩ `*`
| ⟨*Strat*⟩ `+`

The *sequential composition* or *concatenation* α `;` β rewrites the subject term with α and then with β. Hence, every result of α `;` β is a result of applying β to a term obtained from α. Of course, if α fails, no result will be obtained regardless of β. The *nondeterministic choice* or *alternation* combinator α `|` β rewrites the subject term using either α or β, chosen nondeterministically. The solutions of the alternation is the union of the solutions of both strategies.

```
Maude> srewrite 1 2 ; 3 b using left ; up | up ; left .

Solution 1
rewrites: 24
result Puzzle: b 2 ; 1 3

Solution 2
rewrites: 32
result Puzzle: b 1 ; 3 2

No more solutions.
rewrites: 32
```

Notice that the previous strategy is parenthesized as `(left ; up) | (up ; left)`, with the usual precedence of regular expressions. The *iteration* α `*` executes α zero or more times consecutively. It can be described recursively as α `*` $\equiv$ **`idle`** `|` α `;` α `*`.

[2] Suppose $+$ is an associative and commutative (AC) symbol, f is a unary symbol, and a, b, and c are constants. Consider the rule $f(x)+a \Rightarrow f(x)+b$. This rule cannot be applied at the top to the term $(f(x)+c)+a$. However, it can be applied *with extension* modulo AC if we add the AC-extension rule $(f(x)+a)+y \Rightarrow (f(x)+b)+y$. See [16, §4.8] for a detailed explanation with examples of extension rules.

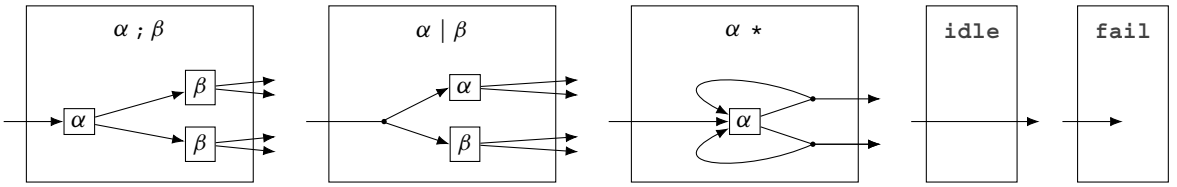


Fig. 2. Schema for the operators of the regular expressions sublanguage.

```
Maude> srewrite 1 b 2 3 using right * .

Solution 1
rewrites: 0
result Row: 1 b 2 3

Solution 2
rewrites: 1
result Row: 1 2 b 3

Solution 3
rewrites: 2
result Row: 1 2 3 b

No more solutions.
rewrites: 2
```

Iterations are the simplest form of a loop in the strategy language, although they finish nondeterministically after each iteration. The typical loop `while` P `do` α `end` where P is a predicate on the subject term can be encoded as `(match` X `s.t.` $P(X)$ `;` α`) *; match` X `s.t. not` $P(X)$ for an appropriate variable X. They can also be useful to expand the transitive closure of a strategy, like in the example above, where we obtain all states reachable by successive applications of the `right` rule.

The already seen **`idle`** and **`fail`** constants complete the regular expressions sublanguage. The usual notation for the non-empty iteration α `+` is also available in the language as a derived combinator α `+` $\equiv$ α `;` α `*`. Fig. 2 summarizes in a diagram the meaning of the main operators presented in this section.

3.5. Conditional and derived operators

Conditional statements and expressions are an elementary control mechanism in any programming language. The next strategy combinator is the typical *if-then-else* construct. However, its condition is a strategy too, which is considered satisfied if it provides at least one result. This idea has been borrowed from Stratego [12] and ELAN [8].

⟨*Strat*⟩ ::= ⟨*Strat*⟩ `?` ⟨*Strat*⟩ `:` ⟨*Strat*⟩

More precisely, the evaluation of the expression α `?` β `:` γ starts by evaluating the condition α. If α succeeds, its results are continued with the positive branch β just as if $\alpha;\beta$ were run. But if α fails, the negative branch γ is evaluated instead on the initial term. In case α is just a test, the conditional behaves like a typical Boolean conditional.

```
Maude> srewrite 1 b 2 3 4 using right * ; (right ? fail : idle) .

Solution 1
rewrites: 6
result Row: 1 2 3 4 b

No more solutions.
rewrites: 6
```

Using the conditional strategy, multiple useful derived combinators can be defined and are available in the language:

⟨*Strat*⟩ ::= ⟨*Strat*⟩ `or-else` ⟨*Strat*⟩
| `not(` ⟨*Strat*⟩ `)`
| ⟨*Strat*⟩ `!`
| `try(` ⟨*Strat*⟩ `)`
| `test(` ⟨*Strat*⟩ `)`

The combinator α `or-else` β evaluates to the result of α unless α fails; in that case, it evaluates to the result of β. Consequently, it is equivalent to

$$\alpha \texttt{ or-else } \beta \equiv \alpha \texttt{ ? idle : } \beta$$

This is one of the most used combinators of the strategy language because rule precedences appear in many situations. The negation combinator `not`(α) is defined as

$$\texttt{not}(\alpha) \equiv \alpha \texttt{ ? fail : idle},$$

thus reversing the binary result of evaluating α. More precisely, the initial term is obtained when α fails, and `not` fails when α succeeds. The *normalization* operator α `!` evaluates a strategy repeatedly until just before it fails.

$$\alpha \texttt{ ! } \equiv \alpha \texttt{ * ; not}(\alpha)$$

Observe that the strategy in the `srewrite` example above can now be simply written as `right !`. The strategy `try`(α) works as if α were evaluated, but when α fails it results in $\{t\}$ instead, where t was the initial subject term.

$$\texttt{try}(\alpha) \equiv \alpha \texttt{ ? idle : idle}$$

Finally, `test`(α) checks the success/failure result of α, but it does not change the subject term.

$$\texttt{test}(\alpha) \equiv \texttt{not}(\alpha) \texttt{ ? fail : idle}$$

Notice that this is the same as `not(not`(α)`)`.

3.6. Rewriting of subterms

As seen in Section 3.2, rules are applied anywhere within the subject term by default, but we may be interested in concentrating their application into selected subterms. The following family of combinators allows rewriting multiple selected subterms using some given strategies. Additionally, it can be used to extract information from the subject term and use it for the control of the strategy execution.

⟨*Strat*⟩ ::= `amatchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩
 | `matchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩
 | `xmatchrew` ⟨*Pattern*⟩ [`s.t.` ⟨*EqCondition*⟩] `by` ⟨*VarStratList*⟩

⟨*VarStratList*⟩ ::= ⟨*VarId*⟩ `using` ⟨*Strat*⟩
 | ⟨*VarStratList*⟩ `,` ⟨*VarStratList*⟩

The subterms to be rewritten are selected by matching against a pattern P. Some of the variables in the pattern $x_1, \ldots, x_n$ will match selected subterms $u_1, \ldots, u_n$ to which the strategies $\alpha_1, \ldots, \alpha_n$ are respectively applied.

$$\texttt{matchrew } P[x_1, \ldots, x_n, x_{n+1}, \ldots, x_m] \texttt{ s.t. } C \texttt{ by } x_1 \texttt{ using } \alpha_1 \texttt{, ..., } x_n \texttt{ using } \alpha_n$$

These variables must be distinct, but they may appear more than once in the pattern. There may be additional variables $x_{n+1}, \ldots, x_m$ that do not designate subterms to be rewritten.

The strategy explores all possible matches of the pattern P for the subject term that satisfy the condition C. For each, it extracts the subterms u_i to u_n that match the variables x_1 to x_n, and rewrites them separately in parallel using their corresponding strategies α_1 to α_n. Every combination of their results is reassembled in the original term in place of the original subterms. Like for the tests, three variants can be selected by changing the initial keyword: **`matchrew`** for matching on top, **`xmatchrew`** for doing so with extension, and **`amatchrew`** for matching anywhere. The behavior of the **`amatchrew`** operator is illustrated in Fig. 3.

```
Maude> srewrite 1 b 2 ; 3 b 4 using matchrew RU ; RD
                                  by RU using left, RD using right .

Solution 1
rewrites: 2
result Puzzle: b 1 2 ; 3 4 b

No more solutions.
rewrites: 2
```

The following simple example shows that patterns may be non-linear and that multiple matches are possible.

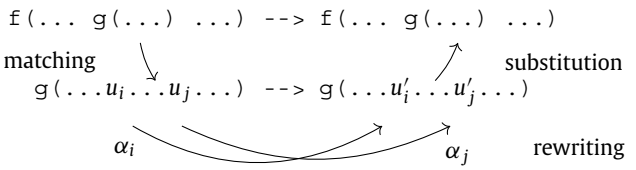


Fig. 3. Behavior of the `amatchrew` combinator.

```
Maude> srewrite 1 b ; 1 b ; 2 b ; 2 b using xmatchrew R ; R by R using left .

Solution 1
rewrites: 1
result Puzzle: b 1 ; b 1 ; 2 b ; 2 b

Solution 2
rewrites: 2
result Puzzle: 1 b ; 1 b ; b 2 ; b 2

No more solutions.
rewrites: 2
```

Variables bound in the outer scope are instantiated in the pattern P and condition C, like in tests. Moreover, the variable scope of the substrategies $\alpha_1, \ldots, \alpha_n$ is extended with the variables in P and C. Notice that **`matchrew`** and its variants are the only strategies that define static variable scopes, and the only ones that bind variables along with strategy calls. In fact, **`matchrew`** operators are often used for binding variables and introduce their values in the strategy control flow, as shown in almost all examples in Section 4, even with patterns as simple as a single variable. A common idiom is **`matchrew`** `S` **`s.t.`** P `:=` `S` **`by`** `S` **`using`** α where α needs to be applied to the whole term, but depends on information that can be obtained by pattern matching from some part of it.[3]

3.7. Pruning of alternative solutions

The strategy language combinators are nondeterministic in different ways: rule applications may produce multiple rewrites, the alternation operator $\alpha \,|\, \beta$ could choose α or β, the iteration α `*` may execute α any number of times, and **`matchrew`** could start with different matches. The default behavior of the strategy rewriting engine is to perform an exhaustive search on all the allowed rewriting paths, and show all reachable results. However, the user may be interested in picking any solution without distinction, or may know that a single solution exists. In these cases and for efficiency, the `one` combinator is available.

⟨*Strat*⟩ ::= `one(` ⟨*Strat*⟩ `)`

`one``(`α`)` evaluates α on the given term. If α fails, so **`one`**`(`α`)` does. Otherwise, a solution for α is chosen nondeterministically as its single result. However, since the purpose of introducing this strategy is efficiency, the chosen solution will actually be the first that is found.

```
Maude> srewrite 1 b 2 3 using one(right +) .

Solution 1
rewrites: 1
result Row: 1 2 b 3

No more solutions.
rewrites: 1
```

Notice that all deterministic choices will still be explored until a solution is found. This means that **`one`** will be more effective the fewer steps are required to execute its argument. Another consequence is that **`one`** cannot be used to limit the search of a **`matchrew`** to a single match, which may be desired in some situations, and multiple matches may be tried until a solution of the whole **`matchrew`** is encountered.

More meaningful usage examples of `one` can be found in Section 4.3. ELAN [8] counts with similar constructs `dc one` and `first one`, whose behaviors coincide with our `one` when a single strategy argument is provided (see Section 7).

[3] In a draft version of the language, arbitrary subpatterns could be used to select the subterm to be rewritten, like in **`matchrew`** `f(X, g(Y))` **`by`** `g(Y)` **`using`** α. However, this option was removed for simplicity and to avoid some pathological cases with overlapping patterns.

3.8. Strategy calls

Instead of writing huge monolithic strategy expressions and copy them verbatim to be executed, complex strategies are better split into several subexpressions referred to by meaningful names. These strategies are defined in strategy modules, which will be explained in Section 3.10, but also extend the language of expressions with a *strategy call* constructor.

⟨*Strat*⟩ ::= ⟨*StratCall*⟩

⟨*StratCall*⟩ ::= ⟨*StratId*⟩ (⟨*TermList*⟩)
| ⟨*StratId*⟩

Strategies are invoked by a strategy label, and may receive input arguments of any sort in the module, according to their declarations. These parameters are written in a comma-separated list between parentheses. For strategies without parameters, the parentheses can be omitted, except when there is a rule with the same label in the module.

For example, suppose we have defined a strategy `move` that moves the blank as many horizontal and vertical positions as indicated by its two arguments (this strategy will actually be defined in Section 3.10). The following command executes it with an offset of two horizontal positions:

```
Maude> srewrite 1 b 2 3 4 using move(2, 0) .

Solution 1
rewrites: 2
result Puzzle: 1 2 3 b 4

No more solutions.
rewrites: 2
```

3.9. Strategy commands

Strategies are evaluated in the interpreter using the `srewrite` and `dsrewrite` commands, which look for solutions of the strategy and show all those they find.

⟨*Commands*⟩ ::= `srewrite` [[⟨*Nat*⟩]] [`in` ⟨*ModId*⟩] ⟨*Term*⟩ `using` ⟨*Strat*⟩ .
| `dsrewrite` [[⟨*Nat*⟩]] [`in` ⟨*ModId*⟩] ⟨*Term*⟩ `using` ⟨*Strat*⟩ .

Following the usual convention of Maude commands, an optional bound on the number of solutions to be calculated can be specified between brackets after the command keyword. Nevertheless, more solutions can be requested afterwards using the `continue` command. By default, the command will be executed in the current Maude module, but a different module could be specified preceded by `in`. The `srewrite` keyword can be abbreviated to `srew`, and `dsrewrite` to `dsrew`.

The `srewrite` command explores the rewriting tree using a *fair* policy that ensures that all solutions are eventually found if there is enough memory. Not being completely a breadth-first search, it may explore multiple execution paths in parallel. More details can be found in Section 6.

```
Maude> srew [2] b 1 2 ; b 3 using right + .

Solution 1
rewrites: 2
result Puzzle: 1 b 2 ; b 3

Solution 2
rewrites: 3
result Puzzle: b 1 2 ; 3 b
```

On the contrary, the `dsrewrite` command performs a depth-first exploration of the tree. It is usually faster and uses less memory, but some solutions may not be reached because of nonterminating execution branches.

```
Maude> dsrew [2] b 1 2 ; b 3 using right + .

Solution 1
rewrites: 1
result Puzzle: 1 b 2 ; b 3

Solution 2
```

```
rewrites: 2
result Puzzle: 1 2 b ; 3 b
```

Notice that the order in which solutions are obtained may differ depending on the type of search, as they do in the execution above. The displayed rewrite count reflects all the equational and rule rewrites that have been applied until the solution was found, but its origin could be in other execution branches not yet completed, or abandoned because they do not lead to a solution.

The search conducted by the `srewrite` and `dsrewrite` commands theoretically explores the subtree of the rewriting tree pruned by the restrictions of the strategy, but their search space is actually a graph. The execution engine is able to detect already visited execution states, thus preventing the redundant evaluation of the same strategy on the same term. Consequently, the strategy evaluation may finish in situations where, operationally, nonterminating executions are involved.

```
Maude> srew 1 b using (left | right) * .

Solution 1
result Row: 1 b

Solution 2
result Row: b 1

No more solutions.
```

However, strategy evaluation is not always terminating, since the underlying rewriting system may have infinitely many states. In addition, the cycle detector does not operate for strategy calls, unless they are tail recursive and do not have parameters.

3.10. Strategy modules and recursion

As anticipated in Section 3.8, callable strategies can be declared and defined in strategy modules. Apart from the benefits already described, this increases the expressiveness of the strategy language via recursive and mutually recursive strategies, which can also keep a control state in their parameters.

Strategy modules are a third level of Maude modules, devoted to represent the control of rewriting systems by means of the strategy language, as the classical functional and system modules were dedicated to represent equational and rewrite theories, respectively. They are introduced by the **`smod`** keyword and closed by **`endsm`**.

⟨*Module*⟩ ::= `smod` ⟨*ModId*⟩ [⟨*ParameterList*⟩] `is` ⟨*SmodElt*⟩* `endsm`

⟨*SmodElt*⟩ ::= ⟨*ModElt*⟩ | ⟨*StratDecl*⟩ | ⟨*StratDef*⟩

Strategy modules are extensions of system modules in a similar way as system modules are extensions of functional modules. Therefore, they may include any declaration or statement that is allowed in these lower-level modules. However, to promote a clean separation between the rewriting theory specification and its control, we encourage including only strategy-related statements in strategy modules, apart from importations and variable declarations. Only strategy modules are able to import other strategy modules, but they can import modules of any kind using the usual statements: `including`, `extending`, `generated-by`, and `protecting`. The semantic difference between these importation modes is described in [15, §10.2.1].

The strategy-related statements are strategy declarations and strategy definitions.

⟨*StratDecl*⟩ ::= `strat` ⟨*SLabel*⟩+ [`:` ⟨*Type*⟩*] `@` ⟨*Type*⟩ `.`

⟨*StratDef*⟩ ::= `sd` ⟨*StratCall*⟩ `:=` ⟨*Strat*⟩ `.`
| `csd` ⟨*StratCall*⟩ `:=` ⟨*Strat*⟩ `if` ⟨*Condition*⟩ `.`

The following line declares a strategy *slabel* that receives n parameters of sorts $s_1, \ldots, s_n$, and that is intended to control rewriting of terms of sort s.

`strat` *slabel* `:` s_1 `...` s_n `@` s `.`

This latter sort s is understood as a mere comment and ignored by Maude. Many strategies with a common signature can be defined in the same declaration, by writing multiple identifiers, in which case the plural keyword **`strats`** is preferred. The input parameter sorts and the colon are omitted if the strategy has no parameters.

Strategies are defined by means of conditional or unconditional strategy definitions.

```
sd slabel(p1, ..., pn) := α .
csd slabel(p1, ..., pn) := α if C .
```

These definitions associate the strategy name *slabel* to an expression α whenever the input parameters match the patterns $p_1, \ldots, p_n$. The syntax of conditions is the same as that of equations and tests, explained in Section 3.3. The lefthand sides of the definitions are strategy calls as described in Section 3.8. Variables in the pattern and the condition may appear in the strategy expression α. When a strategy is called, all strategy definitions that match the input arguments will be executed, and the union of all their results is the result of the call. Hence, strategies without any definition at all behave like `fail`.

