Metalevel transformation of strategies *

Rubén Rubio, Narciso Martí-Oliet, Isabel Pita, Alberto Verdejo
Universidad Complutense de Madrid, Spain
{rubenrub,narciso,ipandreu,jalberto}@ucm.es

The Maude strategy language was introduced to avoid using complex and verbose metalevel programs to control rule application. However, just as multiple levels of reflection are required for some tasks, reflective manipulation and generation of strategies are convenient in multiple situations. Some examples of reflective strategy transformations are presented, which implement special forms of evaluation or extend the strategy language while preserving its advantages.

1 Introduction

Rewriting logic and its implementation Maude are reflective languages where important aspects of its own metatheory can be represented. Hence, manipulating, transforming, and analyzing rewriting logic theories specified in Maude can be easily done within Maude. This feature has been extensively used, for example, in Full Maude [4, Part II], a language extension written in Maude itself and whose additional constructs and commands are finally translated to the Core Maude language and executed by its interpreter, and the Maude Formal Environment [8] to check properties like confluence and termination on Maude specifications.

These reflective features have also been used to control rewriting. By default, the executions of a rewriting system are sequences of independent rule applications where the next rule and position are chosen nondeterministically. Sometimes, it is convenient to consider a limited subset of behaviors and express them at a higher level, without modifying the base system. This is the purpose of *rewriting strategies* [2], expressed in Maude since its beginnings using its reflective features. Since programming metalevel computations is hard for beginners and verbose, an object-level strategy language has been proposed, tested, and finally made available in Maude 3.0 [4]. Although the strategy language has been introduced to avoid the need for the metalevel, the language itself and its operations have been metarepresented, and users may still resort to the metalevel to analyze strategy specifications and construct strategies depending on metatheoretic aspects. This move to the metalevel does not limit the interaction at the object-level and the usage of verification tools like the model checker for systems controlled by strategies [13].

After reviewing the basics of rewriting and Maude, including its reflective features and its strategy language, this paper presents three examples of metalevel transformations generating strategies to solve specific problems. The first example is related to context-sensitive rewriting, the second one is an extension of the Maude strategy language with additional constructs such as congruence operators and generic traversals, and the third is a framework to specify compositional or agent-based strategy-controlled systems. Maude 3.0 can be downloaded from `maude.cs.illinois.edu` and its extension with the strategy-aware model checker is available at `maude.ucm.es/strategies`, as well as the different examples appearing here, for which a direct hyperlink is provided next to each section title.

* Research partially supported by MCI Spanish project *TRACES* (TIN2015-67522-C3-3-R). Rubén Rubio is partially supported by MU grant FPU17/02319.

2 Rewriting logic and Maude

Rewriting logic [11] was proposed in 1992 as a unified model of concurrency extending membership equational logic with nondeterministic and possibly conditional rewriting rules. A rewrite theory $\mathscr{R} = (\Sigma, E, R)$ consists of a signature Σ of order-sorted operators, a set of equations E, and a set of rewriting rules R. Terms are considered modulo equations and also structural axioms like commutativity, associativity, and identity that cannot be naively handled as regular equations due to its reversible nature. Maude [4] is a specification language based on rewriting logic, where rewrite systems can be specified compositionally, executed, and analyzed. Specifications are written in a mathematical-like notation and organized in modules of different kinds: functional modules (`fmod`) represent equational theories with `sort`, `subsort`, and `op` declarations, and equations of the form **`eq`** *`l = r .`*; system modules (`mod`) are complete rewrite theories with the addition of rules **`rl`** *`l => r .`*; and now strategy modules (`smod`) specify alternative ways of applying these rules using strategies. Maude specifications are executable under certain requirements [4] and the Maude system offers several commands to `reduce` terms equationally, to `rewrite` a term with the rules modulo equations and axioms, to `search` the rewriting graph, etc. Moreover, it includes an LTL model checker [4, §12].

