Simulating and model checking membrane systems using strategies in Maude [*]

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

Membrane systems are a biologically-inspired computational model based on the structure of biological cells and the way chemicals interact and traverse their membranes. Although its dynamics is described by rules, encoding membrane systems into rewriting logic is not straightforward due to its complex control mechanisms, and multiple alternatives have been proposed in the literature and implemented in the Maude specification language. The recent release of the Maude strategy language and its associated strategy-aware model checker [15] allow specifying these systems more easily, so that they become executable and verifiable for free. An interactive environment based on a previous prototype by Oana Andrei and Dorel Lucanu [4] is available to operate with the membranes.

1 Introduction

A *membrane system* or *P system* [14] is an unconventional distributed and parallel computational model inspired on the structure and interactions of biological cells, proposed in 1998 by Gheorghe Păun. Its theoretical study has led to interesting results like its Turing completeness and the ability to compute NP problems in polynomial time, albeit at an exponential space grow, and its applications cover both biological and non-biological fields [7]. Although simulating P systems is complex, due to its nondeterministic and distributed nature, some simulators have been developed for research and educational purposes [7]. Verification of these systems through model checking has also been addressed [11, 1].

The connection with rewriting logic and rewriting strategies has been explored in [1, 3, 2, 4]. These works propose different ways of implementing the membrane control mechanisms in the specification language Maude [9] based on rewriting logic. The latter presents a prototype capable of running single evolution steps, which was programmed in Full Maude using a primitive version of the Maude strategy language and some metalevel *strategy controllers* that generate the strategies dynamically. In this paper, we specify the membrane control using strategies generated at *compile-time* from the membrane specifications and valid to evaluate all possible instances. The interactive prototype maintains the membrane specification language of Andrei and Lucanu, but it is reimplemented and enhanced with new features like loading specifications from file using the new external objects of Maude 3.0, showing the multiset of rules applied, computing complete membrane executions, and model checking linear time properties expressed in a builtin but extensible language of atomic propositions. Model checking is directly backed by our model checker for systems controlled by strategies [15], which is able to consider membrane evolution steps as the transitions of the model. The interactive prototype and the strategy-aware model checker can be downloaded at `http://maude.ucm.es/strategies`.

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

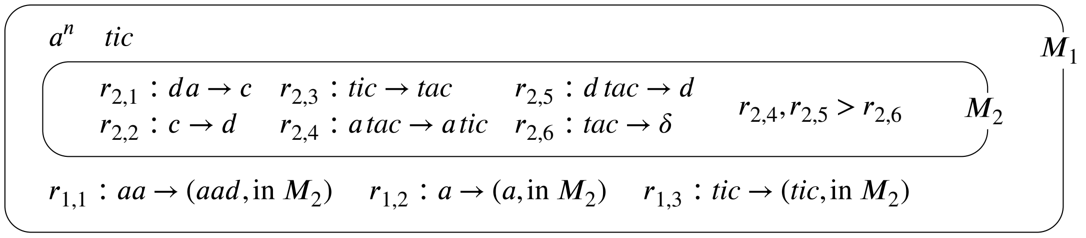
a^n tic
M_1
$r_{2,1} : da \to c$
$r_{2,2} : c \to d$
$r_{2,3} : tic \to tac$
$r_{2,4} : atac \to atic$
$r_{2,5} : d\,tac \to d$
$r_{2,6} : tac \to \delta$
$r_{2,4}, r_{2,5} > r_{2,6}$
M_2
$r_{1,1} : aa \to (aad, \text{in } M_2)$
$r_{1,2} : a \to (a, \text{in } M_2)$
$r_{1,3} : tic \to (tic, \text{in } M_2)$