The following is an example of strategy module that imports the system module `15PUZZLE`, declares two strategies, `loop` and `move`, and gives definitions for them.[4]

```
smod 15PUZZLE-STRATS is
  protecting 15PUZZLE .
  protecting INT .

  strat loop           @ Puzzle .
  strat move : Int Int @ Puzzle .

  var N : Nat . var M : Int .

  sd loop := left ; up ; right ; down .

  sd move(0, 0)       := idle .
  sd move(s(N), M)    := right ; move(N, M) .
  sd move(- s(N), M)  := left  ; move(- N, M) .
  sd move(0, s(N))    := down  ; move(0, N) .
  sd move(0, - s(N))  := up    ; move(0, - N) .
endsm
```

While `loop` simply displaces the blank in a fixed loop, `move` moves it some offset in the board. Its five definitions have disjoint patterns, so that only a single definition will be executable for any given input. Therefore, the strategy is deterministic. If `M` were written instead of `0` in the last two definitions, multiple definitions could be activated for the same call, and both vertical and horizontal displacements would be mixed nondeterministically to bridge the distance.

```
Maude> srewrite 1 2 3 ; 4 5 6 ; 7 b 8 using move(1, -2) .

Solution 1
rewrites: 70
result Puzzle: 1 2 b ;
               4 5 3 ;
               7 8 6

No more solutions.
rewrites: 70
```

3.11. Parameterization

Maude's support for *parameterized programming* [25,15, §6.3] in its functional and system modules has also been extended to strategy modules. To describe parameterization for strategy modules, we will first introduce the basic and common building blocks: theories, parameterized modules, and views. *Theories* are used to express the interface of parameterized modules, by describing the requirements that any actual parameter must satisfy. Their syntax is almost identical to that of functional and system modules, but they are delimited by keywords **`fth`** and **`endfth`**, for functional theories, and **`th`** and **`endth`** for system theories. However, their declarations are understood as formal objects, and the executability requirements that a module must obey are not required for theories. The simplest one, although extensively used, is the following theory `TRIV` specifying a single parameter sort:

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

A *parameterized module* includes in its header `PM{X1 :: TH1, ..., Xn :: THn}` a list of one or more formal parameters, each bound to a theory and identified by a name. In its body, the parameter sorts are referred to by their

[4] The `15PUZZLE-STRATS` module only includes unconditional strategy definitions. For an example using conditional definitions, see the `solveLoop` strategy in Section 4.2.

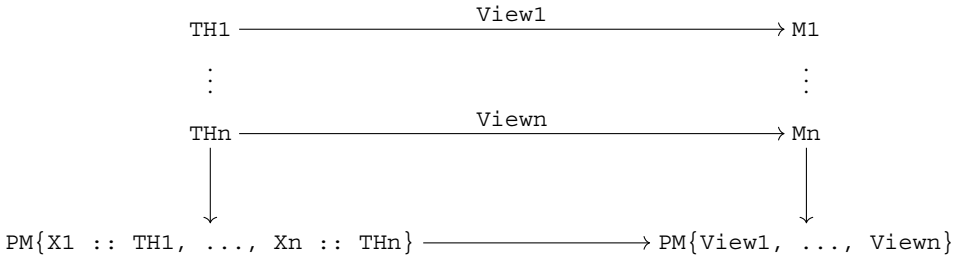


Fig. 4. Diagram of a parameterized module instantiation.

formal names prefixed by the parameter name and a dollar sign `$`, while the formal operators are referred to by their original names. For example, a parameterized module for lists can be specified as follows:

```
fmod LIST{X :: TRIV} is
  sorts NeList{X} List{X} .
  subsort X$Elt < List{X} .

  op nil : -> List{X} .
  op __ : List{X} List{X} -> List{X} [comm assoc id: nil] .

  *** (more declarations and equations)
endfm
```

How a module satisfies the requirements of a theory[5] is specified using a *view*, which maps the formal objects in the theory to the actual objects in the chosen target module where the theory is interpreted. Views are then used to instantiate the parameterized modules as depicted in Fig. 4. For example, the following view interprets `NAT` as a `TRIV` by mapping the formal sort `Elt` to the actual sort `Nat` of natural numbers.

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

Then, the parameterized module `LIST{X :: TRIV}` is instantiated by the view `Nat` to produce the module `LIST{Nat}` of lists of natural numbers. Such module instantiation expressions can appear in importation statements to be used in the importing module.

```
fmod COUNTDOWN is
  protecting LIST{Nat} .
  op countdown : -> List{Nat} .
  eq countdown = 5 4 3 2 1 0 .
endm
```

Strategy modules can be parameterized by functional and system theories too, but also with formal strategies expressed as *strategy theories*.

⟨*Theory*⟩ ::= `sth` ⟨*ModId*⟩ `is` ⟨*SmodElt*⟩* `endth`

Strategy theories are syntactically identical to strategy modules. They should only contain strategy declarations and definitions, but they can also include functional or system theories using the **`including`** mode (or modules using any importation mode). The formal strategies declared in the theory should be realized by the actual target modules. Modules other than strategy modules cannot be parameterized by strategy theories. A simple example of strategy theory is the following `STRIV` that declares a single strategy without arguments operating on the formal sort of the `TRIV` theory.

```
sth STRIV is
  including TRIV .
  strat st @ Elt .
endsth
```

Parameterized by this theory, we can then specify the following module `REPEAT`, which defines a strategy `repeat(`n`)` that applies n times the parameter strategy `st`.

[5] That the requirements of theory T are satisfied by view V is a *proof obligation* that is not checked by Maude. It can instead be discharged with the help of tools in the Maude Formal Environment [21].

```
smod REPEAT{X :: STRIV} is
  protecting NAT .

  strat repeat : Nat @ X$Elt .

  var N : Nat .

  sd repeat(0)    := idle .
  sd repeat(s(N)) := st ; repeat(N) .
endsm
```

Again, views are required to interpret strategy theories and instantiate strategy modules. Thus, the syntax of views is extended to support strategy mappings:

⟨*ViewElt*⟩ ::= `strat` ⟨*StratId*⟩ `to` ⟨*StratId*⟩ .
| `strat` ⟨*StratId*⟩ [: ⟨*Type*⟩ *] @ ⟨*Type*⟩ `to` ⟨*StratId*⟩ .
| `strat` ⟨*StratCall*⟩ `to expr` ⟨*Strat*⟩ .

Three different forms of mapping are offered with the same structure as operator mappings [15, §6.3.2]: using **`strat`** *formalName* : s_1 ... s_n @ s **`to`** *actualName*, the formal strategy with the given name *formalName* and signature is mapped to an actual strategy in the target module whose name is *actualName* and whose input arguments' sorts are the translation of the formal signature according to the sort mappings of the view. To map all overloaded strategies with a given name at once, another mapping **`strat`** *formalName* `to` *actualName* is available. Finally, **`strat`** *slabel*$(x_1, \ldots, x_n)$ **`to expr`** α maps the strategy *slabel* whose input types are those of $x_1, \ldots, x_n$ to the strategy expression α that may depend on these variables. Only variables are allowed as arguments in the lefthand side.

For instance, we can repeatedly apply the `loop` strategy defined in the strategy module `15PUZZLE-STRATS` at the beginning of this section, by instantiating the `REPEAT` module with the following view from `STRIV` to `15PUZZLE-STRATS`.

```
view Loop from STRIV to 15PUZZLE-STRATS is
  sort Elt to Puzzle .
  strat st to loop .
endv
```

`Loop` maps the `Elt` sort to the `Puzzle` sort, and the formal strategy `st` to the `loop` strategy in `15PUZZLE-STRAT`. Instantiating `REPEAT{X :: STRIV}` with the view `Loop` by writing `REPEAT{Loop}`, we can now apply `repeat` for `loop`.

```
Maude> srew 1 2 ; 3 b using repeat(2) .

Solution 1
rewrites: 64
result Puzzle: 3  1 ;
               2  b

No more solutions.
rewrites: 64
```

Typically, the target of a view from a strategy theory is a strategy module but, using the `to expr` mapping, views can be directly specified for system modules. For example, the following view `Right` to the system module `15PUZZLE` maps `st` to the rule application expression `right`.

```
view Right from STRIV to 15PUZZLE is
  sort Elt to Puzzle .
  strat st to expr right .
endv
```

Then, in a module including `REPEAT{Right}`, we can execute:

```
Maude> srew b 1 2 3 4 using repeat(3) .

Solution 1
rewrites: 3
result NeRow: 1 2 3 b 4

No more solutions.
rewrites: 3
```

Other combinations of initial theories and target modules or theories are possible, including views from functional or system theories to strategy modules or theories. These possibilities and their implications are discussed in [15, §6.3.2]. Several other examples of parameterized strategy modules are available in [48].

3.12. Reflecting strategies at the metalevel

As pointed out in the introduction, Maude is a reflective language. This means that functional and system modules (and, as we shall see below, also strategy modules) can be treated as *data* and can then be transformed and manipulated in very powerful *meta-programming*, reflective ways within Maude. For example, virtually all formal verification tools, which of course must inspect, manipulate, and sometimes transform Maude modules as data, are built this way.

In fact, all new features added to Maude are routinely reflected at the metalevel to make Maude metaprogramming even more powerful. For strategies, as for any other Maude feature, what this allows us to do is to make Maude's object language *user-extensible*. This is related to our remark in the introduction that Maude's strategy language design, as any other such design, does not try to support *all* conceivable features, but tries to make some judicious design decisions about what features to directly support. In a non-reflective setting, such language design decisions are somewhat dramatic, since if a desirable feature is not supported we may be out of luck. But this is not so with reflection, since the user can easily extend the given object language with new features implemented in Maude at the metalevel. Furthermore, reflecting strategies at the metalevel is also useful to be able to access the new strategic functionality for meta-programming purposes, and so that interactive applications and specific frameworks built on top of Maude can benefit from those, and potential formal tools can reason about strategies.

The Maude metalevel is a hierarchy of modules specifying the different Maude entities and operations [15, §16]. Terms are metarepresented in the `META-TERM` module as terms of sort `Term`, modules are defined in `META-MODULE` as terms of sort `Module` along with its statements, views are represented in `META-VIEW` as terms of sort `View`, and `META-LEVEL` represents operations like reduction, rule application, rewriting, etc., as *descent functions*. Thus, to reflect the strategy language, we have specified it in a new module `META-STRATEGY`, extended the `META-MODULE` and `META-VIEW` modules with strategy modules and views, and incorporated the strategy-rewriting operations into `META-LEVEL`. First, the strategy language constructs have been defined as follows:

```
sorts RuleApplication CallStrategy Strategy StrategyList .
subsorts RuleApplication CallStrategy < Strategy < StrategyList .

ops fail idle : -> Strategy [ctor] .
op all : -> RuleApplication [ctor] .
op _[_]{_} : Qid Substitution StrategyList -> RuleApplication [ctor ...] .
op top : RuleApplication -> Strategy [ctor] .
op match_s.t._ : Term Condition -> Strategy [ctor ...] .
op _|_ : Strategy Strategy -> Strategy [ctor assoc comm id: fail ...] .
op _;_ : Strategy Strategy -> Strategy [ctor assoc id: idle ...] .
op _or-else_ : Strategy Strategy -> Strategy [ctor assoc ...] .
op _+ : Strategy -> Strategy [ctor] .
op _?_:_ : Strategy Strategy Strategy -> Strategy [ctor ...] .
op matchrew_s.t._by_ : Term Condition UsingPairSet -> Strategy [ctor] .
op _[[_]] : Qid TermList -> CallStrategy [ctor prec 21] .
op one : Strategy -> Strategy [ctor] .
*** and others (see [15, §16.3] or the Maude distribution)
```

Using this metarepresentation of strategy expressions, strategy declarations and definitions are represented as operators in `META-MODULE`:

```
sorts StratDecl StratDefinition .
op strat_:_@_[_]. : Qid TypeList Type AttrSet -> StratDecl [ctor ...] .
op sd_:=_[_].     : CallStrategy Strategy AttrSet -> StratDefinition [ctor ...] .
op csd_:=_if_[_]. : CallStrategy Strategy EqCondition AttrSet
                      -> StratDefinition [ctor ...] .
```

And its `Module` sort has also been extended with new symbols for strategy modules and theories with slots to hold these new module items.

```
op smod_is_sorts_._______endsm : Header ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
  StratDeclSet StratDefSet -> StratModule [ctor ...] .
op sth_is_sorts_._______endsth : Header ImportList SortSet
  SubsortDeclSet OpDeclSet MembAxSet EquationSet RuleSet
  StratDeclSet StratDefSet -> StratTheory [ctor ...] .
```

Similarly, the view symbol of `META-VIEW` has been extended with a new entry for strategy bindings.

Finally, the functionality of the commands `srewrite` and `dsrewrite` is meta-represented in a descent function defined in the `META-LEVEL` module.

```
op metaSrewrite : Module Term Strategy SrewriteOption Nat
                  ~> ResultPair? [special (..)] .

sort SrewriteOption .
ops breadthFirst depthFirst : -> SrewriteOption [ctor] .
```

`metaSrewrite` receives the meta-representations of a module,[6] a term, a strategy, and the search type for the results of rewriting the term according to the strategy in that module. As expected from Section 3.9, `breadthFirst` corresponds to `srewrite` and `depthFirst` to `dsrewrite`. Since this may lead to multiple solutions, the last parameter is used to enumerate them in increasing order until the `failure` constant is returned. For example, `srewrite 0 b 0 using right *` at the metalevel is:

```
Maude> red in META-LEVEL : metaSrewrite(upModule('15PUZZLE, false),
 '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 0) .
rewrites: 2
result ResultPair: {'__['0.Zero,'b.Tile,'0.Zero],'Row}

Maude> red metaSrewrite(upModule('15PUZZLE, false),
 '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 1) .
rewrites: 2
result ResultPair: {'__['0.Zero,'0.Zero,'b.Tile],'Row}

Maude> red metaSrewrite(upModule('15PUZZLE, false),
 '__['0.Zero, 'b.Tile, '0.Zero], ('right[none]{empty}) *, breadthFirst, 2) .
rewrites: 2
result ResultPair: (failure).ResultPair?
```

Using its metarepresentation, the strategy language can be extended in different ways. In [51], a general schema for extending the language at the metalevel is presented and applied to some constructs of other languages that are directly available in Maude, as discussed in Section 7. They are the *congruence operators* from ELAN/Stratego and *generic traversals* from Stratego. These and other extensions are available in the Maude strategy language webpage [22].

4. Examples

The strategy language has already been applied to several examples related to semantics of programming languages [26,11,37], proof systems [3,53,57], the ambient calculus [44], neural networks [55], membrane computing [3,53], games [51], a Sudoku solver [54], etc. In this section we show the features of the strategy language in action with examples from different fields:

1. Two simple introductory examples in Section 4.1. They illustrate several aspects of the language including the **`matchrew`** combinator, strategy modules, recursive strategies, and parameterization.
2. A strategy for the running example of Section 3, the 15-puzzle, that solves the game. This section shows how a rather complex algorithm can be implemented by a strategy in a modular way. The very useful mutually recursive strategies, conditional strategy definitions, and strategy arguments are exemplified here. Moreover, this covers games, one of the most relevant application areas for strategies.
3. The RIP protocol in Section 4.3 demonstrates the application of strategies to specify and simulate realistic examples like a communication protocol, which can later be used for verification [45]. In this example, we also care about the performance of the simulation and make use of the **`one`** combinator for that purpose. The controlled delivery of messages in this object-oriented model can also be translated to other specifications of this kind.
4. Finally, the Knuth-Bendix is a paradigmatic example of the advocated separation of concerns between rules and strategies, because it specified several completion algorithm in separate strategy modules that operate on exactly the same deduction system. Moreover, it is a classical example in field of automated deduction, where strategies are also quite relevant.

[6] `upModule('`*name*`, false)` can be used to obtain the meta-representation of the module *name*, where `false` can be replaced by `true` to obtain a flattened version where importations are resolved. Similarly, `upTerm(`t`)` can be used to obtain the meta-representation of a term t.

4.1. Two simple introductory examples

We start the section with two simple examples proposed in the beginnings of the strategy language [35]: a blackboard game and a generic backtracking scheme. The following system module defines a blackboard as a multiset of natural numbers along with a rule `play` that replaces two of them by their arithmetic mean.

```
mod BLACKBOARD is
  protecting NAT .

  sort Blackboard .
  subsort Nat < Blackboard .

  op __ : Blackboard Blackboard -> Blackboard [assoc comm] .

  vars M N : Nat .

  rl [play] : M N => (M + N) quo 2 .
endm
```

The goal of the game is to obtain the greatest number after reducing to whole blackboard to a single figure, since the order in which means are calculated affects the result, as we can see by running `play` exhaustively.

```
Maude> srew 8 7 4 3 2 1 using play ! .

Solution 1
rewrites: 1407
result NzNat: 6

[...] (the omitted solutions are 5, 4, 3)

Solution 5
rewrites: 13077
result NzNat: 2

No more solutions.
rewrites: 24510
```

Since choosing the next two numbers at random is not suited to win the game, a player may try some strategies: combining first the two greatest numbers, the two lowest, the greatest and the lowest, etc. These strategies are specified in the following strategy module, using the `matchrew` combinator and some auxiliary functions to obtain the required information from the term:

```
smod BLACKBOARD-STRAT is
  protecting BLACKBOARD .

  vars X Y M N : Nat .
  var  B       : Blackboard .

  strats maxmin maxmax minmin @ Blackboard .

  sd maxmin := (matchrew B s.t. X := max(B) /\ Y := min(B)
                by B using play[M <- X , N <- Y] ) ! .
  sd maxmax := (matchrew B s.t. X := max(B)
                             /\ Y := max(remove(X, B))
                by B using play[M <- X , N <- Y] ) ! .
  sd minmin := (matchrew B s.t. X := min(B)
                             /\ Y := min(remove(X, B))
                by B using play[M <- X , N <- Y] ) ! .

  ops max min : Blackboard -> Nat .
  op remove : Nat Blackboard -> Blackboard .

  eq max(N) = N .
  eq max(N B) = if N > max(B) then N else max(B) fi .
  eq min(N) = N .
  eq min(N B) = if N < min(B) then N else min(B) fi .
  eq remove(X, X B) = B .
endsm
```

After executing them, the player can see that `minmin` is the better option and `maxmax` the worst (and then prove it mathematically).

```
Maude> srew 8 7 4 3 2 1 using maxmax .

Solution 1
rewrites: 106
result NzNat: 2

No more solutions.
rewrites: 106

Maude> srew 8 7 4 3 2 1 using minmin .

Solution 1
rewrites: 106
result NzNat: 6

No more solutions.
rewrites: 106

Maude> srew 8 7 4 3 2 1 using maxmin .

Solution 1
rewrites: 117
result NzNat: 3

No more solutions.
rewrites: 117
```

The second example is a generic backtracking algorithm. The specification of the backtracking problem must adhere to the requirements of the strategy theory `BT-ELEMS`.

```
sth BT-ELEMS is
  protecting BOOL .
  sort State .
  op isSolution : State -> Bool .
  strat expand @ State .
endsth
```

It includes a sort `State` for the problem states, a predicate `isSolution` to check whether a state is a solution, and a strategy `expand` to generate the successors of a given state in the search. With these elements the generic algorithm is defined as a strategy `solve` in the parameterized strategy module `BACKTRACKING`.

```
smod BACKTRACKING{X :: BT-ELEMS} is
  var S : X$State .

  strat solve @ X$State .
  sd solve := (match S s.t. isSolution(S)) ? idle
    : (expand ; solve) .
endsm
```

The strategy `solve` recursively applies `expand` to look for a solution and stops when it finds one.