2.1 Reflection and metalevel computations

Rewriting logic is a reflective logic, whose objects and operations can be consistently represented in itself. Maude offers a predefined *universal theory* [4, §17] to metatheoretically represent terms, equations, rules, modules, and so on. Operations like matching, reduction, and rule application can be programmed generically using regular operators and equations, but Maude provides special operators backed by the object-level implementation in C++ to allow efficient reflective computations. Metarepresentations can in turn be metarepresented and terms be moved between different levels, yielding arbitrarily high reflective towers.

This universal theory is specified in the `META-LEVEL` and its imported modules, and it relies on the `Qid` sort of *quoted identifiers*, arbitrary words prefixed by an apostrophe. A variable `X` of sort `Nat` is metarepresented as the quoted identifier `'X:Nat`, and the constant `'Nat` of sort `Qid` is `''Nat.Qid`. Terms with arguments are represented using the operator `_[_] : Qid NeTermList -> Term`, like `'_+_['X:Nat, 's_['0.Zero]]` for `X + s 0`. Operator declarations, equations, rules, and similar statements are represented as terms with a syntax similar to the object-level reference. For example, the operator `+` may have a declaration `op '_+_ : 'Nat 'Nat -> 'Nat [comm assoc] .` and be involved in an equation `eq '_+_['X:Nat, '0.Zero] = 'X:Nat [none] .` where the trailing brackets enclose the set of operator or statement attributes. Metamodules are terms with argument slots like `fmod_is_sorts_.____endfm` for each kind of module member. Auxiliary functions `getOps`, `getEqs`, `getRls`, etc., are defined to obtain these components.

Operations are accessible through some *descent functions* like `metaMatch` for matching, `metaApply` for rule application, `metaReduce` for equational reduction, `metaRewrite`, etc. For instance, `metaReduce` receives the metarepresentations of a module and a term, and produces a pair containing the reduced term and its calculated sort. The complete specification of the metalevel is in the Maude prelude and explained in [4, §17].

2.2 The Maude strategy language

Strategies have been specified since the beginnings of Maude using the reflective features explained in the previous section, but to make strategy specification more accessible and understandable, an object-level strategy language was proposed, prototyped using Full Maude, tested, and finally implemented at the C++ level in Maude 3.0, with new features like compositional and parameterized strategy modules [14]. Its design is inspired on other strategy languages like ELAN [1] and Stratego [3]. A strategy expression α restricts the possible next steps during the rewriting process and can be described as a transformation from an initial term t to the set of terms that this controlled but not necessarily deterministic rewriting yields as a result. This is what the command `srewrite` t `using` α and its depth-first version `dsrewrite` show, by exploring all allowed execution paths.

The application of a rule $rl\,[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n]$ is the basic element of the language, referred by its label rl and taking an optional initial substitution. Tests `match` P `s.t.` C discard executions when the subject term does not match P or satisfy the condition C. The `match` keyword can be changed to `amatch` to match anywhere within the term. These elements can be combined with the concatenation $\alpha\,;\beta$ that executes β on the results of α, the disjunction $\alpha\,|\,\beta$ that includes the executions allowed by any of its arguments, the iteration $\alpha*$ and normalization $\alpha!$ operators that iterate α any number of times or until no more iterations are possible, and the conditional $\alpha?\beta:\gamma$ that evaluates α and then β on its results, but if α does not produce any, it executes γ on the initial term. Two constants `idle` and `fail` represent the strategy that produces the initial term as result and the strategy that does not produce any result. The last combinator allows rewriting selected subterms `matchrew` P `s.t.` x_1 `using` α_1, ..., x_n `using` α_n: the terms matched by the variables $x_1, \ldots, x_n$ in the pattern are rewritten in parallel using $\alpha_1, \ldots, \alpha_n$, respectively, and their results are combined to produce the global results. Moreover, named strategies taking arguments can be declared as **strat** *name* : $s_1 \cdots s_n$ `@` s . with its signature, defined **sd** *name*`(`p_1`,` ..., p_n`)` `:=` α ., and called *name*`(`t_1`,` $\cdots$`,` t_n`)` even recursively. More details can be found in [4, §10] and examples are shown in the following sections.