Figure 1: Venn diagram of a divisor-calculator membrane system

2 Membrane systems

A *P system* is a collection of cell membranes that can be seen as multisets populated by other nested membranes, *objects* playing the role of chemicals, and *evolution rules* describing their reactions and communication. All of them are assumed to be contained inside a single topmost *skin membrane*. Objects are usually opaque symbols represented by letters, and evolution rules $u \to v$ consist of a multiset u of objects and a multiset v of *targets* of the form (w, t) where w is multiset of objects and t determines whether these must stay in the membrane (*here*), be transferred to the enclosing one (*out*), or to a nested one (in_j). Moreover, a special symbol δ causes the enclosing membrane to be dissolved. An *evolution step* is the nondeterministic parallel application of as many evolution rules as possible to the objects of each membrane, often regulated by priority relations. Membrane *computations* are sequences of these steps until no more are possible, and sometimes their results are interpreted as the number of objects yield in a designated membrane. For example, the Venn diagram of Fig. 1 is a membrane system to compute divisors of a number, read as the number of d in the skin membrane. Many variants of P systems have been defined including special objects as promoters and inhibitors, allowing membranes to be created or duplicated, etc.

Membrane configurations are written like $\langle M_1 \mid a\,b\,c\ \langle M_2 \mid c\,d \rangle\rangle$ where the membrane M_1 contains the objects a, b and c and the membrane M_2, which in turn contains two objects, c and d. More precisely, an evolution step consists of the following phases:

1. Applying the evolution rules to each membrane in a *maximal parallel* manner (see below).
2. Sending and receiving the objects contained in *out*, *in* and *here* targets.
3. Dissolving membranes containing δ, thus dropping its objects and membranes to the enclosing membrane.

The *maximal parallel rewrite* is described by a multiple choice of rules or multiset $A_i : R_i \to \mathbb{N}$ for each membrane M_i with rules R_i. A multiset of objects W can be applied a multiset of rules A if the union of their left-hand sides with multiplicities is contained in W, and the result gets that union replaced by the union of the right-hand sides in A. Such a choice A is *maximal* if $A \cup \{r\}$ cannot be applied to W for no matter which rule r. Hence, a maximal parallel rewrite is the application of a maximal multiset of rules A_i to each membrane with at least one non-empty A_i. The choice of each A_i may not be unique, so this phase is nondeterministic. Moreover, when a priority relation ρ_i is imposed, not all choices are admissible. We consider two ways of understanding the rule priorities: a *strong* sense, in which a choice A is admissible if for all $r >_{\rho_i} r'$, $A(r) > 0$ implies $A(r') = 0$; and a *weak* sense, if for $r \in R_i$ no B can be applied such that $B(r) > A(r)$ and $B(r') = A(r')$ for all r' such that $r \not>_{\rho_i} r'$. Intuitively, a rule cannot be applied at all in the strong sense if a rule with higher priority is applicable, but it could be applied in the weak sense as long as it does not limit the application of any higher priority one.

3 Rewriting logic, Maude and its strategy language

Rewriting logic [13] was proposed by José Meseguer as a unified model of concurrency where nondeterministic and possibly conditional rewriting rules are defined on top of an equational logic. A rewrite theory $\mathscr{R} = (\Sigma, E, R)$ consists of a signature Σ of sorts and operators, a set of equations E, and a set of rewriting rules R. The executions of a rewriting system are the successive and independent application of these rules on terms, modulo equations and axioms like commutativity, associativity and identity. Maude [9] is a specification language based on rewriting logic, where rewrite systems can be specified compositionally, executed and analyzed. 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 allow specifying arbitrary constraints on how rules should be applied.

Although strategies have been used in Maude since its beginnings thanks to its reflective features, to make strategy specification more accessible and understandable, an object-level strategy language was proposed, tested, and finally implemented at the C++ level in Maude 3.0. Its design is inspired on other strategy languages like ELAN [5] and Stratego [6]. The main ingredient of the language is the application of rules rl`[`x_1 `<-` t_1`,` ... x_n `<-` t_n`]` referred by their labels rl and taking an optional initial substitution. Tests `match` P `s.t.` C discard executions unless the subject term matches P and satisfies the condition C. The initial 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 permits the executions allowed by any of its arguments, the iteration $\alpha*$ and normalization $\alpha\,!$ operators that iterate α any number of times or until the last successful iteration, 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. Another 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, strategies can be given names to be called, receive parameters, and be defined recursively in strategy modules. More details can be found in [9, §10] and examples are shown in the following sections.