The generic algorithm can be instantiated with as many instances as desired, for example, the labyrinth problem [23], the Hamiltonian cycle problem [48], the m-coloring problem [22], etc. Here we will show the 8-queens problem:

```
mod QUEENS is
  protecting LIST{Nat} .
  protecting SET{Nat} .
  protecting EXT-BOOL .

  op isSolution : List{Nat} -> Bool .

  vars N M Diff : Nat .
  var  L        : List{Nat} .
  var  S        : Set{Nat} .

  eq isSolution(L) = size(L) == 8 .
```

```
  crl [next] : L => L N if N,S := 1, 2, 3, 4, 5, 6, 7, 8 .

  op isValid : List{Nat} Nat -> Bool .
  op isValid : List{Nat} Nat Nat -> Bool .

  eq isValid(L, M) = isValid(L, M, 1) .
  eq isValid(nil, M, Diff) = true .
  eq isValid(L N, M, Diff) = N =/= M
    and-then N =/= M + Diff and-then M =/= N + Diff
    and-then isValid(L, M, Diff + 1) .
endm
```

The states of the 8-queens problem are lists of natural numbers, where a value m in the position n means that there is a queen in the position (n, m) of the board. Such a list is a solution when its size is eight since all the queens have been placed. States are extended by the `next` rule, which appends a new queen to the board. However, not all possible appends are valid, since the new queen must not share a row or a diagonal with a previous one; and this is what the `isValid` predicate checks.[7] Then, the strategy `expand` is defined as follows:

```
smod QUEENS-STRAT is
  protecting QUEENS .

  strat expand @ List{Nat} .

  var L : List{Nat} . var N : Nat .
  sd expand := top(next) ; match L N s.t. isValid(L, N) .
endsm
```

Specifying how the module `QUEENS-STRAT` is an instance of a `BT-ELEM` problem is achieved by defining a view. Since `expand` and `isSolution` in the theory correspond to their homonym elements in the target module, no explicit mapping is required.

```
view QueensBT from BT-ELEMS to QUEENS-STRAT is
  sort State to List{Nat} .
endv
```

Finally, we can instantiate the parameterized `BACKTRACKING` module and execute `solve` to find solutions for the problem.

```
smod BT-QUEENS is
  protecting BACKTRACKING{QueensBT} .
endsm


Maude> dsrew [1] nil using solve .

Solution 1
rewrites: 22104 in 6ms cpu (8ms real) (3340991 rewrites/second)
result NeList{Nat}: 1 5 8 6 3 7 2 4

Maude> srew [1] nil using solve .

Solution 1
rewrites: 404353 in 123ms cpu (122ms real) (3283151 rewrites/second)
result NeList{Nat}: 1 5 8 6 3 7 2 4
```

The depth-first search of the `dsrewrite` command finds the first solution after fewer rewrites and less time than the fair search of `srewrite`, and using less memory. However, with `srewrite` additional solutions can be obtained faster, since they are already calculated.

```
Maude> continue 1 .

Solution 2
rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
result NeList{Nat}: 1 6 8 3 7 4 2 5
```

[7] The `and-then` operator from the `EXT-BOOL` is short-circuit logical conjunction, since the default `and` operator always reduces both arguments.

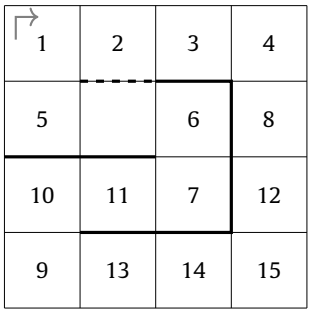


Fig. 5. A circuit in the 15-puzzle.

4.2. The 15-puzzle

In this section we come back to the 15-puzzle introduced in Section 3 and show a strategy to solve it. Remember that the game, which dates from the 1870s, consists of fifteen tiles numbered from 1 to 15 lying on a framed square surface of side length 4. The blank left by the absent sixteenth tile can be used to move the numbers from their positions. Given any arrangement of the puzzle, the goal is to put the tiles as in Fig. 1, in ascending order from left to right and from the top to the bottom, with the blank at the last position. However, only half of the initial settings can be solved, because there exists a relation between the parity of the permutation and the position of the blank, invariant by the allowed moves [61, §8]. The puzzle can be generalized to any side length n with $n^2 - 1$ tiles.

In order to solve the puzzle, a search can be executed with the `search` command or with the strategy commands `srewrite` and `dsrewrite` using the strategy below. However, only puzzles that are near the solution can be expected to be solved like this, since the search space size is impractical for an unguided search, as it has approximately $2.1 \cdot 10^{13}$ states.

```
Maude> search [1] puzzle1 =>* P:Puzzle s.t. P:Puzzle = solved .

Solution 1 (state 571)
states: 572
rewrites: 20713 in 20ms cpu (23ms real) (1035650 rewrites/second)
P:Puzzle --> *** solved

Maude> srew [1] puzzle1 using (left | right | up | down) * ;
                      match P:Puzzle s.t. P:Puzzle = solved .

Solution 1
rewrites: 52906 in 72ms cpu (71ms real) (734805 rewrites/second)
result Puzzle: *** solved
```

where

```
eq puzzle1 =  b   6   2   4  ;          eq solved =   1   2   3   4  ;
              1   5   3   8  ;                        5   6   7   8  ;
              9  10   7  11  ;                        9  10  11  12  ;
             13  14  15  12  .                       13  14  15   b  .
```

In fact, this problem is a classic example used to illustrate search heuristics and algorithms like A*. Using this approach and the strategy-parameterized branch-and-bound implementation described in [48], we have written an instance of this problem that is available in the example collection of the strategy language [22]. Although finding (even the length of) the shortest sequence of moves that lead to the solution is NP-complete [43], non-optimal solutions can be found in polynomial time with deterministic algorithms not based on search. Next, we describe a strategy that solves the puzzle in $\mathcal{O}(n^6)$ moves, where n is the side length, and is inspired by the solution method discussed in [61]. The strategy can be clearly improved in several ways, but we want to keep it as simple as possible.

The key fact is that moving the tiles in a closed circuit through the board does not change the relative position of the numbers within it. Fig. 5 depicts a possible circuit, where the thick lines represent barriers that cannot be crossed. The act of moving the blank through the circuit, say *rotating*, does not alter the sequence of numbers (1, 2, 3, 4, 8, 12, 15, 14, 13, 9, 10, 11, 7, 6, 5) that can be read clockwise from 1 in the figure. However, all tiles are shifted one position in the direction opposite to the movement of the blank. Using successive rotations, we can place any number above or below the dashed line, and change its relative order in the sequence by slipping it across that line. Like a sorting algorithm, every element could be moved to its correct position in the sequence, except that tiles cannot be swapped but can only jump over pairs.

These rotations are described in Maude by the strategies `rotate` and `reverse`, assuming that the blank is initially below the dashed line, which move the blank in the direction indicated by the arrow and in the reverse one respectively. Remember that numbers are moved in the opposite direction.

```
strats rotate reverse godown goup goback @ Puzzle .

sd rotate := left ; up ; right ; right ; right ;
             down ; down ; down ; left ; left ; left ;
             up ; right ; right ; up ; left .
```

Their definitions are explicit concatenations of rules that follow immediately from Fig. 5. Always without crossing any thick line, the strategies `goup` and `godown` move the blank between the positions below and above the dashed line, and `goback` puts it in the lower-right corner, its desired final position. Their definitions are available in [22].

The solving strategy is called `solve`. Its first action is moving the blank below the dashed line with `moveTo`, which uses `move` from Section 3.10. Tile `1` is then placed above the line by successive rotations, and the *sorting loop* in `solveLoop` starts. At the end, the strategy will place `1` above the dashed line again and execute `goback`, which moves the blank to the lower-right corner, leaving `1` and the other tiles in their wanted positions.

```
strat  solve                  @ Puzzle .
strat  move         : Nat Nat @ Puzzle .
strats place solveLoop : Tile @ Puzzle .

sd moveTo(X, Y) := matchrew P by P using
          move(X - blankColumn(P), Y - blankLine(P)) .

sd solve := moveTo(1, 1) ; place(1) ; solveLoop(1) ; place(1) ; goback .

sd place(T) := (match P s.t. T =/= atPos(P, 1, 0) ; rotate) ! .
```

The `solveLoop` strategy iterates on the expected tile sequence, sorting the numbers in the board accordingly. The goal `sequence` of numbers has been defined explicitly for the circuit. The precondition is that the input parameter `T` is above the dashed line and preceded by the correct prefix `LL` in the circuit, and the purpose is to make the next expected tile `NT` occupy the next position. `NT` is found in the board by `findNext` using successive rotations until the expected tile is above the dashed line, as `place` did. Its second parameter counts the distance, which is finally passed on to `move` to displace it back as many times as indicated by this number.

```
vars LL LR : Row . vars T NextT Pen Last : Tile .

csd solveLoop(T) := rotate ; findNext(NextT, 0) ; solveLoop(NextT)
 if LL T NextT LR Pen Last := sequence .

csd solveLoop(T) := idle if LL T Pen Last := sequence .

strat findNext : Tile Nat @ Puzzle .
strat move     : Nat      @ Puzzle .

sd findNext(T, N) := match P s.t. T = atPos(P, 1, 0) ? move(N)
    : (rotate ; findNext(T, s(N))) .
```

The definitions of `solveLoop` are conditional to allow obtaining the next tile from the sequence by matching.

The `move` strategy is defined recursively. In case the distance is greater than two, the tile is moved down across the dashed line with `up`, so that it advances two positions against the rotation direction and towards its expected position. Here, an invariant is that the current tile is above the line and the blank is below. To maintain it for the recursive call, `godown` is called and two reverse rotations make the current tile recover its previous position.

```
sd move(0)       := idle .
sd move(1)       := rotate ; goup ; down ; reverse ; reverse .
sd move(s(s(N))) := up ; godown ; reverse ; reverse ; move(N) .
```

This method does not work when the distance is one, but since the tile that occupies the desired position is misplaced, it can be moved clockwise with a similar operation that makes the tile going up across the dashed line instead. This movement is harmless, because it moves the tile to the unordered portion of the sequence. To see that the moved tile does not surpass the first element of the sequence, observe that the last two elements of the list are omitted in `solveLoop` when the sequence is matched to `LL T NT LR Pen Last`. Obviously, nothing should be done for the last element, but neither should it be done for the penultimate, which can never be swapped with its neighbor as follows from the parity analysis of the puzzle. If the last two elements are misplaced, the problem is not solvable. A summary of the `solve` strategy is depicted in Fig. 6.

Now, we can execute the strategy to solve some puzzles:

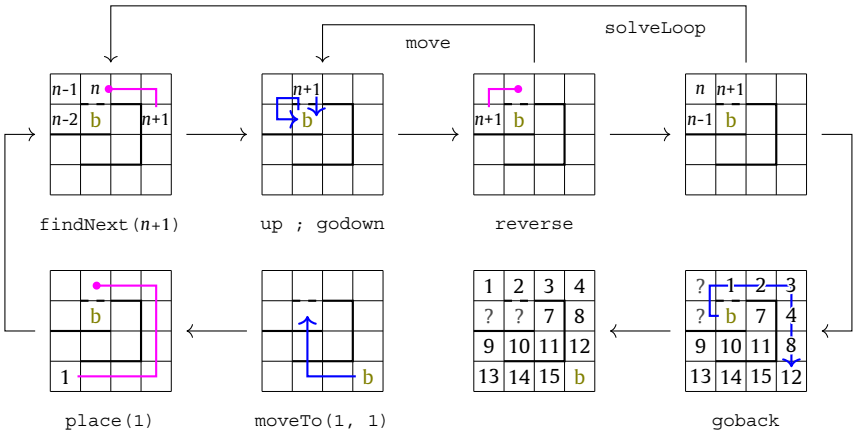


Fig. 6. Overview of the `solve` strategy.

```
Maude> srew (5 1 4 8 ; 2 14 15 3 ; 9 7 6 11 ; 13 10 b 12) using solve .

Solution 1
rewrites: 28868 in 32ms cpu (31ms real) (902937 rewrites/second)
result Puzzle: 1        2       3       4 ;
               5        6       7       8 ;
               9        10      11      12 ;
               13       14      15      b

No more solutions.
rewrites: 28868 in 32ms cpu (31ms real) (902937 rewrites/second)
```

If the starting puzzle is unsolvable, this is revealed by the position of tiles 5 and 6 being swapped.

```
Maude> srew (15 2 1 12 ; 8 5 6 11 ; 4 9 10 7 ; 3 14 13 b) using solve .

Solution 1
rewrites: 40568 in 40ms cpu (41ms real) (1014200 rewrites/second)
result Puzzle: 1        2        3        4 ;
               6        5        7        8 ;
               9        10       11       12 ;
               13       14       15       b

No more solutions.
rewrites: 40568 in 40ms cpu (41ms real) (1014200 rewrites/second)
```

When we introduced this example in Section 2.4, we admitted that the data representation was not intended to be efficient but just easily readable, to illustrate the strategy language at work. However, we can give the board a more efficient representation as a set of position-to-content pairs:

```
op [_,_,_] : Nat Nat Tile -> Puzzle [ctor] .
op __ : Puzzle Puzzle -> Puzzle [ctor assoc comm id: empty] .
```

Like this, the rules `up` and `down` can be implemented by simpler unconditional rules:

```
rl [up]   : [X, Y, T] [X, s(Y), b] => [X, Y, b] [X, s(Y), T] .
rl [down] : [X, Y, b] [X, s(Y), T] => [X, Y, T] [X, s(Y), b] .
```

And since the strategies above are built solely on the rule names and the function `atPos`, the strategies do not need to be modified to work with the new representation, in which the problem is solved using fewer rewrites.

```
Maude> srew [0,0,5] [0,1,2] ... [2,3,b] ... [3,3,12] using solve .

Solution 1
rewrites: 1630 in 20ms cpu (21ms real) (81500 rewrites/second)
result Puzzle: [0,0,1] [0,1,5] [0,2,9]  [0,3,13]
               [1,0,2] [1,1,6] [1,2,10] [1,3,14]
               [2,0,3] [2,1,7] [2,2,11] [2,3,15]
               [3,0,4] [3,1,8] [3,2,12] [3,3,b]
```

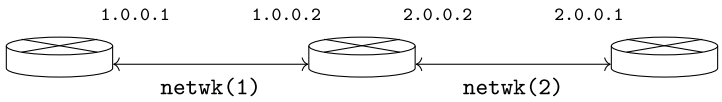


Fig. 7. A simple network topology.

```
No more solutions.
rewrites: 1630 in 20ms cpu (21ms real) (81500 rewrites/second)
```

4.3. The RIP protocol

This section describes a simple application of the strategy language to the specification of a communication protocol. The *Routing Information Protocol* [33] is an interior gateway protocol for the interchange of routing information based on distance vectors.

Internet is composed of different interconnected networks. A message between two hosts in two different networks may travel through a sequence of adjacent networks. The devices that connect them are the *routers*, which decide where to send the data packages so that they arrive to their destinations, preferably by the shortest path. In order to do their job, they need to know the topology of the interconnected networks. This information can be manually established by the system administrator as a table associating a network or arrangement of networks to a physical interface of the router or to the next intermediate router. Such association is called a *routing table*, and this approach, *static routing*. However, routing tables can be constructed dynamically and be aware of network changes, based on data shared by the routers themselves. This is called *dynamic routing*, and RIP is one of its first representative protocols.

RIPv2 routers periodically exchange their routing tables with their neighbors. When using the IP protocol, each table entry locally describes the next step to reach an aggregation of networks given by an IP address and a mask, here represented in *Classless Inter-Domain Routing* notation.[8] The other relevant fields are the IP address of the next-step router in case the network is not directly reachable, the physical interface that must be used to reach the destination, and the distance, which usually measures the number of router jumps and is called *hop count*.

```
op <_,_,_,_,_> : CIDR IPAddr Interface Nat Nat -> Route [ctor] .
```

RIP considers hop-counts above 15 as infinity, so that networks at a greater distance are unreachable. Moreover, the table includes an invalidation timer, to discard entries when no information about them has been received for a significant amount of time.

In this example, the whole network specification is object-oriented [15, §8].[9] The objects are the routers, whose attributes are a routing table, and an interface list that enumerates their IP addresses for each directly connected network.

```
class Router | table : RouteTable,        *** sets of Route terms
               interfaces : Interfaces .  *** explained below

op none : -> Interfaces [ctor] .
op _|>_ : NetworkId CIDR -> Interfaces [ctor prec 31] .
op __ : Interfaces Interfaces -> Interfaces [ctor assoc comm id: none] .
```

Networks are not represented structurally, but by means of unique identifiers of sort `NetworkId`. Since broadcast and multicast messages used by this protocol are distributed inside the boundaries of a network, this information will be relevant. Every IP message contains a sender and a receiver IP address, and a payload. Multicast and broadcast messages additionally include a network identifier.

```
msg IPMessage : IPAddr IPAddr Payload -> IPMsg .  ** see msg as a synonym of op
msg IPMessage : IPAddr NetworkId IPAddr Payload -> IPMsg .
```

In the case of RIP messages, the payload consists of some fields described in the protocol specification, whose main part is the exchanged routing table. The actual definitions of all the previous elements and some operations required to manipulate them are specified in various functional and system modules (`IP-ADDR`, `IP-MESSAGES`, `RIP-ENTRIES`, `ROUTER`, etc. in the example source).

[8] IPv4 addresses are 32-bit words. Networks (or aggregations of them) are collections of IP addresses with a common prefix, whose extension is indicated by the mask. CIDR identifies a network by an address followed by the prefix length.

[9] Maude traditionally supports object-oriented specifications where *objects* are typically written as $< i : c \mid a_1 : v_1, \ldots, a_n : v_n >$ with an identifier i, a class c, and some attributes; they communicate by exchanging *messages* as specified by rewrite rules; and both object and messages live in a soup or multiset called *configuration*. Since Maude 3.3 and previously in Full Maude, syntactic sugar is provided in object-oriented modules (`omods`) to define classes along with their attributes, messages, and to simplify the definition of equations and rules. In particular, attributes can be omitted in these statements when their values do not change.

The basic operation of the RIP protocol is the interchange of routing tables. This can be triggered as a response to an initial request, or by the detection of changes in the network and the routing tables. However, we will focus on the *gratuitous responses* that routers send approximately every 30 seconds to all their interfaces. These messages are multicast to all the routers, and strategies will be used to deliver them once to all the interested receivers. But first, the router actions are described as rules in a system module `RIP`. For example, the emission of a (gratuitous) response is specified by the following rule, where the message address 224.0.0.9 is a multicast address recognized by all RIPv2 routers.

```
rl [response] :
   < I : Router | table: RT, interfaces: NId |> A / Mask Ifs >
=> < I : Router | >
   IPMessage(224 . 0 . 0 . 9, NId, A, ripMsg(2, 2, export(RT))) .
```

This simple rule will not be admissible in a specification without strategies, because it can be endlessly applied. Its counterpart is the rule in charge of processing the RIP message and updating the routing tables accordingly:

```
crl [readResponse] :
   < I : Router | table: RT, interfaces: NId |> A / Mask Ifs >
   IPMessage(224 . 0 . 0 . 9, NId, O, ripMsg(2, 2, RE))
=> < I : Router | table: import(O, NId, RE, RT) >
   IPMessage(224 . 0 . 0 . 9, NId, O, ripMsg(2, 2, RE))
if O =/= A .
```

Notice that the message is not removed from the configuration, since it is a multicast message. The strategy will remove it when it has been received by all the routers, using the `remove-message` rule. The passage of time is performed by the rule `update-timer` that increments the internal timers of the routers.