The strategy language and strategy modules are also represented at the metalevel, faithfully reproducing the object-level syntax in most cases. Its combinators are specified as terms of the `Strategy` sort in the `META-STRATEGY` module. For instance, a simple rule application is written `'label[none]{empty}` and a strategy call `'name[[`*TL*`]]` with *TL* a possibly `empty` list of metarepresented terms. Strategy modules `smod_is_sorts_._____endsm` as well as strategy declarations and definitions are specified too, and the commands `srewrite` and `dsrewrite` are accessible through the `metaSrewrite` descent function. Notice that previous prototypes of the strategy language were specified within Maude, so strategy expressions were Maude terms that can be directly manipulated at its object-level or at its metalevel if necessary. These prototypes were more easily extensible, since the execution of strategies was implemented in Maude itself, at the expense of efficiency.

2.3 Interactive interfaces

Writing interactive interfaces in Maude is relatively easy, and it is usually done to offer a convenient interface to the logic and semantic frameworks specified in the language. The archetype is Full Maude [4, §15], an extended interpreter written in Maude where many features implemented in C++ have been first tested. The functionality of the Core Maude interpreter is replicated there along with additional features like tuple types and object-oriented modules. Users can also extend Full Maude to include their own features and commands. Moreover, since Maude 3.0 [5], the interactive capabilities of Maude have increased due to new external objects that allow reading and writing files as well as the standard input and

output streams.

3 An introductory example

Context-sensitive rewriting [10] is a restricted form of term rewriting defined by simple constraints attached to the symbols of the signature. Maude has builtin support for this kind of restrictions, but their direct application is not enough to obtain real normal forms, as we will see with a lazy programming example, for which strategies are needed. Generating these strategies from the context-sensitive restrictions will be the purpose of our first metalevel transformation. Let us introduce first the following functional module [6] that attempts to specify a lazy integer list:

```
fmod LAZY-LIST is
  protecting INT .
  sort LazyList .

  op nil : -> LazyList [ctor] .
  op _:_ : Int LazyList -> LazyList [ctor] .

  var E : Int . var N : Nat . var L : LazyList .

  op take : Nat LazyList -> LazyList .
  eq take(0, L) = nil .
  eq take(s(N), E : L) = E : take(N, L) .

  op natsFrom : Nat -> LazyList .
  eq natsFrom(N) = N : natsFrom(N + 1) .
endfm
```

Even though `natsFrom(`n`)` represents an infinite list, containing all natural numbers from n, we would expect that the lazy evaluation of a term like `take(3, natsFrom(0))` leads to `0:1:2:nil`. Maude's `reduce` command is eager, so the evaluation of this term will not terminate because of the continuous reduction of the second argument of the list in the `natsFrom` definition. Fortunately, Maude allows appending some attributes to operator declarations that exclude arguments from rewriting: regarding equations, the attribute `strat` restricts reduction to the arguments whose indices are provided as a zero-terminated list; regarding rules, the attribute `frozen` inhibits rewriting with rules inside a given subset of arguments. For example, the attributes of the `_:_` operator can be changed to `[ctor strat (1 0)]` to avoid reducing inside the second argument. However, this still does not produce a valid result.

```
Maude> reduce take(3, natsFrom(0)) .
rewrites: 2
result LazyList: 0 : take(2, natsFrom(0 + 1))
```

In the vocabulary of context-sensitive rewriting, `strat` and `frozen` annotations correspond to *replacement maps* $\mu : \Sigma \to \mathscr{P}(\mathbb{N})$ where $\mu(f) \subseteq \{1,\ldots,\mathrm{ar}(f)\}$ for all $f \in \Sigma$. Reduction is only allowed in the *μ-replacing positions* of any term t, defined recursively as the top position and every μ-replacing position of each t_i if $t = f(t_1,\ldots,t_n)$ and $i \in \mu(f)$. Exhaustively reducing in these positions leads to *μ-normal forms*, and this is exactly what the previous command did for `take(3, natsFrom(0))` and $\mu(\_:\_) = \{1\}$. As we have seen, μ-normal forms are not necessarily actual normal forms, but they can be useful as part of complete and lazy normalization procedures. Among the different proposed approaches to *normalization via μ-normalization* [6], we will implement a *layered evaluation* that safely resumes reduction on

the subterms of non-replacing positions. This will be achieved by means of a generated signature-aware strategy, proposed by Salvador Lucas.