Over these rules, the protocol operation on a time window of 30 seconds is specified in the `iteration` strategy. This strategy has two other overloaded versions, in which it delegates more specific tasks.

```
strat iteration                  @ Configuration .
strat iteration : Set{Oid}       @ Configuration .
strat iteration : Oid Interfaces @ Configuration .

var C : Configuration .    var I : Oid .

sd iteration := matchrew C by C using (
                    one(update-tick(allOids(C))) ;
                    iteration(allOids(C))
                ) .

op allOids : Configuration -> Set{Oid} .

eq allOids(none) = empty .
eq allOids(< I : Router | > C) = I, allOids(C) .
eq allOids(M:Msg C) = allOids(C) .
```

The parameterless `iteration` strategy uses the `allOids` function to collect all the object identifiers of the configuration `C` captured by **`matchrew`**, and passes them to its second overloaded version. Before that, the timers of the routers are updated by the strategy `update-tick`, which executes the rule `update-timer` once per router.

```
var RT : RouteTable .   var IS : Set{Oid} .
var D : Configuration . var Ifs : Interfaces .

sd iteration(empty) := handleResponse ! .
sd iteration((I, IS)) := try(
    matchrew C s.t. < I : Router | table: RT, interfaces: Ifs > D := C
          by C using one(iteration(I, Ifs)) ;
    handleResponse *
  ) ;
  iteration(IS) .
```

Using that set of identifiers, the overloaded version `iteration(Set{Oid})` iterates over all the objects in the configuration. Each object is matched against a `Router` object pattern, where the identifier `I` is the same as the `I` in the strategy argument. If the matching succeeds, the third overloaded version of `iteration` is called to send responses from every interface of router, and an undetermined number of them or of those sent by previous routers will be received using the `handleResponse` strategy. These two strategies will be explained later, but we already see that the emission and reception of responses can be freely interleaved. Otherwise, if the matching does not succeed, the object identifier does not designate a router and the strategy fails inside the **`try`**, jumping to the recursive call to `iteration` for `IS`. Since the argument is a set, in each strategy call the matches will be multiple, and `I` will be bound to every element in the set.

Hence, objects will be visited nondeterministically in any possible order, and the strategy only enforces that they are visited only once in each rewriting path. This is convenient, because the order in which the routers issue their response is not fixed, and different outcomes could be obtained with different orders. The third overloaded version of `iteration` sends responses by invoking the `response` rule for the router `I` on each interface `NId`.

```
sd iteration(I, none) := idle .
sd iteration(I, NId |> N Ifs) := response[I <- I, NId <- NId]
                               ; one(iteration(I, Ifs)) .
```

Unlike the previous strategies, how `iteration` visits the interfaces is irrelevant, because the table is not updated in the meanwhile. Thus, every such call is surrounded by a `one` to avoid unnecessary computation. Alternatively, we could have forced an ordering of the interfaces, either by not making the `Interfaces` constructor commutative or by sorting them explicitly in the strategy, but we found the usage of `one` more abstract.

As we have anticipated, some response messages may remain in the configuration and others may be processed before the next object is addressed due to the `handleResponse *` expression in the `iteration` strategy over routers. At the end, where the set of identifiers is `empty`, all the remaining messages are processed using the normalization operator. The strategy `handleResponse` takes any multicast message in the soup and delivers it to all its recipients.

```
strat handleResponse            @ Configuration .
strat handleResponse : Set{Oid} @ Configuration .

sd handleResponse := matchrew C s.t.
  IPMessage(A, NId, O, P) D := C
  by C using (one(handleResponse(NId, O, P, allOids(C))) ;
         remove-message[A <- A, NId <- NId, O <- O, P <- P]) .

sd handleResponse(NId, O, P, empty) := idle .
sd handleResponse(NId, O, P, (I, IS)) :=
  try(readResponse[I <- I, NId <- NId, O <- O, P <- P]) ;
  one(handleResponse(NId, O, P, IS)) .
```

It is assumed that all the messages are received simultaneously by all the routers, and so `one` is used. The definition of this strategy follows the same scheme of the `iteration` overloaded versions, making each object accept the selected message by fixing the variables in the `readResponse` rule when applying it.

As an example, let us apply an iteration on the network of Fig. 7, where each router has its table filled with the networks to which it is connected, namely 1.0.0.0/8 and 2.0.0.0/8.

```
Maude> srew linear using iteration .

Solution 1
rewrites: 136244 in 699ms cpu (699ms real) (194856 rewrites/second)
result Configuration: < r1 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(1), 0, 0 >
    < 2 . 0 . 0 . 0 / 8, 1 . 0 . 0 . 2, netwk(1), 1, 0 >,
    interfaces: netwk(0) |> 1 . 0 . 0 . 1 / 8 >
  < r2 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(1), 0, 0 >
    < 2 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(2), 0, 0 >,
    interfaces: netwk(0) |> 1 . 0 . 0 . 2 / 8
                netwk(1) |> 2 . 0 . 0 . 2 / 8 >
  < r3 : Router | table:
    < 1 . 0 . 0 . 0 / 8, 2 . 0 . 0 . 2, netwk(2), 1, 0 >
    < 2 . 0 . 0 . 0 / 8, 0 . 0 . 0 . 0, netwk(2), 0, 0 >,
    interfaces: netwk(1) |> 2 . 0 . 0 . 1 / 8 >

No more solutions.
rewrites: 136244 in 699ms cpu (699ms real) (194856 rewrites/second)
```

Since this network is so small, an iteration is enough to propagate all the routing information, no matter in which order the responses occur. If a similar linear topology is designed with four routers, two iterations are needed to complete the routing information of the routers at both ends, and multiple results are obtained in the first iteration. Other configurations may lead to unwanted situations due to known faults of the protocol like the count to infinity problem. Techniques to mitigate them like the *divided horizon* and *poisoned response* [33] have also been specified [22].

4.4. Knuth-Bendix completion

In equational logic, given a set E of equations, the *word problem* is deciding whether two given terms t_1 and t_2 satisfy $t_1 =_E t_2$. Even though the problem is undecidable, incomplete procedures can be constructed for a wide range of instances.

$$\text{Deduce}\ \frac{\langle E, R\rangle}{\langle E \cup \{s = t\}, R\rangle}\ \text{if } s \leftarrow_R u \rightarrow_R t \qquad \text{Simplify}\ \frac{\langle E \cup \{s = t\}, R\rangle}{\langle E \cup \{u = t\}, R\rangle}\ \text{if } s \rightarrow_R^+ u$$

$$\text{Orient}\ \frac{\langle E \cup \{s = t\}, R\rangle}{\langle E, R \cup \{s \rightarrow t\}\rangle}\ \text{if } s > t \qquad \text{R-Simplify}\ \frac{\langle E, R \cup \{s \rightarrow t\}\rangle}{\langle E, R \cup \{s \rightarrow u\}\rangle}\ \text{if } t \rightarrow_R^+ u$$

$$\text{Delete}\ \frac{\langle E \cup \{s = s\}, R\rangle}{\langle E, R\rangle} \qquad \text{L-Simplify}\ \frac{\langle E, R \cup \{s \rightarrow t\}\rangle}{\langle E \cup \{u = t\}, R\rangle}\ \text{if } s \rightarrow_R^{\sqsupset} u$$

Fig. 8. Bachmair and Dershowitz's inference rules.

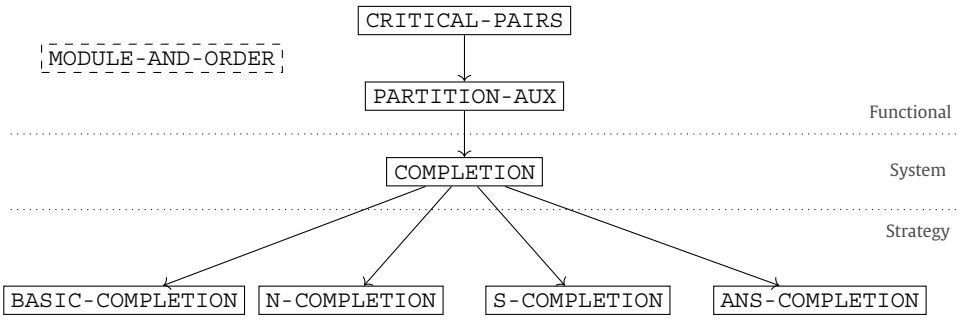


Fig. 9. Module structure of the basic Knuth-Bendix-completion procedures.

A *completion procedure* [4, §7] is a method to transform a set of equations into a convergent (confluent and terminating) rewriting system that goes back to Knuth and Bendix [30]. Whenever it succeeds, provable E-equalities can be decided by exhaustively reducing both sides using the calculated rules, and comparing their canonical forms syntactically.

Many concrete versions of the Knuth-Bendix completion procedure can be expressed as particular ways of applying a set of inference rules (see Fig. 8) proposed by Bachmair and Dershowitz [5], in other words, as specific strategies for that inference system. Lescanne [32] described various such procedures as a combination of *transition rules + control*, and implemented them in CAML. Completion procedures have also been implemented in ELAN [29], at the Maude metalevel [18], and even using the Maude strategy language [57]. However, the separation between rules and control is not clearly enforced in these works, since the rules are adapted to the specific data structure of each method. In this section, we propose an improvement of [57] in which a single fixed set of rules is used to express the different specific procedures as stateful strategies. These strategies are specified in separate strategy modules on top of the same system module `COMPLETION`, as shown in Fig. 9.

The basic state of the Knuth-Bendix-completion procedures is a pair (E, R) of equations and rules. The procedure begins with a set of equations $(E, \emptyset)$, and concludes with a set of rules $(\emptyset, R)$, unless it fails in finite time or it loops forever. Equations are oriented to become rules and critical pairs are calculated during the process. The syntax for equations and rules, the calculation of critical pairs, and the rest of the infrastructure are defined in the functional module `CRITICAL-PAIRS`. Since each example of equational system to be completed has its own term signature and since an order on the terms is required by the completion procedures, almost every module in Fig. 9 is parameterized by the `MODULE` theory or by its extension `MODULE-AND-ORDER`. Multiple term orders have been proposed in the literature and the algorithms can be instantiated with any of them by means of a view. For facilitating the specification of the order, we provide a parameterized module `LPO` that derives a *lexicographic path ordering* $>$ on the terms from a strict ordering $\gg$ on the symbols in the signature. The most basic definitions are those for equations `Eq`, rules `Rl`, and sets of these:

```
op _=._ : Term Term -> Eq [comm prec 60] .
op _->_ : Term Term -> Rl [prec 60] .

op __ : EqS EqS -> EqS [assoc comm id: mtEqS prec 70] .
op __ : RlS RlS -> RlS [assoc comm id: mtRlS prec 70] .
```

Bachmair and Dershowitz's inference rules and the state pair on which they operate are defined in the system module `COMPLETION`[10]:

```
mod COMPLETION{X :: MODULE-AND-ORDER} is
  pr CRITICAL-PAIRS{ForgetOrder}{X} .
```

[10] Since `CRITICAL-PAIRS` does not depend on the order, it is parameterized only by `MODULE`, and the `ForgetOrder` view from `MODULE` to `MODULE-AND-ORDER` allows instantiating it by the parameter `X`.

```
  sort System .
  op <_,_> : EqS RlS -> System [ctor] .

  var  E     : Eqs .
  vars R QR  : RlS .
  vars s t u : Term .

  *** [...] (the completion inference rules as rewrite rules)
endm
```

First, the rule `Orient` takes an equation $s \approx t$ and orients it as $s \to t$ whenever $s > t$.

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

Since the operator `=.` is commutative, the equation sides are interchangeable. Trivial equations are removed by `Delete`, and `Simplify` reduces any side of an equation by exhaustively rewriting it with the already generated rules. This is implemented, using Maude's metalevel facilities, by the `reduce` function imported from `CRITICAL-PAIRS`.

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

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

For their part, rules are simplified using the `R-Simplify` and `L-Simplify` rules, which simplify the left and righthand side of a rule respectively. Observe that there are two sets in the rule pattern, and the term is only reduced by the rules in `R`, while the set `QR` of *quiet rules* is not used. The different completion procedures could select which rules to reduce with by fixing `R` and `QR` conveniently.

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

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

Since reductions are decreasing with respect to the term order, the righthand side simplification still satisfies $s > u$. However, the order may not be preserved with `L-Simplify`, so the rule is removed and reinserted as an equation. Moreover, the lefthand side term s is not reduced exhaustively by all the rules in R as before, instead a single arbitrary rule $l \to r$ in R whose lefthand side l cannot be reduced by $s \to t$ is applied. This is the meaning of the $\to_{\overline{R}}^{\square}$ arrow in Fig. 8, and the behavior of the `reduce>` function. Finally, `Deduce` infers new identities by calculating the critical pairs of a rule `r` with a selected subset of rules `R`. Here, we have also included a set `QR` to allow restricting the rules whose critical pairs are calculated.

```
crl [Deduce] : < E, R QR >
            => < E equations(critical-pairs(r, R)), R QR >
if r R' := R QR .
```

Applying the inference rules carelessly does not always lead to a terminating and confluent set of rules, even if one exists. The second entry of the state is always a terminating rewriting system because of the strict order on the terms, but it may not be confluent if some critical pairs have not been calculated. Moreover, the deduction system itself is not terminating, since generating critical pairs leading to known identities may always be possible. Thus, strategies must be used to control the rule execution.

In the following, we describe a naive completion procedure taken from [56, Table 7.7] and the N-completion procedure of [32], both as strategies on Bachmair and Dershowitz's rules. The slightly more complex S-completion and ANS-completion procedures of [32] are available in the source code of the example [22].

4.4.1. Basic completion

The basic completion procedure [56, Table 7.7] follows the script below for a state (E, R):

1. Select an equation from E. If there is none, stop with success.
2. Simplify its both sides using the rules in R.
3. If both sides of the equation become identical, remove the equation and go back to step 1. Otherwise, continue to step 4.

4. Orient the equation and add the resulting rule to R. If this is not possible, stop with a failure.
5. Calculate the critical pairs between the new rule and the whole rule set, and add them as equations to E. Go back to step 1.

The procedure can be specified by the following `compl` strategy, which uses `deduction` as an auxiliary strategy. It consists of a **`matchrew`** combinator, whose content implements the five steps of the script, inside a normalization operator `!` that iterates them until they fail.

```
sd compl := (matchrew Sys s.t. < E s =. t, R > := Sys by
               Sys using (try(Simplify[E <- E, s <- s]) ;
                          try(Simplify[E <- E, s <- t]) ;
                          try(Delete[E <- E]) ;
                          (match < E, R > or-else
                             (Orient[E <- E] ;
                              deduction(R))
                          )
                         )
            ) ! .
```

First, the **`matchrew`** patterns select an equation from the state. If there is none, the matching will fail and the strategy execution will stop due to the normalizing operator. Otherwise, the equation is simplified (step 2) using the `Simplify` rule. By fixing the variable `E` in the lefthand side `< E s =. t, R >` of the rule, the simplification is applied only to the selected equation. Moreover, setting `s` in the rule to `s` and `t` alternatively in the strategy context ensures that both sides are reduced, and since the simplification rule fails when the term is already simplified, these rule applications are surrounded by a **`try`** operator. The simplified version of `s =. t` may coincide with another equation in `E`, and so be removed by the idempotence equation of the set. Moreover, the strategy attempts to `Delete` the equation if it has become a trivial identity (step 3). In either case, the substrategy finishes successfully and another iteration is started because of the normalizing operator. Otherwise, if the equation is already there, the **`or-else`** alternative is executed. The rule `Orient` orients the selected equation unless it is not orientable (step 4). In this latter case, the application fails, and so the iteration loop and the whole procedure stops. As there is still an equation in the state, the failure can be observed in the procedure's output. On success, the `deduction` strategy is called with the set of rules `R` from the matching, which contains all rules except the new one.

```
sd deduction(R) := matchrew Sys s.t. < E, R s -> t > := Sys by Sys using
                     Deduce[r <- s -> t, QR <- mtRlS] .
```

In order to generate the critical pairs between the new rule and the whole rule set, the `deduction` strategy definition applies `Deduce` with its variable `r` instantiated to that rule and `QR` to the empty set. The new rule `s -> t` is recovered in the **`matchrew`** by matching against `< E, R s -> t >` where `R` is instantiated to its value from the context, which is the argument of the strategy that contained all the rules except the new one. Using this procedure, all the critical pairs are generated for every rule, so whenever the strategy terminates with an empty equation set, the resulting set of rules is a convergent rewriting system [27].

As a simple example to illustrate the procedure, consider a signature with two unary symbols f and g, a lexicographic path ordering whose precedence is $f \gg g$, and an equation $f(f(x)) = g(x)$. The algorithm is run by:

```
Maude> srew < 'f['f['x:S]] =. 'g['x:S], mtRlS > using compl .

Solution 1
rewrites: 6943 in 6ms cpu (6ms real) (1048000 rewrites/second)
result System: < mtEqS, 'f['f['x:S]] -> 'g['x:S]
                        'f['g['x:S]] -> 'g['f['x:S]] >

No more solutions.
rewrites: 6943 in 6ms cpu (6ms real) (1048000 rewrites/second)
```

The algorithm has successfully finished, providing a confluent term rewriting system that solves the word problem for this equational theory.

4.4.2. N-completion

N-completion refines the previous procedure by the simplification of the rules and a more efficient computation of critical pairs. These rules are partitioned into a set T of rules whose critical pairs have not been calculated yet, and its complement $R \setminus T$ (*marked rules* in Huet's terminology [27]). This partition should be part of the procedure state and so, in the original implementation [57], the state pair was extended to a triple and the completion inference rules were changed as a consequence. Here, the information will be held in the strategy parameters:

```
strats N-COMP simplify-eqs @ System .
```

```
strats N-COMP success orient deduce simplify-rules : RlS @ System .

sd N-COMP    := N-COMP(mtRlS) .
sd N-COMP(T) := success(T) or-else (match < mtEqS, R >
                  ? deduce(T) : orient(T)) .
```

N-completion is implemented by a collection of strategies that take a set of rules as argument, which stands for the set T mentioned before, a subset of rules of the state term. The entry point is the parameterless overloaded version of `N-COMP` that calls its homonym strategy with T as an empty set. During execution, T is updated and passed on as an argument among the different mutually recursive strategies. According to the `N-COMP` definition, the procedure tries to execute one of the auxiliary strategies `success`, `deduce`, and `orient`. The first one tests whether the procedure has successfully finished, which is exactly when there are neither pending equations, nor rules whose critical pairs have not yet been calculated.

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

Remember that a call that does not match any definition is a failure, so when T is non-empty this strategy fails. In that case, the procedure continues either with the `deduce` or with the `orient` strategy, depending on whether the equation set is empty or not. Hence, equations are greedily oriented and added to T by `orient`, and only when there are no equations, the critical pairs are calculated by `deduce`. This strategy nondeterministically takes a rule `r` from T, deduces the identities from its critical pairs with respect to the rest of the rules, and then calls `simplify-rules`. Since the critical pairs for `r` have just been calculated, `r` is no longer included in the strategy argument. The definition is only executed if T is non-empty, but this is an invariant at this point because otherwise `success` would have succeeded earlier.

```
sd deduce(r T) := Deduce[r <- r, QR <- mtRlS] ;
                  simplify-rules(T) .

sd simplify-rules(T) := matchrew Sys s.t. < E, R > := Sys by Sys using (
          (L-Simplify[QR <- mtRlS] | R-Simplify[QR <- mtRlS])
          ? matchrew Sys' s.t. < E', R' > := Sys' by Sys' using
              simplify-rules(combine(T, R, R')) :
          : N-COMP(T)
        ) .
```

The `simplify-rules` strategy tries to simplify either the left or righthand side of any rule (inside or outside T) using the rest of the rules (for that reason, the quiet rules variable `QR` is set to the empty set), and calls itself recursively when it succeeds to make the simplification exhaustive. If no more simplifications are possible, a recursive call to `N-COMP` continues the procedure. `L-Simplify` and `R-Simplify` can modify the rule or convert it into an equation, so we should track the changes to update T. Using the two nested **`matchrew`**, we probe the rule set before and after the simplification, to compare them and find out what changed. The new T is calculated by a function `combine` defined in `PARTITION-AUX` by some simple equations.

$$\text{combine}(T, R, R') = \begin{cases} T & \text{if } r \notin T \\ (T \setminus (R \setminus R')) \cup R' \setminus R & \text{if } r \in T \end{cases}$$

The third auxiliary strategy, `orient`, is in charge of simplifying and orienting equations. The **`matchrew`** is added a condition `s > t` to know which is the rule that `Orient` would add, so that it can be included in the set T. Incidentally, this condition reduces the number of distinct matches of the subterm operator due to the commutativity of equation symbols.

```
sd orient(T) := simplify-eqs ;
                (match < mtEqS, R >
                  ? N-COMP(T)
                  : matchrew Sys s.t. < s =. t E, R > := Sys
                    /\ s > t by Sys using (
                      Orient[E <- E] ;
                      N-COMP(s -> t T)
                    )
                ) .

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

N-completion is more efficient than the initial procedure, and therefore fewer rewrites are required to compute the same completion of the previous section.

```
Maude> srew < 'f['f['x:S]] =. 'g['x:S], mtRlS > using N-COMP .
```

```
Solution 1
rewrites: 2279 in 3ms cpu (3ms real) (691654 rewrites/second)
result System: < mtEqS, 'f['f['x:S]] -> 'g['x:S]
                        'f['g['x:S]] -> 'g['f['x:S]] >

No more solutions.
rewrites: 2279 in 3ms cpu (3ms real) (691654 rewrites/second)
```

In order to show a more realistic example and to illustrate the importance of choosing an efficient strategy, we will apply the two procedures to the group axioms $e * x = x$, $I(x) * x = e$ and $(x * y) * z = x * (y * z)$, where $*$ is the binary group operation, I is the inverse, and e the identity element, whose precedence is set to $I \gg * \gg e$ for the lexicographic path ordering. The basic completion algorithm does not terminate in hours for this problem, because of the rapid growth of its search space caused by the inefficient calculation of critical pairs and the lack of rule simplification. On the contrary, N-completion quickly finds a solution using the depth-first search of the `dsrewrite` command.

```
Maude> dsrew [1]
      < '*['e.S ,'x:S] =. 'x:S  '*['I['x:S], 'x:S] =. 'e.S
        '*['*['x:S, 'y:S], 'z:S] =. '*['x:S, '*['y:S, 'z:S]],
        mtRlS > using N-COMP .

Solution 1
rewrites: 222369 in 309ms cpu (311ms real) (718093 rewrites/second)
result System: < mtEqS,
            '*['e.S,'x:S] -> 'x:S
            '*['x3:S,'*['I['x3:S],'z5:S]] -> 'z5:S
            '*['x3:S,'I['x3:S]] -> 'e.S
            '*['z2:S,'e.S] -> 'z2:S
            '*['*['x:S,'y:S],'z:S] -> '*['x:S,'*['y:S,'z:S]]
            '*['I['x:S],'x:S] -> 'e.S
            '*['I['y1:S],'*['y1:S,'z1:S]] -> 'z1:S
            'I['e.S] -> 'e.S
            'I['*['x3:S,'y5:S]] -> '*['I['y5:S],'I['x3:S]]
            'I['I['y1:S]] -> 'y1:S >
```

Using the calculated rewriting system (`result` in the following), we can check whether the term $I(x * (y * z))$ is equal to $(I(z) * I(y)) * I(x)$ by reducing both terms to normal form and comparing the results syntactically.

```
Maude> red reduce('I['*['x:S, '*['y:S, 'z:S]]], result) .
rewrites: 21 in 3ms cpu (2ms real) (6300 rewrites/second)
result Term: '*['I['z:S],'*['I['y:S],'I['x:S]]]

Maude> red reduce('*['*['I['z:S], 'I['y:S]], 'I['x:S]], result) .
rewrites: 19 in 0ms cpu (2ms real) (~ rewrites/second)
result Term: '*['I['z:S],'*['I['y:S],'I['x:S]]]
```

The strategies for the more complex S-completion and ANS-completion procedures follow the same principles, but these strategies receive more parameters standing for finer rule partitions. All the details can be found in [22].

5. Semantics

An informal description of the semantics of strategy expressions has already been given in the previous sections. Here, we complete and formalize this in two equivalent forms:

1. A denotational semantics that describes the results of a strategy execution as a set of terms. It is based on the partial, similar descriptions in [23,35].
2. A small-step operational semantics based on a rewrite theory transformation and expressed in Maude. It is an updated version of the one described in [36].

The first semantics is more abstract; it emphasizes the results of the strategy, as shown by `srewrite`, while the second details the evolution of the state as it is rewritten according to the strategy. Intermediate states are also interesting and relevant for some forms of analysis of the controlled systems like model checking. A similar operational semantics has been used to define model checking for such systems in [52,50]. Another advantage of the rewriting-based semantics is that it is executable and shows that the controlled system can be expressed in rewriting logic itself. This approach has been followed by other works like [13].

5.1. Set-theoretic semantics

The behavior of the strategy language expressions has been described in Section 3 by the results they produce for any given initial term. In previous conference papers [23,36], this description has been partially formalized as a *set-theoretic semantics*, where the denotation of a strategy expression α is a function from terms to sets of terms:

$$[\![\alpha \,@\, \bullet]\!] : T_\Sigma \to \mathcal{P}(T_\Sigma)$$

Following this approach, we present here an updated formal semantics of the whole language. To cover all the combinators, some circumstances should be taken into account that complicate the semantic description as follows:

- Strategy definitions and `matchrew` operators can bind variables to values that are accessible within the definition body and the substrategies, respectively. To pass these values on, variable environments are incorporated as input to the semantic function, and we write $[\![\alpha]\!](\theta, t)$ instead of $[\![\alpha \,@\, t]\!]$. Variable environments θ are represented by substitutions $\mathrm{VEnv} = X \to T_\Sigma$.
- With strategy modules, the semantic value of a strategy expression does not only depend on its sole content, but also on the strategy definitions D of the module in which it is evaluated. We describe these definitions as tuples $(sl, \vec{p}, \delta, C)$, where sl is the strategy name and δ is the expression to be executed when the input parameters match the lefthand side patterns $\vec{p}$ and satisfy the condition C. We assume that they are numbered from 1 to m, but we will later see that this order is immaterial.
  From the technical point of view, the possibility of defining recursive strategies implies that the semantics cannot be provided by a series of well-founded compositional definitions for each combinator. Hence, we have to resort to more complex tools from domain theory [1]. For the presentation of the language semantics, we will assume the existence of a denotation d_k for each definition such that $d_k = [\![\delta_k]\!]$, and express the meaning of a strategy call in these terms. Later, we will justify that $\Delta = (d_1, \ldots, d_m)$ can be obtained by a fixed point calculation.
- Recursive definitions and iterations make nonterminating executions possible. The denotation should indicate whether a computation is terminating or not, and this cannot be expressed by returning only a term set. This fact is valuable even for the compositional definition of the semantics, whose conditional expression is meant to execute its negative branch only when the condition strategy does not provide any result, which must be decided in finite time. Moreover, a non-terminating strategy evaluation can still provide solutions on other rewriting paths, as the `srewrite` and `dsrewrite` command do, so any combination of a set of results and a termination status may be possible.
  For those reasons, we extend the output range of the denotation by allowing a symbol $\bot$ representing non-termination to appear in the result sets, now $\mathcal{P}(T_\Sigma \cup \{\bot\})$. However, to avoid undesired ambiguities, infinite sets will be identified regardless of whether they contain $\bot$ or not, since only nonterminating executions are able to produce infinitely many results. This motivates the following definition for any set M:

  $$\mathcal{P}_\bot(M) := \mathcal{P}(M \cup \{\bot\}) \,/\sim$$

  where

  $$A \sim B \iff A = B \vee (A \text{ is not finite and } A \oplus B = \{\bot\})$$

  where $\oplus$ stands for the symmetric difference of sets $A \oplus B = A \cup B \setminus A \cap B$. Thus, $\mathcal{P}_\bot(T_\Sigma)$ will be the range of the semantic function, and we will also use $\mathcal{P}_\bot(\mathrm{VEnv})$ for some auxiliary operations. In the following, we do not refer to the equivalence classes explicitly but to the sets themselves, taking infinite sets with $\bot$ as representatives of their classes. Hence, we write $\bot \in A$ to express that A contains the symbol $\bot$ or it is infinite.

According to the previous comments, the final form of the denotation for a strategy α is

$$[\![\alpha]\!]_\Delta : \mathrm{VEnv} \times T_\Sigma \to \mathcal{P}_\bot(T_\Sigma)$$

As for a standard denotational definition in the domain theory framework, we have to see the class of denotations $\mathrm{SFun} = \mathrm{VEnv} \times T_\Sigma \to \mathcal{P}_\bot(T_\Sigma)$ as a *chain-complete partially ordered set* (ccpo), and prove that for any α the $\mathrm{SFun}^m \to \mathrm{SFun}$ functional that maps Δ to $[\![\alpha]\!]_\Delta$ is monotonic and continuous to calculate the definitions' semantics using the Kleene fixed-point theorem. Here, we will only highlight the basic ideas, and refer to [49] for additional details and proofs. The first step is endowing $\mathcal{P}_\bot(M)$ with an order to make it a ccpo[11]:

$$A \leq B \iff A = B \vee (\bot \in A \wedge A \setminus \{\bot\} \subseteq B)$$

Intuitively, the order expresses how results can be extended by further computation. When approaching $[\![\alpha]\!](\theta, t)$ by the results A_n of executions with at most n nested recursive calls, $\bot \in A_n$ if some recursive calls have not reached their base

[11] This order is a particular realization of a flat Plotkin powerdomain [42].

cases yet. These results can grow with more solutions as larger depths are allowed, and they can eventually get rid of $\bot$ or keep it forever if the execution does not terminate. On the contrary, sets without $\bot$ are definitive solutions, since all base cases have been reached, and so we call those sets *final*.

Proposition 1. *$(\mathcal{P}_\bot(M), \leq)$ is a chain-complete partially ordered set. Its minimum is the class of $\{\bot\}$, and its maximal elements are the classes of M and the final sets. The union of $\sim$-equivalence classes is well-defined by the union of its representatives, and for any chain $F \subseteq \mathcal{P}(\mathcal{P}_\bot(M))$, $\sup F = \bigcup_{A\in F} A$ if $\bot \in A$ for all $A \in F$, and $\sup F = Z$ if there is a $Z \in F$ such that $\bot \notin Z$. In this case, it is unique.*

The denotations $\text{SFun} = \text{VEnv} \times T_\Sigma \to \mathcal{P}_\bot(T_\Sigma)$ and $\text{SFun}^m \to \text{SFun}$ are also ccpos by standard results.[12] Since various strategy combinators (like concatenation) involve feeding a second strategy with the results of a first one, we will use this operation frequently. In the abstract, we can see it as a function $\text{let} : \mathcal{P}_\bot(N) \times (N \to \mathcal{P}_\bot(M)) \to \mathcal{P}_\bot(M)$ defined as follows for any $A \in \mathcal{P}_\bot(N)$ and $B : N \to \mathcal{P}_\bot(M)$:

$$\text{let}(A, B) := \{\bot \mid \bot \in A\} \cup \bigcup_{x \in A \setminus \{\bot\}} B(x)$$

For readability, instead of $\text{let}(A, B)$ we will use the informal notation $\text{let } x \leftarrow A : B(x)$ where $B(x)$ is any set expression depending on x. This functional is monotonic and continuous, and using it we can define a monotonic and continuous composition in SFun. Given two functions $f, g \in \text{SFun}$, we define $g \circ f$ as

$$(g \circ f)(\theta, t) := \text{let } u \leftarrow f(\theta, t) : g(\theta, u)$$

Then $(\text{SFun}, \circ)$ is a monoid with identity $\text{id}(\theta, t) = \{t\}$. As usual, we write f^n for the n-times composition $f \circ \cdots \circ f$ of f and f^0 for the identity.

Another prerequisite of the semantic infrastructure is matching. For the given rewrite theory $\mathcal{R} = (\Sigma, E \cup B, R)$, we assume that there is a function $\text{match} : T_\Sigma(X) \times T_\Sigma(X) \to \mathcal{P}(\text{VEnv})$ that provides all the minimal matching substitutions of a pattern p into a term t, i.e. all $\sigma : X \to T_\Sigma(X)$ that satisfy $\sigma(p) = t$ and $\sigma(y) = y$ for any variable $y \in X$ that does not occur in p. For matching anywhere and with extension, the functions amatch and xmatch are also considered respectively. Their output is represented by a pair $\text{VEnv} \times T_\Sigma(X \cup \ominus)$ where the first component is a substitution σ, and the second is the context c where the match occurs, marked by a distinct variable $\ominus \notin X$ such that $c[\ominus/\sigma(p)] = t$. However, we often write $c(t)$ for $c[\ominus/t]$.

5.1.1. Idle and fail

The semantics of the `idle` and `fail` constants follow directly from their descriptions.

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

5.1.2. Rule application

The evaluation of a rule application expression requires finding all matches of each rule with the given label and instantiated with the given initial substitution. In case of rules with rewriting conditions $l_i \texttt{ => } r_i$, the given strategies must be evaluated in l_i, instantiated by the substitutions derived from the previous fragments, and their results matched with r_i to check the conditions and instantiate their variables. We formally describe the application of all rules labeled rl with initial substitution ρ and the mentioned strategies as

$$\text{ruleApply}(rl, \rho, \alpha_1 \cdots \alpha_m, \theta, t) = \bigcup_{\substack{(rl,l,r,C)\in R \\ \text{nrewf}(C)=m}} \bigcup_{(\sigma_0,c)\,\in\,\text{amatch}(\rho(l),t)} \text{let } \sigma \leftarrow \text{check}(C, \sigma_0 \circ \rho, \alpha_1 \cdots \alpha_m, \theta) : \{c(\sigma(r))\}$$

where $\text{nrewf}(C)$ is the number of rewriting condition fragments in C. Then, the semantics of the application combinator is

$$[\![rl\,[x_1 \texttt{ <- } t_1, \ldots, x_n \texttt{ <- } t_n]\,\{\alpha_1, \ldots, \alpha_m\}]\!]_\Delta(\theta, t) = \text{ruleApply}(rl, \text{id}[x_1 \mapsto \theta(t_1), \ldots, x_n \mapsto \theta(t_n)], \alpha_1 \cdots \alpha_m, \theta, t)$$

In the definition of ruleApply, the strategies in the expression are passed to the check function. Since the strategies should be evaluated in the strategy variable context, the context substitution is also passed to the check function. Its full recursive definition is as follows:

[12] For any ccpos $(D, \leq)$ and $(E, \preceq)$ and any set N, $N \to D$ is a ccpo with the order $f \leq g$ iff $f(x) \leq g(x)$ for all $x \in N$, and $E \times D$ is a ccpo with the order $(x, y) \leq (u, v)$ iff $x \preceq y \wedge u \preceq v$.

$$\begin{aligned}
\text{check}(\texttt{true}, \sigma, \vec{\alpha}, \theta) &= \{\sigma\} \\
\text{check}(l = r \wedge C, \sigma, \vec{\alpha}, \theta) &= \text{check}(C, \sigma, \vec{\alpha}, \theta) \textbf{ if } \sigma(l) = \sigma(r) \textbf{ else } \emptyset \\
\text{check}(t : s \wedge C, \sigma, \vec{\alpha}, \theta) &= \text{check}(C, \sigma, \vec{\alpha}, \theta) \textbf{ if } \sigma(t) \in T_{\Sigma/E,s}(X) \textbf{ else } \emptyset \\
\text{check}(l := r \wedge C, \sigma, \vec{\alpha}, \theta) &= \cup_{\sigma' \in \text{match}(\sigma(l), \sigma(r))} \text{check}(C, \sigma' \circ \sigma, \vec{\alpha}, \theta) \\
\text{check}(l :=> r \wedge C, \sigma, \alpha\vec{\alpha}, \theta) &= \text{let } t \leftarrow [\![\alpha]\!]_\Delta(\theta, \sigma(l)) : \cup_{\sigma' \in \text{match}(\sigma(r), t)} \text{check}(C, \sigma' \circ \sigma, \vec{\alpha}, \theta)
\end{aligned}$$

The range of the check function is $\mathcal{P}_\perp(\text{VEnv})$, since the evaluation of rewriting condition fragments may not terminate. However, if the rule does not contain any such fragment, the result is always a plain finite substitution set.

If the rule application is surrounded by the `top` modifier, the matching must only occur at the top, so the amatch in the ruleApply definition is replaced by a xmatch. Finally, the `all` strategy constant represents a standard rewrite step with the rules in the current module. The rule's rewriting fragments are resolved by an unrestricted search, which is equivalent to using `all *` as the controlling strategy of the fragment.

5.1.3. Tests

Tests evaluate to a singleton set with the initial term whenever there is a match of the pattern such that the condition is satisfied. When the pattern does not match the term or no match makes the condition hold, its value is the empty set.

$$[\![\texttt{match}\ P\ \ \texttt{s.t.}\ C]\!]_\Delta(\theta, t) = \begin{cases} \{t\} & \exists \sigma \in \text{match}(\theta(P), t) \quad \text{check}(C, \sigma \circ \theta) \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

The matching pattern is previously instantiated by the context substitution. Notice that the last two parameters of check have been omitted, since C is an equational condition. Other test variants, such as `xmatch` and `amatch`, use their corresponding matching functions instead.

5.1.4. Regular expressions

The semantic value of concatenation is the composition of denotations

$$[\![\alpha\,;\beta]\!]_\Delta = [\![\beta]\!]_\Delta \circ [\![\alpha]\!]_\Delta.$$

This means that its results are the collection of the results of β for each result of α. In case the whole calculation of α does not terminate, the $\perp$ symbol is propagated to the composition. The alternation operator $\alpha \mid \beta$ has a straightforward definition

$$[\![\alpha \mid \beta]\!]_\Delta(\theta, t) = [\![\alpha]\!]_\Delta(\theta, t) \cup [\![\beta]\!]_\Delta(\theta, t)$$

as does the iteration strategy

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

From these definitions and $\alpha + \ \equiv\ \alpha\ \ ;\ \ \alpha *$, it follows that $[\![\alpha +]\!]_\Delta(\theta, t) = \bigcup_{n \geq 1} [\![\alpha]\!]^n_\Delta(\theta, t)$.

5.1.5. Conditionals

As described in Section 3, the condition of the *if-then-else* operator is a strategy itself that is evaluated to decide which branch to take. Any result for the condition α is continued by the positive branch, and discards the execution of the negative one.

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

The negative branch γ is only executed if the evaluation of α terminates without obtaining any solution. When $[\![\alpha]\!]_\Delta(\theta, t) = \{\perp\}$, neither β nor γ are evaluated, and the result of the conditional is $\{\perp\}$ by the first case.

The semantics of the derived operators can be obtained from the previous definitions. For example,

$$[\![\alpha\ \texttt{or-else}\ \beta]\!]_\Delta(\theta, t) = \begin{cases} [\![\alpha]\!]_\Delta(\theta, t) & [\![\alpha]\!]_\Delta(\theta, t) \neq \emptyset \\ [\![\beta]\!]_\Delta(\theta, t) & \text{otherwise} \end{cases}$$

$$[\![\texttt{test}\,(\alpha)\,]\!]_\Delta(\theta, t) = \begin{cases} \{t\} & \perp \notin [\![\alpha]\!]_\Delta(\theta, t) \neq \emptyset \\ \{\perp\} & \perp \in [\![\alpha]\!]_\Delta(\theta, t) \\ \emptyset & \text{otherwise} \end{cases}$$

$$[\![\texttt{not}\ (\alpha)\ ]\!]_\Delta(\theta,t) = \begin{cases} \{t\} & [\![\alpha]\!]_\Delta(\theta,t) = \emptyset \\ \{\bot\} & \bot \in [\![\alpha]\!]_\Delta(\theta,t) \\ \emptyset & \text{otherwise} \end{cases}$$

5.1.6. Rewriting of subterms

The rewriting of subterms operator is easily defined compositionally as

$$[\![\texttt{matchrew}\ P\ \texttt{s.t.}\ C\ \texttt{by}\ x_1\ \texttt{using}\ \alpha_1\ ,\ \ldots\ ,\ x_n\ \texttt{using}\ \alpha_n\ ]\!]_\Delta(\theta,t)$$
$$= \bigcup_{\sigma \in \text{mcheck}(t,P,C,\theta)} \text{let}\ t_1 \leftarrow [\![\alpha_1]\!]_\Delta(\sigma,\sigma(x_1)),\ldots,t_n \leftarrow [\![\alpha_n]\!]_\Delta(\sigma,\sigma(x_n)) : \sigma[x_1 \mapsto t_1,\ldots,x_n \mapsto t_n](P)$$

where $\text{mcheck}(t,P,C,\theta) = \bigcup\{\text{check}(C,\sigma_m \circ \theta) : \sigma_m \in \text{match}(\theta(P),t)\}$. The matched subterms $\sigma(x_k)$ are rewritten using the strategy α_k in the variable context $\sigma \circ \theta$, and their results replace them in the subject term, by reinstantiating the pattern with the modified matching substitution. Each combination of subterm results generates a potentially different solution, and if any of the subterm computations contains $\bot$, so does the matchrew.

The variations of the matchrew combinator, `amatchrew` and `xmatchrew`, have similar definitions but replacing match by the appropriate matching function, and rebuilding the matching context.

5.1.7. Pruning of alternative solutions

These semantics do not take the `one` operator into account, because that would add another layer of non-determinism and harden the understanding of this exposition. Although the rewriting process controlled by strategies can be nondeterministic, the set of results shown by the `srewrite` command and described in this section are ideally deterministic in the absence of `one`. However, `one` (α) nondeterministically chooses one of results that α produces, passing from the set $[\![\alpha]\!]_\Delta(\theta,t)$ to any singleton set $\{u\}$ with u in the previous set. In that case, the denotations should produce elements of $\mathcal{P}(\mathcal{P}_\bot(T_\Sigma))$.

5.1.8. Strategy calls

Strategy calls are resolved using the definition context $\Delta = (d_1,\ldots,d_m)$,

$$[\![sl\ (t_1,\ldots,t_n)\ ]\!]_\Delta(\theta,t)$$
$$= \bigcup_{(sl,p_1\cdots p_n,C,\delta_k)\in D\ \ \sigma\in \text{mmatch}(\theta(t_1)\cdots\theta(t_n),p_1\cdots p_n,C)} d_k(\sigma,t)$$

where D is the set of strategy definitions in the module, and $\text{mmatch}(t_1\cdots t_n, p_1\cdots p_n, C)$ is defined as the union of all $\text{check}(C,\sigma)$ for all minimal substitutions satisfying $\sigma(p_k) = t_k$ for all k. In other words, all strategy definitions matching the input arguments and satisfying the condition are executed, and their results are gathered in the final one. If no definition can be activated, the result is thus the empty set.

5.1.9. Correctness and strategy definition calculation

The exhaustive and well-founded definition of the semantics on the structure of strategy expressions implies its correctness for any environment $\Delta = (d_1,\ldots,d_m)$. However, it still remains pending how to formally construct such an environment where $d_k = [\![\delta_k]\!]_\Delta$. As anticipated in the introduction, the solution passes through Klenee's fixed point theorem and the monotonicity and continuity of the denotations.

Theorem 1. *For any strategy expression α, $[\![\alpha]\!]_{(d_1,\ldots,d_m)}$ is monotone and continuous in $d_1,\ldots,d_m$, and so is the operator $F : \text{SFun}^m \to \text{SFun}^m$*

$$F(d_1,\ldots,d_m) := \left([\![\delta_1]\!]_{(d_1,\ldots,d_m)},\ldots,[\![\delta_m]\!]_{(d_1,\ldots,d_m)}\right)$$

Hence, F has a least fixed point $\mathsf{FIX}\,F \in \text{SFun}^m$ which can be calculated as

$$\mathsf{FIX}\,F = \sup\{F^n(\{\bot\},\ldots,\{\bot\}) : n \in \mathbb{N}\}$$

Then, the denotation for the definition k is formally defined as

$$d_k := (\mathsf{FIX}\,F)_k\,,$$

and satisfies $d_k = F_k(\mathsf{FIX}F) = [\![\delta_k]\!]_\Delta$.

If Maude were running on an idealized unbounded-memory machine, the strategy search command `srewrite in SM : t using` α would eventually return any solution in $[\![\alpha]\!]_\Delta(\text{id},t)$, and it would finish if and only if $\bot \notin [\![\alpha]\!]_\Delta(\text{id},t)$. In the same situation, `dsrewrite` will also return the full set of solutions, but if $\bot$ is in the denotation, some solutions may be missed.

5.2. Rewriting semantics

This section presents a rewriting-based semantics that transforms a pair (M, SM), i.e. a system module M along with a strategy module that defines strategies for M, into a rewrite theory $\mathcal{S}(M, SM)$, where strategy expressions can be written and applied to terms. The transformed module implements the syntax of the strategy language and the infrastructure and rules to apply them. For this purpose, some function definitions and rules should be added to the transformed module for each strategy construct. First of all, the signature of the transformed module should include some auxiliary infrastructure for substitutions and matching. Declarations with a type annotation like S are generated for each sort in M.