The following function `csrTransform` implements a metalevel module transformation that extends the metarepresentation of the input module `M` with strategy declarations and definitions to normalize terms as described in the previous paragraph. Its global shape is given by the following equation[1].

```
op csrTransform : Module -> StratModule .
eq csrTransform(M) = smod getName(M) is
  getImports(M)                                        *** module importation
  sorts 'AnyTerm ; getSorts(M) .                             *** sort decls
  getSubsorts(M)                                          *** subsort decls
  strat2frozen(getOps(M))                                 *** operator decls
  getMbs(M)                                          *** sort membership axioms
  none                                                        *** equations
  getRls(M)                                                       *** rules
  eqs2rls(getEqs(M))
  getStrats(M)                                            *** strategy decls
  (strat 'norm-via-munorm : nil @ 'AnyTerm [none] .)
  (strat 'munorm : nil @ 'AnyTerm [none] .)
  (strat 'decomp : nil @ 'AnyTerm [none] .)
  getSds(M)                                          *** strategy definitions
  (sd 'norm-via-munorm[[empty]] :=
       'munorm[[empty]] ; try('decomp[[empty]]) [none] .)
  (sd 'munorm[[empty]] := one(all) ! [none] .)
  (sd 'decomp[[empty]] := makeDecomp(getOps(M)) [none] .)
endsm .
```

Apart from the new strategies, the transformed module is essentially a copy of the original one. However, since the Maude strategy language can only control rule application, we translate all equations to rules, and all `strat` attributes to `frozen` annotations.

```
eq eqs2rls(none) = none .
eq eqs2rls(eq L = R [Attrs] . Eqs) = rl L => R [Attrs] . eqs2rls(Eqs) .
```

The entry point for the layered normalization strategy is `norm-via-munorm`, which executes two auxiliary strategies `munorm` for μ-normalization, and then `decomp` for resuming normalization inside frozen arguments. `munorm` is implemented by exhaustively (`!`) applying the rules in the module respecting the `frozen` restrictions (`all`). Assuming the input system is μ-confluent, i.e. under the context-sensitive restrictions, the order in which rules are applied does not affect the result, so `all` is executed for efficiency under the `one` operator that discards alternative rewrite orders. The `decomp` strategy continues normalization on the symbol arguments and consists of the disjunction of `matchrew` combinators for each $f \in \Sigma$ that apply `norm-via-munorm` to each subterm:

$$\texttt{matchrew } f(x_1, \ldots, x_n) \texttt{ by } \ldots, x_i \texttt{ using norm-via-munorm, } \ldots$$

Since they depend on the signature of the module, `decomp` is reflectively generated by the `makeDecomp` function that walks through the operators declared in the module. Some auxiliary functions like `makeVar` and `makeVarList` are used to generate sequentially-numbered variable metarepresentations of the given sorts.

[1] Strategy declarations must include the intended sort to which they will be applied. However, the strategies defined here are somehow polymorphic, so we declare `AnyTerm` just to take its place.

```
op makeDecomp : OpDeclSet -> Strategy .
eq makeDecomp(none) = fail .
eq makeDecomp(op Q : nil -> Ty [Attrs] . Ops) = makeDecomp(Ops) .
eq makeDecomp(op Q : NeTyL -> Ty [Attrs] . Ops) =
  (matchrew Q[makeVarList(NeTyL, 1)] s.t. nil
    by makeUsingPart(NeTyL, 1)) | makeDecomp(Ops) .

op makeUsingPart : NeTypeList Nat -> UsingPairSet .
eq makeUsingPart(Ty, N) = makeVar(N, Ty) using 'norm-via-munorm[[empty]] .
eq makeUsingPart(Ty NeTyL, N) = makeUsingPart(Ty, N),
                                makeUsingPart(NeTyL, s(N)) .
```

Since no `matchrew` is generated for constants, the strategy `decomp` would fail when applied to one, so this strategy is surrounded by a $\mathtt{try}(\alpha) \equiv \alpha \ ?\ \mathtt{idle} : \mathtt{idle}$ combinator to avoid it.