```
sort Substitution .

op _<-_ : VarS S -> Substitution [ctor] .
op none : -> Substitution [ctor] .
op _,_ : Substitution Substitution -> Substitution [ctor assoc id: none] .

op _·_ : S Substitution -> S .
```

A substitution is defined as a list of variable-to-term bindings, and an infix dot operator represents the application of a substitution to a term.

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

op [] : -> S [ctor] .

op getMatch : S S EqCondition -> MatchSet .
op getAmatch : S S' EqCondition -> MatchSet .
op getXmatch : S S EqCondition -> MatchSet .
```

Each of the last three operators returns all matches of its second argument into the first, respectively on top, anywhere, or on top with extension. A match is described by a pair containing a substitution and a context. The context *hole* is indicated by means of an overloaded constant `[]`.

```
sorts Condition EqCondition .
subsort EqCondition < Condition .

op trueC : -> EqCondition  [ctor] .
op _=_ : S S -> EqCondition [ctor] .
op _:=_ : S S -> EqCondition [ctor] .
op _: S : S -> EqCondition [ctor] .
op _=>_ : S S -> Condition [ctor] .

op _/\_ : Condition Condition -> Condition [ctor assoc id: trueC] .
op _/\_ : EqCondition EqCondition -> EqCondition [ditto] .

op _·_ : Condition Substitution -> Condition .
```

Equational and rule conditions are defined with the usual syntax, and a substitution can also be recursively applied to them.

Second, strategy language constructs are expressed as Maude operators. Its signature is similar to the meta-representation of strategies in Section 3.12, but applied at the object level. Here we only include some of them as examples.

```
sorts RuleApp Strat StratCall .
subsorts RuleApp StratCall < Strat < StratList .

op _[_]{_} : Label Substitution StratList -> RuleApp [ctor] .
op match_s.t._ : S EqCondition -> Strat [ctor] .
op _;_ : Strat Strat -> Strat [ctor] .
*** and more

op _·_ : Strat Substitution -> Strat .
```

Substitutions can be applied to strategy expressions too. The equational definition is straightforward, except for the **matchrew** case. Pattern variables that designate subterms to be rewritten cannot be replaced syntactically, because the reference would be lost. However, the conflicting substitution assignment can be translated into an equality condition fragment to be added to the strategy expression.

```
ceq matchrew(P:S, C, VSL) · Sb =
        matchrew(P:S · SSb, SCond /\ C · Sb, VSL · Sb)
 if { SSb ; SCond } := splitSubs(Sb, VSL) .

eq splitSubs(X:S <- T:S ; Sb, X:S using E) = { Sb ; X:S = T:S } .
eq splitSubs(Sb, X:S using E) = { Sb ; nil } [owise] .
ceq splitSubs(X:S <- T:S ; Sb, (X:S using E, VSL)) =
  { Sb' ; C /\ X:S = T:S) if { Sb' ; C } := splitSubs(Sb, VSL) .
eq splitSubs(Sb, (X:S using E, VSL)) = splitSubs(Sb, VSL) [owise} .
```

Third, the strategy execution infrastructure is based on a series of tasks and continuations.

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

op <_@_> : Strat S -> Task [ctor] .
op sol :  S -> Task [ctor] .
op <_;_> : Tasks Cont -> Task [ctor] .

op chkrw : Condition StratList S S -> Cont [ctor] .
op seq : Strat -> Cont [ctor] .
op ifc : Strat Strat S -> Cont [ctor] .
op mrew : S S' Substitution VarStratList -> Cont [ctor] .
op onec : -> Cont [ctor] .
```

The application of a strategy α to a term t is represented by a task `< α @ t >`, and solutions are captured in `sol(t)` tasks. Tasks can be rewritten and fork new tasks, which represent different search states. They are all gathered in an associative and commutative soup of sort `Tasks`. Nested searches are represented by the `<_;_>` constructor, which additionally contains a *continuation* that the results from the inner search must execute to be a solution for the outer execution level. Continuations are specified as terms of sort `Cont`.

5.2.1. Idle and fail

The **`idle`** and **`fail`** meaning is given by the following rules that convert the `idle` task to a solution, and remove the `fail` task.

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

5.2.2. Rule application

For each unconditional rule or each conditional rule without rewriting fragments l `=>` r with label *label* in M, a rule as below is appended to the transformed module:

```
crl < label[Sb]{empty} @ T:S > => gen-sols(MAT, r · Sb)
if MAT := getAmatch(l · Sb, T:S, C) .

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

The possibly-empty substitution `Sb` from the application expression is applied to both sides of the rule. Then the partially instantiated lefthand side is matched against the subject term, and the resulting matches are passed to the `gen-sols` function. This function traverses the set generating a solution task for each match, by instantiating the righthand side of the rule with the matching substitution, and building up the context with `replace`.

The treatment of rewriting conditions is much more involved, because they must be rewritten according to the given strategies. To handle this situation, we make use of a continuation. Consider a rule

```
crl [label] : l => r if C0 /\ u1 => v1 /\ C1 /\ ... /\ Cn-1 /\ un => vn /\ Cn .
```

where C_k are equational conditions, which may be empty. Let RC be the condition fragments from C_1 to C_n. For each such rule, we generate

```
var C  : EqCondition .
var RC : Condition .
```

```
crl < label[Sb]{E1, ..., En} @ T:S >
   => gen-rw-tks(MAT, u1 · Sb, (u1 => v1 /\ RC) · Sb,
                 E1 ... En, r · Sb)
   if MAT := getAmatch(l · Sb, T:S, C0) .

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

The function `gen-rw-tks` traverses the set of matches like `gen-sols`, but a continuation task is generated for each match. Its nested computation applies the first given strategy to the lefthand side of the first rewriting fragment instantiated by the matching substitution. Its `chkrw` continuation stores the condition, the pending controlling strategies, the rule righthand side and the subterm context. This allows checking the condition recursively and stepwise. When a solution is obtained in the nested computation, it must be matched against the righthand side of the fragment and all the matches must be continued as potentially different condition solutions.

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

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

Here, `gen-rw-tks2` walks over the matches for the righthand side of the previous condition fragment, and generates continuation tasks that evaluate the next condition fragment as already done for the initial fragment. Clearly, the base case of this process is reached when no rewriting fragment remains.

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

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

The function `gen-sols2` finally composes the solutions of the rule application by rebuilding the term using the successively instantiated righthand side of the rule. In the case that any of the nested strategy evaluations fails, the whole rule application fails.

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

5.2.3. Tests

As described before, tests behave like `idle` if there is a match satisfying the condition, and like a `fail` otherwise.

```
crl < match P:S s.t. C @ T:S > => sol(T:S)
 if < Sb, Cx:S > MAT := getMatch(P, T:S, C) .

crl < match P:S s.t. C @ T:S > => none
 if getMatch(P:S, T:S, C) = none .
```

The **`amatch`** and **`xmatch`** variants are defined by similar pairs of rules. The only difference is the search function, `getAmatch` and `getXmatch` respectively, which can be implemented in Maude by means of the family of `metaMatch` functions.

5.2.4. Regular expressions

Regular expressions can be handled by a series of simple rules:

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

The rule for alternation splits the task into two subtasks, where each of them continues with one of the alternatives. The rule for concatenation creates a nested task to evaluate the first of the concatenated strategies and leaves the second strategy pending using the `seq` continuation. Each solution found in the nested search is then continued using the strategy in the continuation. When the subsearch runs out of tasks, the task is discarded. The iteration rule, following its recursive definition, produces both a solution for the empty iteration, and a task that evaluates the iteration body concatenated with the iteration itself. The non-empty iteration is equationally reduced to this equivalent expression.

5.2.5. Conditionals

The semantics of conditionals is also expressed by a continuation and a subsearch for the strategy condition. The `ifc` continuation maintains the strategies for both branches of the conditional and the initial term, which will be used if the negative branch has to be evaluated.

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

When a solution is found for the condition, the result is given a task to be continued by the positive branch strategy. Moreover, the conditional `ifc` continuation is transformed in a `seq` continuation, since the execution of the negative branch is already discarded. On the other hand, if the tasks in the subcomputation get exhausted, the negative branch is evaluated in the initial term by means of a new task.

The semantics of the derived operators is implicitly given by equationally translating them into their equivalent expressions:

```
eq E or-else E' = E ? idle : E' .
eq not(E) = E ? fail : idle .
eq try(E) = E ? idle : idle .
eq test(E) = not(not(E)) .
```

5.2.6. Rewriting of subterms

The rewriting of subterms operator requires rewriting each subterm found by the given strategy. Like for rewriting conditions, this is handled using a continuation `mrew(P, Sb, Cx, X, VSL)` that holds the main pattern `P`, the substitution `Sb` and context `Cx` of its occurrence in the subject term, the variable whose subterm is currently being rewritten, and the list of pending $\overline{t}$ **using** $\overline{\alpha}$ pairs.

```
crl < amatchrew(P:S, C, VSL) @ T:S0 > => gen-mrew(MAT, P:S, VSL)
 if MAT := getAmatch(P:S, T:S0, C) .

eq gen-mrew(none, P:S, VSL) = none .
ceq gen-mrew(< Sb, Cx:S0 > MAT, P:S, VSL) =
    < < E · Sb @ X:S' · Sb > ; mrew(P:S, Cx:S0, Sb, VSL) >
    gen-mrew(MAT, P:S, VSL)
 if X:S' using E := firstPair(VSL) .
```

For each match of the main pattern in the subject term, a continuation task is created. It starts to evaluate the first strategy in the matched subterm, which is recovered by `X:S' · Sb`. The substitution `Sb` is also applied to the strategy, since it is allowed to contain free occurrences of the pattern and condition variables.

When the evaluation of a subterm gives a solution, the `mrew` task is split into two subtasks: the first one keeps looking for other solutions for the same subterm, and another one continues with the evaluation of the next subterm. The creation of the last task is similar to the initial case, but the information is instead obtained from the continuation. The result of the subterm rewriting is substituted in the copy of the main pattern carried by the continuation. This way, when all the subterms are processed the copy of the pattern will have the initial subterms replaced by some results, so that the rest of the variables can be instantiated with the initial substitution, and the initial term is rebuilt with the new subterms by means of the context stored in the continuation.

```
crl < sol(T:S') TS ; mrew(P:S, Cx:S0, Sb, (X:S' using E', VSL)) > =>
   < TS ; mrew(P:S, Cx:S0, Sb, (X:S' using E', VSL)) >
```

```
    < < E · Sb @ Y:S'' · Sb > ; mrew(P:S · (X:S' <- T:S'), Cx:S0, Sb, VSL) >
  if Y:S'' using E := firstPair(VSL) .

rl < sol(T:S') TS ; mrew(P:S, Cx:S0, Sb, X:S' using E) > =>
   < TS ; mrew(P:S, Cx:S0, Sb, X:S' using E) >
   sol(replace(Cx:S0, P · (X:S' <- T:S')) · Sb)  .

rl < none ; mrew(P:S, Cx:S0, Sb, VSL) > => none .
```

When the subterm search tasks are exhausted, the whole **amatchrew** execution is discarded. Identical rules are used for the other variants, **matchrew** and **xmatchrew**, except for the first rule, where `genAmatch` should be replaced by the appropriate function.

5.2.7. Pruning of solutions

The semantics of the `one` combinator can be expressed using a trivial continuation `onec`:

```
rl < one(E) @ T:S > => < < E @ T:S > ; onec > .
rl < sol(T:S) TS ; onec > => sol(T:S) .
rl < none ; onec > => none .
```

Which solution is selected depends on the internal strategy of the rewriting engine for applying rules and ordering matches. Using the `search` command, every possible solution will be selected in some rewriting branch. Better performance is obtained if the second rule above is run just after the first solution appears inside the task, so that no unnecessary work is done. This is a situation were strategies are valuable "at the meta-level", that is for the rewrite theory $\mathcal{S}(M, SM)$.

5.2.8. Strategy modules and calls

Strategy modules, their declarations and definitions, can be represented as Maude terms, like we did for the metalevel in Section 3.12. To simplify this presentation, we assume that the strategy definitions are collected in a definition set `DEFS`.