Finally, the term `csrTransform(upModule('LAZY-LIST, true))` can be reduced to obtain the transformed `'LAZY-LIST` module, where `upModule` evaluates to the metarepresentation of the module whose name is given. Then, the `norm-via-munorm` strategy can be applied to a term using the `metaSrewrite` function or using Full Maude[2]:

```
(select CSR-TRANSFORM .)
(load csrTransform(upModule('LAZY-LIST, true)) .)
(select LAZY-LIST .)
(srewrite take(3, natsFrom(0)) using norm-via-munorm .)
srewrite in LAZY-LIST : take(3, natsFrom(0)) using norm-via-munorm .

Solution 1
result LazyList: 0 : 1 : 2 : nil

No more solutions
```

The evaluation now terminates and its result is meaningful. The generated `norm-via-munorm` can be applied to any term in this module, but only in this module since it explicitly refers to its signature. The following example shows an extension of the Maude strategy language adding new combinators based on the subject module in which strategies are to be applied.

4 Theory-dependent extensions of the strategy language

When the Maude strategy language was designed, the objective was not to include a vast repertory of operators to concisely express a wide range of tasks, like in the case of Stratego [3], but to be compact and expressive enough. Thanks to the reflectivity, the language can be extended to better suit a specific purpose or to incorporate a missing feature. In this section, we will apply this postulate and extend the language with the so-called *congruence operators* from ELAN [1] and Stratego, and the *generic traversals* from Stratego. Both operator families depend on the signature of the subject module where strategies are applied, so they will be implemented as module transformations. For the extension to be useful, the user should not be forced to renounce to the advantages of the strategy language, so we provide the means to use extended strategies at the object level and with the strategy-aware model checker.

[2] Full Maude commands are typed between parentheses, once the `full-maude.maude` file is loaded. Its last version can be downloaded from `maude.cs.illinois.edu`.

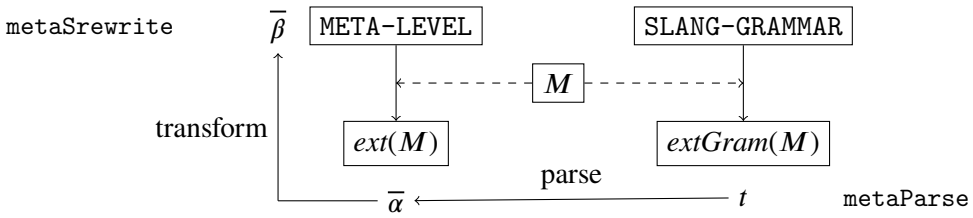

Figure 1: Typical structure of a strategy language extension

Congruence operators are strategy combinators that reproduce the data operators of the target module with their arguments replaced by strategies. The strategies in the congruence operator are applied to the corresponding arguments of the subject term's top symbol if they coincide, i.e. they are semantically equivalent to a `matchrew` construct of the form

$$f(\alpha_1, \ldots, \alpha_n) \equiv \texttt{matchrew } f(x_1, \ldots, x_n) \texttt{ by } x_1 \texttt{ using } \alpha_1, \ldots, x_n \texttt{ using } \alpha_n.$$

On the other hand, generic traversals are operators that allow applying a strategy along the structure of any term without explicitly mentioning it: `all(`α`)` applies α to all arguments of the top symbol, `one(`α`)` applies α to the first argument from left to right in which it succeeds[3], and `some(`α`)` is equivalent to `all(try(`α`))`. For example, the `decomp` strategy in Section 3 can be defined using generic traversals as simply `all(norm-via-munorm)`.

The approach used to implement them, which is directly applicable to other extensions, consists of the steps illustrated in Figure 1. Given a module M, the predefined module `META-LEVEL` is extended like in Section 3 with the metarepresentations of the congruence operators of sort `Strategy`:

```
op generateCongOps : OpDeclSet -> OpDeclSet .
eq generateCongOps(none) = none .
eq generateCongOps(op Q : TyL -> Ty [ctor Attrs] . Ops) =
  (op Q : repeatType('Strategy, size(TyL))
          -> 'Strategy [ctor removeId(Attrs)] .) generateCongOps(Ops) .
eq generateCongOps(Op Ops) = generateCongOps(Ops) [owise] .
```

However, the builtin `metaSrewrite` does not know these new operators so it will be unable to execute them. We should then implement their semantics from scratch or translate them to the standard language. The second option is simpler and automatically allows using all the strategy-related machinery of the interpreter with extended strategies, including the model checker. This translation is defined in the extended `META-LEVEL` as a function `transform` between terms of sort `Strategy`. Notice that the equations defining `transform` are generated by the module transformation, so they must metarepresent `Strategy` terms and involve two levels of reflection. The complete example is available in the strategy language example collection [9].

Since the strategy language does not provide means to perform generic traversals of terms and since we have chosen to translate extended strategies to standard ones, we should implement generic traversals using module-specific strategies. Namely, we can translate the strategy `all(`α`)` via the disjunction of $f(\alpha, \ldots, \alpha)$ for all $f \in \Sigma$, and `one(`α`)` using the disjunction for all f of

$$f(\alpha, \texttt{ idle}, \ldots, \texttt{idle}) \texttt{ or-else } \cdots \texttt{ or-else } f(\texttt{idle}, \texttt{ idle}, \ldots, \alpha).$$

[3]Not to be confused with the `one` and `all` operators of the strategy language. In the implementation, the generic traversal operators are renamed to `gt-all`, `gt-one`, and `gt-some`.

Finally, we want to write and use extended strategies at the object level. In order to do so, an extensible grammar of the strategy language, named `SLANG-GRAMMAR` in Figure 1, is defined along with a metalevel extension function that incorporates new productions for the extended strategies. A `parse` function is also provided to convert the term parsed by the predefined parsing function `metaParse` into the metarepresentation of extended strategies. All these components have been programmed generically, so that they can be easily reused for other language extensions. An interactive interface based on the external objects of Maude 3.0 is available to experiment with the extensions [9]. For example, in a module with rules **rl** `[swap] : f(X, Y) => f(Y, X) .` and **rl** `[next] : a => b .`, the extended strategy `f(swap, gt-all(next))` can be executed:

```
SLExt> select EXAMPLE .
Module EXAMPLE is now the current module.
SLExt> srew f(f(a, b), f(a, a)) using f(swap, gt-all(next)) .
Solution 1:     f(f(b, a), f(b, b))
No more solutions.
```

5 Multistrategies

The strategy-controlled system model proposed in Maude is the combination of a rewrite system and a strategy expression that controls this system as a whole. However, many systems are better specified compositionally. A typical example are object- or agent-oriented systems, in which each object or agent would follow its own strategy. Likewise, describing the interaction of players in games with a single sequential strategy control flow is cumbersome. Hence, we propose the following model transformation to facilitate this specification problem. Instead of a single strategy expression α, the system control will be specified by a *multistrategy*: an undetermined number of strategies $\alpha_1, \ldots, \alpha_n$ and a global strategy γ that describes how they are combined. Two builtin γ are provided: a concurrent one, in which the next strategy to take a step can be any of them, and a turn-based one, in which strategies are executed in a fixed order. A fundamental question is which are the atomic steps of the α_i strategies that are interleaved in the global execution, i.e. its granularity. As in the model checker [13], rule application is considered the main atomic step, but a few more strategies are executed atomically like `matchrew`s with a non-trivial pattern and conditions in the conditional operator, since they assume a particular structure or invariant of the term that may not be preserved if another strategy *thread* modifies the term during the while.