```
eq DEFS = (slabel(p1, ..., pn), δ, C) , ... .

rl < SC:StratCall @ T:S > => find-defs(DEFS, SC:StratCall, T:S) .

eq find-defs(none, SC, T:S) = none .
ceq find-defs((Slhs, Def, C) Defs, SC, T:S) =
    find-defs2(MAT, T:S, Def) find-defs(Defs, SC, T:S)
 if MAT := getMatch(Slhs, SC, C) .

eq find-defs2(none, T:S, Def) = none .
eq find-defs2(< Sb, Cx:S > MAT, T:S, Def) =
   < Def · Sb @ T:S > find-defs2(MAT, T:S, Def) .
```

The function `find-defs` traverses all the strategy definitions in `DEFS` and tries to match the strategy call term with their lefthand sides, and check their equational conditions. The strategy `find-defs2` takes these matches and produces a task `< Def · Sb @ T:S >` for each of them, to continue rewriting `T` with the definition strategy, whose free variables are bound according to the matching substitution.

5.3. Relating both semantics

The previous semantics are equivalent in the sense specified in the following proposition, i.e., they produce the same solutions and terminate for the same input data.

Proposition 2 *([49]). In any module (M, SM), for any term $t \in T_{\Sigma/E}$, and for any strategy expression α, $t' \in [\![\alpha]\!]_\Delta(\theta, t)$ iff $< \alpha \,@\, t > \rightarrow^*_{\mathcal{S}(M,SM)} \texttt{sol}\,(t')$ TS for some TS of sort* `Tasks`*. Moreover, $\bot \in [\![\alpha]\!]_\Delta(\theta, t)$ iff there is an infinite derivation from $< \alpha \,@\, t >$ in $\mathcal{S}(M, SM)$.*

Proof sketch. The proof proceeds by generalizing the statement to $t' \in \mathrm{dsem}(TS)$ iff $TS \rightarrow^*_{\mathcal{S}(M,SM)} \texttt{sol}\,(t')\ TS'$, where $\mathrm{dsem} : T_{\mathcal{S}(M,SM)} \rightarrow \mathcal{P}_\bot(T_\Sigma)$ is an extended denotation satisfying $\mathrm{dsem}(< \alpha \,@\, t >) = [\![\alpha]\!]_\Delta(\mathrm{id}, t)$. Induction is carried out, first on the order of the approximants of the denotational semantics, and then structurally on the semantic terms. □

6. Implementation

The first prototype for the language was a reflective implementation built as an extension of Full Maude, similar to the rewriting semantics described in Section 5.2. After some experimentation with that prototype, the strategy language

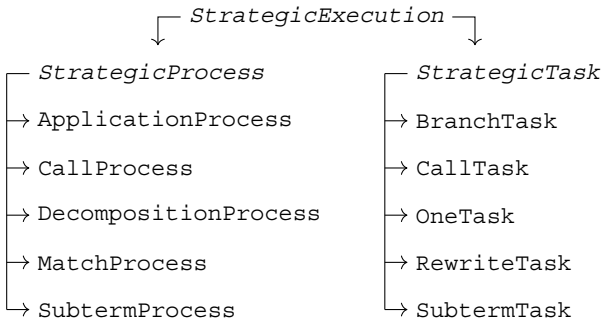


Fig. 10. Class diagram of the main processes and tasks.

started to be incorporated into the builtin Maude functionality, programmed in C++. This implementation provides better performance and integration with the rest of the Maude system, the module hierarchy, and the universal reflective theory.

Implementing a nondeterministic language is a challenging task that involves several nontrivial design decisions. The commands `srewrite` and `dsrewrite` must explore the rewriting graph permitted by the strategy, and this graph may contain loops and other non-terminating rewriting sequences. Moreover, recursion and the existence of rewriting conditions in rules allow a state to have infinitely many successors, i.e., the rewriting graph need not be finitary. Hence, the best we can do, on a both potential infinitely branching and infinitely deep tree, is to support *fairness*: whenever t can be rewritten to t' using a strategy, the solution t' should be found in a finite computation. Nevertheless, this computation may not be practical in terms of memory and time.

The fair implementation consists on a series of *processes* and *tasks* (see Fig. 10). Processes make the tree grow by processing expressions, applying rules, and calling strategies. Tasks are used to confine subsearches and continuations, and delimit variable contexts. Roughly speaking, processes can be understood as the `<_@_>` tasks of the rewriting semantics in Section 5.2, and tasks as `<_;_>`-terms. The same infrastructure is used for the depth-first search `dsrewrite` command, but processes are run differently.

There are other alternatives to implement a strategy language, which can also be compiled instead of interpreted. Specifications in ELAN, where rules and strategies are mixed together, can be compiled to an executable binary through a C code generator [59,40]. However, this will perhaps be simpler in that language where strategies are by default deterministic, unless some combinators are used. Nondeterminism is handled by a custom library supporting choice points and backtracking. In a recent work [46], an alternative implementation of a probabilistic extension of Maude strategies has been written in Python to generate probabilistic graphs. Strategies are compiled into an assembly-like language that is then executed by a virtual machine without explicit processes and tasks. Some aspects of these other approaches could be taken into consideration to improve the Maude implementation in the future.

6.1. Processes

We contemplate the growing search tree as a pool of processes, each with a subject term to rewrite and a stack of strategy expressions to use. For achieving fairness, they are executed in round-robin by the `srewrite` command, while `dsrewrite` command follows a FIFO policy. Processes are similar to the `<_@_>` tasks of the rewriting semantics, where the strategy continuation is replaced by a stack of strategy expressions. This stack essentially corresponds to a concatenation of strategies, and arise naturally from the accumulation of pending strategies during the execution of complex expressions. All processes are kept in a circular double-linked list with a moving pointer to the currently active process, which does a small amount of search computation and hands over the baton to the next one.

There are several types of processes specialized in different tasks, each a subclass of an abstract class `StrategyProcess` in the implementation. The process in charge of the actual rewriting is the *application process*. Generated by an application expression, in each turn it tries to make a new rewrite by finding the next position and rule that can be applied, considering the given label and initial substitution. Another essential process is the *decomposition process*, since it pops strategy expressions from the pending stack, and acts according to their types. Each kind of strategy expression is also a subclass of an abstract class `StrategyExpression` with a method that specifies how each one should be *decomposed* by the decomposition process. For example, **`idle`** is handled by doing nothing, so that the strategy is simply removed from the stack. On the other hand, **`fail`** causes the process to terminate, aborting its search path. Concatenations $\alpha \ ; \ \beta$ simply push their arguments to the stack in the right order, and the application strategy inserts an instance of the aforementioned application process into the list. Nondeterministic strategies like the alternation and the iteration spawn a copy of the decomposition process for each alternative. When this process reaches the bottom of the stack, it informs its parent that the current term is a solution.

Strategy calls are handled by *call processes*, which try to match the strategy call term with the lefthand side of the definitions in the module, and create a decomposition process for each match in any of the definitions, one at a time.

Finally, *subterm processes* find all matches of a `matchrew` main pattern in the subject term, and create a decomposition process for each selected subterm with its associated strategy. These are controlled and gathered under a *subterm task*.

6.2. Tasks

Executing process independently is not enough, as we have seen with the rewriting semantics. Sometimes we want to treat a chunk of the search tree as an entity in its own right, as a subsearch or as a call frame. For example, the execution of the conditional operator $\alpha\,?\,\beta : \gamma$ needs to know whether α has reached a solution to discard or activate the evaluation of the negative branch. This was achieved in the rewriting semantics of Section 5.2 using `<_;_>` terms, which gather a collection of processes along with a continuation. The various tasks of the C++ implementation, each a subclass of an abstract class `StrategicTask`, essentially reproduce the continuations of the semantics.

Since searches and contexts can be nested, tasks form a hierarchy. Each process and each task (except the root task) belongs to some task and lives in a task-local double-linked list, different from the global list of processes. New processes created by processes in a task are usually attached to that same task and sometimes to the parent task, if they do not logically belong to the subsearch. Tasks and processes notify their parents whether they have succeeded or terminated, and unlink themselves from their owner's list.

One of the most important tasks is the *branch task*, charged of dealing with conditionals $\alpha\,?\,\beta : \gamma$ and its derived operators. When informed by one of its processes that a solution has been reached, it creates a decomposition process on the parent task with the original pending stack but with β on top. If the branch task runs out of processes without finding a solution, the negative branch is executed by a decomposition process from the initial term. Similar actions are performed for the combinators `not` and `test`. The execution of the `one`(α) strategy is achieved by a task that executes α and interrupts the subcomputation as the first solution is reported.

Other nested computations are the evaluation of rewriting condition fragments, which is done by *rewrite tasks* and *match processes*, and the rewriting of subterms. In the latter case, a *subterm task* is used to delimit the variable context inside the `matchrew` and manage the term reconstruction with the results of the evaluation of the subterms. Each subterm is processed in a separate child task, and its results are stored in a table from which all the combinations of subterm results are generated. They are searched in parallel, since their processes are part of the same process list. Finally, *call tasks* house strategy call evaluations and separate their variable contexts.

6.3. Modules

Strategy modules and theories are implemented like their functional and system counterparts. In addition to extending the Maude grammar with the statements of strategy modules and the strategy-specific commands, the data structure for modules is extended with slots for two additional kinds of module items: strategy declaration and strategy definitions. When processing the module, strategy declaration are checked to be well-formed and free of unbound variables, like other statements are. All the machinery of importation, renaming, and parameterization has also been extended to work with strategy modules, theories, and strategy mappings in views.

One of the challenges when implementing strategy modules is how to efficiently handle the execution of strategy definitions. This involves matching the strategy call term with all definitions in the module to find which should be executed. We wanted to reuse the existing infrastructure for pattern matching. However, strategy calls are not terms in any signature, only their arguments are, and matching each argument separately is not reasonable. In effect, combining their respective matching substitution is far from being straightforward, and the optimized matching algorithms of Maude rely on a global view of the pattern for efficiency. Our solution is creating a hidden sort for strategy calls in each module and hidden tuple symbols for each strategy signature, which are also used in other parts of the Maude implementation. Each strategy definition and each call expression holds a term constructed with such a tuple for the lefthand side pattern or the call arguments, which can then be matched efficiently.

6.3.1. Metalevel

The reflection of the strategy language, strategy modules and theories at the metalevel consists of the declaration of their metarepresentation in the modules `META-STRATEGY`, `META-MODULE`, and `META-THEORY` of the Maude prelude. Moreover, some more routine changes are needed in the C++ implementation to translate back and forth between metarepresentation and the usual representation of strategy expressions and statements. The `metaSrewrite` descent function does this conversion and then executes the given strategy just like the `srewrite` and `dsrewrite` commands.

6.4. Efficiency considerations

In order to avoid repeating work, tasks maintain a hash set of those states seen in previous executions, identified by their subject term and the pending strategies stack. When the decomposition process is executed, this set is checked and redundant search paths are aborted. This optimization is sound since all processes in the same task belong to the same subsearch and share the same variable environment. Checking the equivalence of two states involves comparing the terms

Table 1
Performance comparison between the original prototype and the C++ implementation.

Example	Time (ms)		Measure proportion (old/new)		
	New	Old	Time	Rewrites	Memory peak
Sudoku [54]	908	90639	99.82	19.9	20.39
Neural networks [55]	1061	26496	24.97	8.16	1.64
Eden [26]	12	1360	113.33	417.61	2.74
Ambient calculus [44]	16	125735	7858.44	85759.7	180.47
Basic completion (Section 4.4)	91	7322	80.46	66.56	2.2
Blackboard [35]	1373	47027	34.25	5451.67	0.27
15-puzzle (Section 4.2)	182	2232	12.26	4.69	3.02

and strategy stacks. The comparison of the latter is straightforward, since stacks are represented as shared nodes in a persistent global tree, whose branches are built by alternative push operations on the same stack.

Regarding strategy calls, we keep a list of definitions for each named strategy, so that they can be quickly checked on a strategy call. For optimizing a common case, call processes are not generated for strategies that are defined by a single unconditional definition without input parameters, whose defining expression is directly pushed on top of the stack.

6.5. Performance comparison

As explained in the introduction, the strategy language was first implemented in Maude as an extension of Full Maude. We have compared the performance of the new C++ implementation with respect to that prototype using the original examples written for it. Since we have fixed some bugs and made some improvements when updating these examples to the current version of the language, instead of using their original code for the benchmarks, we have slightly adapted the renewed versions to run in the prototype. Additionally, we have included the 15-puzzle in the comparison, since it is the running example of this article. Moreover, the strategy language prototype was based on an old version of Full Maude that is incompatible with the current Maude releases, so we have minimally adapted it to run in Full Maude 3.1. This version of the extended interpreter comes with the strategy language as standard through the Core Maude implementation, but we have disabled what may interfere with the prototype. Notice that the meaning of the strategy search command differs from the prototype to the current version: the former's `srew` command looks for a single solution of the strategy, and we identify it with the new `dsrew [1]` command; its `srewall` command is the current `srew` that looks for all solutions of the strategy.

Table 1 collects some cost measures of the execution of both implementations on each example. The last three columns show the quotient of these measures with the C++ implementation in the denominator, so that the improvement is greater as these numbers are. The total time and number of rewrites are calculated as the sum of those printed by Maude for each command, thus excluding module parsing time. The value of the memory peak refers to the whole example execution, and the memory used by the interpreter and the prototype have been subtracted before calculating the coefficients. Otherwise, the prototype always uses much more memory due to Full Maude.

While the improvement is not uniform in all examples, the C++ unsurprisingly outperforms the Maude-based one in all of them. For the semantics of the ambient calculus, which is the example with the highest absolute execution time, the speedup is very significant. The memory usage is also drastically reduced. In the specification of Eden, a parallel Haskell-like programming language, the difference is not bigger because we have not included examples that finish in few milliseconds in the C++ implementation but take too much time to be waited for in the prototype. The same happens with the basic completion procedures of Section 4.4. The higher memory usage peak of the C++ implementation in the blackboard example is caused by the allocation of nodes of the directed acyclic graph in which terms are represented, and it could be explained by the way Maude reserves and reuse memory for them. In absolute terms, the memory peak of the Core Maude implementation is 26.71 Mb while that of the prototype is 119.17 Mb, and the total memory even after discounting the total memory used by Full Maude alone is 22.77 times greater in the Maude-based implementation.

This comparison exhibits some other advantages of the new implementation in addition to performance, since translating some of these examples back to the old prototype was not an easy task in some cases. Its strategy modules do not admit parameterization, module importation, or even selecting which module is being controlled. Hence, modular and parametric strategy specifications must be flattened in a single module of the prototype. This is specially visible in the basic completion example. Moreover, the syntactical analysis of strategy modules is less robust in the prototype, errors are less informative, and strategies with more than two arguments cannot be declared.

7. Strategy language comparison

In the Introduction, we mentioned that the Maude strategy language has been influenced by other existing strategic rewriting systems like ELAN (1993) [8], Stratego (1998) [12], and TOM (2001) [6]. While ELAN and Maude are standalone specification languages, the last two have specialized goals. Stratego is aimed at *program transformation* and it is now distributed as part of the Spoofax Language Workbench [60], a platform for the development of textual, usually domain-specific,

Table 2
Strategy language syntax comparison.

Maude	ELAN	Stratego	TOM	ρLog
`idle`	`id`	`id`	`Identity`	`Id`
`fail`	`fail`	`fail`	`Fail`	
label	*label*	*label*	Inline rules	*label*
`top`	By default	By default	By default	By default
`;`	`;`	`;`	`Sequence`	`∘`
`|`	`dk`	`+`	`Choice`	`|`
`or-else`	`first`	`<+`	Using Java	`Fst`
`match` P	Using rules or library	`?`P	`%match`	Using rules
α `?` β `:` γ	`if` α `then` β `orelse` γ `fi`	α `<` β `+` γ	Using Java	Using rules
`*`	`iterate*`			`*`
`+`	`iterate+`			
`!`	`repeat*`	`repeat`	`Repeat`	`NF`
`one(`α`)`	`dc/first one(`α`)`	Implicit	Implicit	
`try`	`first(`α`, id)`	`try`	`Try`	`Fst[`α`, Id]`
`test`	Using rules	`where`	Using Java	Using rules
$\alpha \equiv sl(t_1, \ldots, t_n)$	`call(`α`)`	`call(`$sl \mid\mid t_1, \ldots, t_n$`)`	α`.apply`	Using rules

programming languages. TOM is a rewriting extension of Java that allows defining signatures, rules, and strategies inside the Java code and interoperate with them. There are other more recent strategy languages, like ρLog (2004) [34] integrated into the Mathematica computing system as a plugin, and Porgy (2009) [24] for strategic graph (instead of term) rewriting.

All these languages have similar foundations and repertories of strategy combinators, as shown in Table 2. Most combinators of any of the languages are either available in the others or can otherwise be expressed with more elaborate expressions. However, we highlight the main differences regarding Maude:

1. Maude enforces a clear separation between rules and strategies, while in ELAN and TOM strategies can be used in the definition of rules and they are sometimes required to cooperate. This dependency is more explicit in ρLog, where strategies are defined using an extension of the syntax for defining rules. Stratego allows and promotes this separation, but also lets strategies directly set the subject term, define new rules, and apply inline ones. The separation between rules and strategies in Maude is a conscious design decision to ease the understanding, analysis, and verification of the specification, respecting the *separation of concerns* principle.
2. Strategies are potentially nondeterministic in all of these languages. However, ELAN, ρLog, and Maude compute the whole set of results for a given term, while Stratego and TOM explore a single rewriting path by resolving the nondeterminism arbitrarily.
3. The application of a rule is the common basic element of these languages. However, all except Maude apply rules at the top by default. Instead, Maude applies them on any subterm within the subject term, unless explicitly indicated by **`top`**.
4. Parameterization is available by means of modules in ELAN and Maude, or at the level of strategy definitions in ρLog and Stratego, whose approach is more succinct. In fact, strategy expressions in Stratego are usually simpler, since the variety of strategy combinators is wider, some combinators are generic, and the signatures of strategies are not declared (types are not checked).
5. Stratego, TOM, and ρLog allow programming generic traversals of terms, which need to be made explicit for each known operator in the signature in Maude. The generic possibilities of ρLog follow from the untyped Wolfram language of Mathematica on which it is based, since variables can be used not only for terms but also for function symbols and argument lists.
6. Maude and Stratego allow binding variables in strategy expressions. In Maude their scopes are delimited nested regions like the **`matchrew`** substrategies, while in Stratego explicit scopes can be declared and variables are defined from left to right within the same expression level.
7. ELAN, Stratego, and TOM include a library of reusable strategy definitions, while no such library currently exists for the Maude strategy language.

Some features, like the generic term traversals of Stratego (item 5), have not been incorporated into the Maude language for the sake of simplicity. This is also the case for *congruence operators* that make each symbol f in the signature a strategy combinator such that $f(\alpha_1, \ldots, \alpha_n)$ applies the given strategies to the corresponding subterms. In [51], these combinators have been implemented in Maude at the metalevel using **`matchrew`** operators, which can be seen as their counterparts in this language. Generic traversals have been addressed in [51] using a similar reflective strategy transformation, but we briefly describe here how they can be written by hand. Specifically, the three primitives or one-step descent operators of Stratego are `all`, to apply a strategy to all the direct subterms of the subject term; `some`, to apply a strategy to all the children in which it does not fail, and as long as it succeeds in at least one; and `one`, to apply the strategy to the first subterm in which it succeeds from left to right. These can be defined in the Maude strategy language for a fixed α with the following definition for each n-ary symbol f:

```
sd st_all := matchrew f(x1, ..., xn) by x1 using α, ...,
                                        xn using α .
sd st_one := matchrew f(x1, ..., xn) by x1 using α or-else
               ... or-else
             matchrew f(x1, ..., xn) by xn using α .
```

The `some` operator can be defined as `test(st_one(α)) ; st_all(try(α))` in pseudocode according to its definition. It can also be given its own definition using a sequence of **`matchrew`** that apply the strategy α on every argument, letting it fail in all but one argument each.

```
sd st_some := (matchrew f(x1, ..., xn) by x1 using α ;
                matchrew f(x1, ..., xn) by x2 using try(α),
                                           ..., xn using try(α))
              or-else ... or-else
              matchrew f(x1, ..., xn) by xn using α .
```

These definitions could be parametric on α by using parameterized strategy modules.

Similarly, the choosing operators from ELAN have not been included in the Maude language either, but almost all of them can be defined easily using Maude's strategy language constructs:

$$\begin{aligned}
\texttt{first}(\alpha_1, \ldots, \alpha_n) &\equiv \alpha_1 \,\texttt{or-else}\, \cdots \,\texttt{or-else}\, \alpha_n \\
\texttt{dk}(\alpha_1, \ldots, \alpha_n) &\equiv \alpha_1 \mid \cdots \mid \alpha_n \\
\texttt{first one}(\alpha_1, \ldots, \alpha_n) &\equiv \texttt{one}(\alpha_1) \,\texttt{or-else}\, \cdots \,\texttt{or-else}\, \texttt{one}(\alpha_n) \\
\texttt{dc one}(\alpha_1, \ldots, \alpha_n) &\equiv \texttt{one}(\alpha_1 \mid \cdots \mid \alpha_n)
\end{aligned}$$

The exception is `dc(`α_1`, ..., `α_n`)` that returns all the results of only one of the α_i chosen nondeterministically. With the **`one`** operator we are only able to take either one or all of the results of any strategy expression. In the opposite direction, there are Maude constructs that are not available in ELAN, but they can also be expressed by more involved strategy expressions, in the worst case resorting to rules.

8. Conclusion

The Maude strategy language is a useful resource to write and execute clear and natural strategy specifications where the description of rules is decoupled from the specification of how they should be applied, following the *separation of concerns* principle. While the Maude strategy language was proposed almost twenty years ago and its first prototype appeared soon after, these goals have not been completely achieved until the recent introduction of the language into the official Maude interpreter with support for first-class strategy modules and reflection. Moreover, the efficient implementation of the language at the C++ level makes more interesting and complex applications possible.

This article includes a comprehensive description of the current design of the strategy language, its complete formal semantics, and the relevant details about its implementation. We have shown that some challenges are involved in both the formal semantics and the implementation, and the benefit of this approach have been shown by various examples. Strategies have been used to specify Knuth-Bendix completion procedures on top of a well-established inference system, which helps reasoning about the correctness of the proposed algorithms. They have also been applied to solve games and to specify communication protocols. Additional examples presented in previous works include applications to the semantics of programming languages, process algebras, membrane systems, neural networks, linear programming, and a Sudoku solver, among others.

Once systems have been described in rewriting logic, verification and analysis become relevant. Testing and comparing specifications executed with alternative controlling strategies could provide interesting information, and verification techniques like model checking [47,52,50] have been developed for these systems. Quantitative extensions of the strategy language have also been implemented and applied for probabilistic and statistical model checking [46]. Moreover, support for strategies can be added to the Maude-based theorem provers and to other related formal tools [2].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

Martí-Oliet, Rubio, and Verdejo are partially supported by Spanish AEI project ProCode (PID2019-108528RB-C22/AEI/10.13039/50110001103). Rubén Rubio has been partially supported by the Spanish Ministry of Universities under grant FPU17/02319.

References

[1] Samson Abramsky, Domain Theory, vol. 3, Clarendon Press, 1994, pp. 1–168.
[2] María Alpuente, Demis Ballis, Santiago Escobar, D. Galán, Julia Sapiña, Safety enforcement via programmable strategies in Maude, J. Log. Algebraic Methods Program. 132 (2023) 100849, https://doi.org/10.1016/j.jlamp.2023.100849.
[3] Andrei Oana, Dorel Lucanu, Strategy-based proof calculus for membrane systems, in: Grigore Roşu (Ed.), Proceedings of the Seventh International Workshop on Rewriting Logic and Its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008, in: Electronic Notes in Theoretical Computer Science, vol. 238(3), Elsevier, 2009, pp. 23–43.
[4] Franz Baader, Tobias Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[5] Leo Bachmair, Nachum Dershowitz, Equational inference, canonical proofs, and proof orderings, J. ACM 41 (2) (1994) 236–276, https://doi.org/10.1145/174652.174655.
[6] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, Antoine Reilles, Tom: piggybacking rewriting on Java, in: Franz Baader (Ed.), Term Rewriting and Applications, 18th International Conference, RTA 2007, Paris, France, June 26-28, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4533, Springer, 2007, pp. 36–47.
[7] H.P. Barendregt, The Lambda Calculus: Its Syntax and Semantics, 2 ed., Studies in Logic and the Foundations of Mathematics, vol. 131, North Holland, 2014.
[8] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Christophe Ringeissen, Rewriting with strategies in ELAN: a functional semantics, Int. J. Found. Comput. Sci. 12 (1) (2001) 69–95, https://doi.org/10.1142/S0129054101000412.
[9] Adel Bouhoula, Jean-Pierre Jouannaud, José Meseguer, Specification and proof in membership equational logic, in: Michel Bidoit, Max Dauchet (Eds.), TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, April 14–18, 1997, Proceedings, in: Lecture Notes in Computer Science, vol. 1214, Springer, 1997, pp. 67–92.
[10] Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty, Hélène Kirchner, Extensional and intensional strategies, in: Maribel Fernández (Ed.), Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009, in: Electronic Proceedings in Theoretical Computer Science, vol. 15, 2009, pp. 1–19.
[11] Christiano Braga, Alberto Verdejo, Modular structural operational semantics with strategies, in: Rob van Glabbeek, Peter D. Mosses (Eds.), Proceedings of the Third Workshop on Structural Operational Semantics, SOS 2006, Bonn, Germany, August 26, 2006, in: Electronic Notes in Theoretical Computer Science, vol. 175(1), Elsevier, 2007, pp. 3–17.
[12] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, Eelco Visser, Stratego/XT 0.17. A language and toolset for program transformation, Sci. Comput. Program. 72 (1–2) (2008) 52–70, https://doi.org/10.1016/j.scico.2007.11.003.
[13] Horatiu Cirstea, Sergueï Lenglet, Pierre-Etienne Moreau, Faithful (meta-)encodings of programmable strategies into term rewriting systems, Log. Methods Comput. Sci. 13 (2017) 4.
[14] Manuel Clavel, Strategies and user interfaces in Maude at work, in: Bernhard Gramlich, Salvador Lucas (Eds.), Proceedings of the 3rd International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2003, Valencia, Spain, June 8, 2003, in: Electronic Notes in Theoretical Computer Science, vol. 86(4), Elsevier, 2003, pp. 570–592.
[15] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, Carolyn Talcott, Maude manual v3.3.1, https://maude.lcc.uma.es/maude-manual.
[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Carolyn L. Talcott, All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350, Springer, 2007.
[17] Manuel Clavel, José Meseguer, Reflection and strategies in rewriting logic, in: José Meseguer (Ed.), Proceedings of the First International Workshop on Rewriting Logic and Its Applications, WRLA'96, Asilomar, California, September 3–6, 1996, in: Electronic Notes in Theoretical Computer Science, vol. 4, Elsevier, 1996, pp. 126–148.
[18] Manuel Clavel, José Meseguer, Internal strategies in a reflective logic, in: Bernhard Gramlich, Hélène Kirchner (Eds.), Proceedings of the CADE-14 Workshop on Strategies in Automated Deduction, Townsville, Australia, 1997, pp. 1–12.
[19] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, Carolyn Talcott, Programming and symbolic computation in Maude, J. Log. Algebraic Methods Program. 110 (2020) 58, https://doi.org/10.1016/j.jlamp.2019.100497.
[20] Francisco Durán, José Meseguer, On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories, J. Log. Algebraic Methods Program. 81 (7–8) (2012) 816–850, https://doi.org/10.1016/j.jlap.2011.12.004.
[21] Francisco Durán, Camilo Rocha, José María Álvarez, Towards a Maude formal environment, in: Gul Agha, Olivier Danvy, José Meseguer (Eds.), Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday, in: Lecture Notes in Computer Science, vol. 7000, Springer, 2011, pp. 329–351.
[22] Steven Eker, Narciso Martí-Oliet, José Meseguer, Isabel Pita, Rubén Rubio, Alberto Verdejo, Strategy language for Maude, https://maude.ucm.es/strategies.
[23] Steven Eker, Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, Deduction, strategies, and rewriting, in: Myla Archer, Thierry Boy de la Tour, César Muñoz (Eds.), Proceedings of the 6th International Workshop on Strategies in Automated Deduction, STRATEGIES 2006, Seattle, WA, USA, August 16, 2006, in: Electronic Notes in Theoretical Computer Science, vol. 174(11), Elsevier, 2007, pp. 3–25.
[24] Maribel Fernández, Hélène Kirchner, Bruno Pinaud, Strategic port graph rewriting: an interactive modelling framework, Math. Struct. Comput. Sci. 29 (5) (2019) 615–662, https://doi.org/10.1017/S0960129518000270.
[25] Joseph A. Goguen, Parameterized programming, IEEE Trans. Softw. Eng. 10 (5) (1984) 528–544, https://doi.org/10.1109/TSE.1984.5010277.
[26] Mercedes Hidalgo-Herrero, Alberto Verdejo, Yolanda Ortega-Mallén, Using Maude and its strategies for defining a framework for analyzing Eden semantics, in: Sergio Antoy (Ed.), Proceedings of the Sixth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2006, Seattle, WA, USA, August 11, 2006, in: Electronic Notes in Theoretical Computer Science, vol. 174(10), Elsevier, 2007, pp. 119–137.
[27] Gérard P. Huet, A complete proof of correctness of the Knuth-Bendix completion algorithm, J. Comput. Syst. Sci. 23 (1) (1981) 11–21, https://doi.org/10.1016/0022-0000(81)90002-7.
[28] Hélène Kirchner, Rewriting strategies and strategic rewrite programs, in: Narciso Martí-Oliet, Peter Csaba Ölveczky, Carolyn L. Talcott (Eds.), Logic, Rewriting, and Concurrency - Essays Dedicated to José Meseguer on the Occasion of His 65th Birthday, in: Lecture Notes in Computer Science, vol. 9200, Springer, 2015, pp. 380–403.
[29] Hélène Kirchner, Pierre-Etienne Moreau, Prototyping completion with constraints using computational systems, in: Jieh Hsiang (Ed.), Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 5–7, 1995, Proceedings, in: Lecture Notes in Computer Science, vol. 914, Springer, 1995, pp. 438–443.
[30] Donald E. Knuth, Peter B. Bendix, Simple word problems in universal algebras, in: John Leech (Ed.), Computational Problems in Abstract Algebra. Proceedings of a Conference Held at Oxford Under the Auspices of the Science Research Council Atlas Computer Laboratory, 29th August to 2nd September 1967, Pergamon Press, 1970, pp. 263–297.
[31] Robert A. Kowalski, Algorithm = logic + control, Commun. ACM 22 (7) (1979) 424–436, https://doi.org/10.1145/359131.359136.

[32] Pierre Lescanne, Implementations of completion by transition rules + control: ORME, in: Hélène Kirchner, Wolfgang Wechler (Eds.), Algebraic and Logic Programming, Second International Conference, Nancy, France, October 1–3, 1990, Proceedings, in: Lecture Notes in Computer Science, vol. 463, Springer, 1990, pp. 262–269.
[33] G. Malkin, RIP Version 2, RFC 2453, Internet Engineering Task Force, 1998, https://tools.ietf.org/html/rfc2453.
[34] Mircea Marin, Temur Kutsia, Foundations of the rule-based system ρLog, J. Appl. Non-Class. Log. 16 (1–2) (2006) 151–168, https://doi.org/10.3166/jancl.16.151-168.
[35] Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, Towards a strategy language for Maude, in: Narciso Martí-Oliet (Ed.), Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004, in: Electronic Notes in Theoretical Computer Science, vol. 117, Elsevier, 2004, pp. 417–441.
[36] Narciso Martí-Oliet, José Meseguer, Alberto Verdejo, A rewriting semantics for Maude strategies, in: Grigore Roşu (Ed.), Proceedings of the Seventh International Workshop on Rewriting Logic and Its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008, in: Electronic Notes in Theoretical Computer Science, vol. 238(3), Elsevier, 2009, pp. 227–247.
[37] Narciso Martí-Oliet, Miguel Palomino, Alberto Verdejo, Strategies and simulations in a semantic framework, J. Algorithms 62 (3–4) (2007) 95–116, https://doi.org/10.1016/j.jalgor.2007.04.002.
[38] José Meseguer, Conditional rewriting logic as a unified model of concurrency, Theor. Comput. Sci. 96 (1) (1992) 73–155, https://doi.org/10.1016/0304-3975(92)90182-F.
[39] José Meseguer, Twenty years of rewriting logic, J. Log. Algebraic Program. 81 (7–8) (2012) 721–781, https://doi.org/10.1016/j.jlap.2012.06.003.
[40] Pierre-Etienne Moreau, REM (Reduce Elan Machine): core of the new ELAN compiler, in: Leo Bachmair (Ed.), Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10-12, 2000, Proceedings, in: Lecture Notes in Computer Science, vol. 1833, Springer, 2000, pp. 265–269.
[41] Alberto Pettorossi, Maurizio Proietti, Program derivation = rules + strategies, in: Antonis C. Kakas, Fariba Sadri (Eds.), Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part I, in: Lecture Notes in Computer Science, vol. 2407, Springer, 2002, pp. 273–309.
[42] Gordon D. Plotkin, A powerdomain construction, SIAM J. Comput. 5 (3) (1976) 452–487, https://doi.org/10.1137/0205035.
[43] Daniel Ratner, Manfred K. Warmuth, The $(n^2 - 1)$-puzzle and related relocation problems, J. Symb. Comput. 10 (2) (1990) 111–138, https://doi.org/10.1016/S0747-7171(08)80001-6.
[44] Fernando Rosa-Velardo, Clara Segura, Alberto Verdejo, Typed mobile ambients in Maude, in: Horatiu Cirstea, Narciso Martí-Oliet (Eds.), Proceedings of the 6th International Workshop on Rule-Based Programming, RULE 2005, Nara, Japan, April 23, 2005, in: Electronic Notes in Theoretical Computer Science, vol. 147(1), Elsevier, 2006, pp. 135–161.
[45] Rubén Rubio, Model checking of strategy-controlled systems in rewriting logic, Ph.D. Dissertation, Universidad Complutense de Madrid, 2022, https://eprints.ucm.es/71531.
[46] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, QMaude: quantitative specification and verification in rewriting logic, in: Marsha Chechik, Joost-Pieter Katoen, Martin Leucker (Eds.), Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings, in: Lecture Notes in Computer Science, vol. 14000, Springer, 2023, pp. 240–259.
[47] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Model checking strategy-controlled rewriting systems, in: Herman Geuvers (Ed.), 4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany, in: LIPIcs, vol. 131, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 34:1–34:18.
[48] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Parameterized strategies specification in Maude, in: José Fiadeiro, Ionuț Țuțu (Eds.), Recent Trends in Algebraic Development Techniques, in: Lecture Notes in Computer Science, vol. 11563, Springer, 2019, pp. 27–44.
[49] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, 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/.
[50] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Strategies, model checking and branching-time properties in Maude, J. Log. Algebraic Methods Program. 123 (2021) 28, https://doi.org/10.1016/j.jlamp.2021.100700.
[51] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Metalevel transformation of strategies, J. Log. Algebraic Methods Program. 124 (2022) 21, https://doi.org/10.1016/j.jlamp.2021.100728.
[52] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Model checking strategy-controlled systems in rewriting logic, Autom. Softw. Eng. 29 (1) (2022) 57, https://doi.org/10.1007/s10515-021-00307-9.
[53] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo, Simulating and model checking membrane systems using strategies in Maude, J. Log. Algebraic Methods Program. 124 (2022) 25, https://doi.org/10.1016/j.jlamp.2021.100727.
[54] Gustavo Santos-García, Miguel Palomino, Solving Sudoku puzzles with rewriting rules, in: Grit Denker, Carolyn Talcott (Eds.), Proceedings of the 6th International Workshop on Rewriting Logic and Its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006, in: Electronic Notes in Theoretical Computer Science, vol. 176(4), Elsevier, 2007, pp. 79–93.
[55] Gustavo Santos-García, Miguel Palomino, Alberto Verdejo, Rewriting logic using strategies for neural networks: an implementation in Maude, in: Juan M. Corchado, Sara Rodríguez, James Llinas, José M. Molina (Eds.), International Symposium on Distributed Computing and Artificial Intelligence, DCAI 2008, University of Salamanca, Spain, 22th-24th October 2008, in: Advances in Soft Computing, vol. 50, Springer, 2009, pp. 424–433.
[56] Terese, Term Rewriting Systems, Cambridge University Press, 2003.
[57] Alberto Verdejo, Narciso Martí-Oliet, Basic completion strategies as another application of the Maude strategy language, in: Santiago Escobar (Ed.), Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011, in: Electronic Proceedings in Theoretical Computer Science, vol. 82, 2011, pp. 17–36.
[58] Eelco Visser, A survey of strategies in rule-based program transformation systems, J. Symb. Comput. 40 (1) (2005) 831–873, https://doi.org/10.1016/j.jsc.2004.12.011.
[59] Marian Vittek, A compiler for nondeterministic term rewriting systems, in: Harald Ganzinger (Ed.), Rewriting Techniques and Applications, 7th International Conference, RTA-96, New Brunswick, NJ, USA, July 27–30, 1996, Proceedings, in: Lecture Notes in Computer Science, vol. 1103, Springer, 1996, pp. 154–167.
[60] Guido Wachsmuth, Gabriël D.P. Konat, Eelco Visser, Language design with the Spoofax language workbench, IEEE Softw. 31 (5) (2014) 35–43, https://doi.org/10.1109/MS.2014.100.
[61] Éduard Lucas, Recréations mathématiques, 2 ed., Albert Blanchard, Paris, 1992.