Multistrategies are implemented using strategies at the metalevel and an augmented execution environment. Essentially, to evaluate the strategies α_1, ..., α_n on the subject term t, they are transformed into the term $\{\ \bar{t}\ ::\ <\ 1\ \%\ \overline{\alpha}_1\ >\ \cdots\ <\ n\ \%\ \overline{\alpha}_n\ >,\ \overline{M}\ \}$ that includes the metarepresentation $\bar{t}$ of the subject term, of the strategies $\overline{\alpha}_i$, and of the module $\overline{M}$ where they are evaluated. The evolution of this execution context is defined by some rules, which modify the strategy representations and execute them according to their semantics, governed by the global strategy γ. The rules that do not alter the subject term (but choose alternatives, expand iterations...) are called *control rules* and those modifying the term with rules of the underlying system are called *system rules*. With `control(N)` and `system(N)` as the disjunction of all control and system rules applied to the thread `N`, global control strategies, like `turn(N, M)` for executing `M` strategies in turns starting from the $\mathtt{N}^{\text{th}}$ one and `freec` to execute them concurrently, can be specified as follows:

```
sd =>>(N) := control(N) * ; system(N) .
sd turns(N, M) := =>>(N) ? turns(s(N) rem M, M) : idle .
sd freec := (matchrew C s.t. { T :: < N % S > TS, M } := C
```

```
by C using =>>(N)) ? freec : idle .
```

Auxiliary operations and an interactive interface are defined to easily execute multistrategies and to obtain meaningful counterexample traces when model checking these systems. The complete commented Maude code is available at [9].

Let us omit in this extended abstract the implementation details and conclude with a simple example: the module `LLIST` specifies a list of letters (`a`, `b`, `c`, ...), which can be appended with a `put` rule, and a strategy `seq` that does so with a list of them in order:

```
rl [put] : LS => LS L [nonexec] .
sd seq(nil) := idle . sd seq(L LS) := top(put[L <- L]) ; seq(LS) .
```

After loading the interactive interface in `multistrat-iface.maude` and the module above, for instance, we can execute multiple `seq` calls by turns or concurrently:

```
MStrat> select LLIST .
MStrat> srew eps using seq(a b), seq(c d) by turns .
Solution 1:     a c b d
No more solutions.
MStrat> srew eps using seq(a b), seq(c d) by concurrent .
Solution 1:     a b c d                                                    [...]
Solution 6:     c d a b
No more solutions.
```

More interesting examples are available at [9], including the specification of the Lamport's bakery algorithm and the Tic-Tac-Toe game, where relevant properties are model checked using different combinations of process or player strategies.

6 Related work and conclusions

As we indicated throughout the paper, the reflective capabilities of Maude have extensively been used to build extensions of Maude and frameworks for specific languages and utilities. Apart from Full Maude and the Maude Formal Environment, other relevant examples are Real Time Maude [12] for specification and verification of real-time systems, and the mobile agents extension Mobile Maude [7]. The strategy language was introduced to control rewriting at the object-level without the conceptual difficulties of reflective computations and the intricate shape of metalevel programs. However, some tasks still require resorting to the metalevel, like writing interactive interfaces or generating strategies depending on the specification or some input data.

With these examples, we aim to show that manipulating, transforming, and generating strategies is accessible and has useful applications. The reflective representation of the object-level strategy language provides the means to easily do this within Maude, while having strategies executed by the efficient builtin engine. Another interesting example of this approach is a framework for simulating and verifying membrane systems [15].

References

[1] Peter Borovanský, Claude Kirchner, Hélène Kirchner & Christophe Ringeissen (2001): *Rewriting with Strategies in ELAN: A Functional Semantics*. *Int. J. Found. Comput. Sci.* 12(1), pp. 69–95, doi:10.1142/S0129054101000412.

[2] Tony Bourdier, Horatiu Cirstea, Daniel J. Dougherty & Hélène Kirchner (2009): *Extensional and Intensional Strategies*. In Maribel Fernández, editor: *Proceedings Ninth International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2009, Brasilia, Brazil, 28th June 2009*, *EPTCS* 15, pp. 1–19, doi:10.4204/EPTCS.15.1.

[3] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas & Eelco Visser (2008): *Stratego/XT 0.17. A language and toolset for program transformation*. *Science of Computer Programming* 72(1-2), pp. 52–70, doi:10.1016/j.scico.2007.11.003.

[4] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio & Carolyn Talcott (2019-12): *Maude Manual v3.0*. Available at `http://maude.lcc.uma.es/maude30-manual-html/maude-manual.html`.

[5] Francisco Durán, Steven Eker, Santiago Escobar, Narciso Martí-Oliet, José Meseguer, Rubén Rubio & Carolyn Talcott (2020): *Programming and Symbolic Computation in Maude*. *Journal of Logical and Algebraic Methods in Computer Programming* 110, doi:10.1016/j.jlamp.2019.100497.

[6] Francisco Durán, Santiago Escobar & Salvador Lucas (2004): *New Evaluation Commands for Maude Within Full Maude*. In Narciso Martí-Oliet, editor: *Proceedings of the Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27-April 4, 2004*, *Electronic Notes in Theoretical Computer Science* 117, Elsevier, pp. 263–284, doi:10.1016/j.entcs.2004.06.014.

[7] Francisco Durán, Adrián Riesco & Alberto Verdejo (2007): *A Distributed Implementation of Mobile Maude*. In Grit Denker & Carolyn Talcott, editors: *Proceedings of the 6th International Workshop on Rewriting Logic and its Applications, WRLA 2006, Vienna, Austria, April 1-2, 2006*, *Electronic Notes in Theoretical Computer Science* 176(4), Elsevier, pp. 113–131, doi:10.1016/j.entcs.2007.06.011.

[8] Francisco Durán, Camilo Rocha & José María Álvarez (2011): *Towards a Maude Formal Environment*. In Gul Agha, Olivier Danvy & José Meseguer, editors: *Formal Modeling: Actors, Open Systems, Biological Systems - Essays Dedicated to Carolyn Talcott on the Occasion of Her 70th Birthday*, *LNCS* 7000, Springer, pp. 329–351, doi:10.1007/978-3-642-24933-4_17.

[9] Steven Eker, Narciso Martí-Oliet, José Meseguer, Isabel Pita, Rubén Rubio & Alberto Verdejo: *Strategy language for Maude*. Available at `http://maude.ucm.es/strategies`.

[10] Salvador Lucas (2002): *Context-Sensitive Rewriting Strategies*. *Inf. Comput.* 178(1), pp. 294–343, doi:10.1006/inco.2002.3176.

[11] José Meseguer (1992): *Conditional rewriting logic as a unified model of concurrency*. *Theoretical Computer Science* 96(1), pp. 73–155, doi:10.1016/0304-3975(92)90182-F.

[12] Peter Csaba Ölveczky (2014): *Real-Time Maude and Its Applications*. In Santiago Escobar, editor: *WRLA 2014, Held as a Satellite Event of ETAPS, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, *LNCS* 8663, Springer, pp. 42–79, doi:10.1007/978-3-319-12904-4_3.

[13] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita & Alberto Verdejo (2019): *Model checking strategy-controlled rewriting systems*. In Herman Geuvers, editor: *FSCD 2019, June 24-30, 2019, Dortmund, Germany*, *LIPIcs* 131, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 34:1–34:18, doi:10.4230/LIPIcs.FSCD.2019.31.

[14] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita & Alberto Verdejo (2019): *Parameterized strategies specification in Maude*. In José Fiadeiro & Ionuț Țuțu, editors: *Recent Trends in Algebraic Development Techniques. 24th IFIP WG 1.3 International Workshop, WADT 2018, Egham, UK, July 2–5, 2018, Revised Selected Papers*, *LNCS* 11563, Springer, pp. 27–44, doi:10.1007/978-3-030-23220-7_2.

[15] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita & Alberto Verdejo (2020): *Simulating and model checking membrane systems using strategies in Maude*. In: *7th International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE 2020*, pp. 1–10.