Strategy language expressions do not necessarily describe a deterministic sequence of rewrites since a rule application may have different outcomes, and various combinators like the disjunction and iteration introduce nondeterministic choices. Two Maude commands `srewrite` and `dsrewrite` explore all possible execution paths and show all the solutions they reach.

4 Representing membrane systems in Maude

Membrane systems have already been specified in rewriting logic and Maude [3, 4]. The nested membrane structure and the multisets of objects are naturally represented by terms with commutative and associative operators, but the challenge is applying evolution rules locally, in a maximal way, and respecting the turns of communication and dissolving. Previous works have solved it by controlling rule application using Maude reflective features [1], by representing evolution rules and computing the steps at the object level [3], or even using a particular combination of reflection and a primitive version of the Maude strategy language [4]. Our specification of membrane configurations in the following module only slightly differs from these:

```
mod P-SYSTEM-CONFIGURATION is
  including QID-LIST .

  sorts Obj Membrane MembraneName Target .
  sorts EmptySoup MembraneSoup ObjSoup TargetSoup Soup .

  subsort Obj < ObjSoup .        subsort Membrane < MembraneSoup .
  subsort Target < TargetSoup .
  subsorts EmptySoup < MembraneSoup ObjSoup TargetSoup < Soup .

  op <_|_>   : MembraneName Soup -> Membrane [ctor] .
  op delta   : -> Obj [ctor] .

  op empty   : -> EmptySoup [ctor] .
  op __      : Soup Soup -> Soup [ctor assoc comm id: empty] .
```

A membrane consists of a name and a multiset juxtaposition of objects, membranes, and targets of sort `Soup`. Each component type defines a subsort, `ObjSoup`, `MembraneSoup`, `TargetSoup` to ease operating with them. Targets are expressed as pairs:

```
  ops here out : -> Target [ctor] .
  op  in_ : MembraneName -> Target [ctor] .
  op `(_,_`) : Soup Target -> TargetSoup [ctor] .
```

Objects with a common target are combined into a common pair by an equation, and three rules are defined to resolve the communication between cells:

```
  vars MN MN'  : MembraneName .
  vars W W' CW : Soup .
  var  T       : Target .

  eq (W, T) (W', T) = (W W', T) .

  rl [here] : (W, here) => W .
  rl [in]   : (CW, in MN) < MN | W > => < MN | W CW > .
  rl [out]  : < MN | W (CW, out) > => < MN | W > CW .
```

Another rule `dis` triggers the effect of the δ symbol by dissolving a non-skin membrane:

```
  rl [dis] : < MN | W < MN' | W' delta > > => < MN | W W' > .
endm
```

Evolution rules are represented as plain rewriting rules, leaving their controlled application to strategies[1]. Part of the definition of these strategies is generic and part depends on the rules and priorities of the membrane system, but not on any particular configuration. For a membrane M with rules $r_1, \ldots, r_n$ and without priorities, such a specific definition will be

```
sd membraneRules(M) := r_1 | ··· | r_n .
```

A maximal parallel step is specified by the `mpr` strategy in the following strategy module, in terms of the system-specific `handleMembrane`.

[1]Another possibility to limit the locality of evolution rules is adding a membrane context: the rule $v \to w$ for the membrane M could also be transformed to `<` M `|` v `S:Soup >` $\Rightarrow$ `<` M `|` w `S:Soup >`.

```
smod P-SYSTEM-STRATEGY is
  protecting P-SYSTEM-CONFIGURATION .

  strat  handleMembrane              : MembraneName @ Soup .
  strats mpr visit-mpr communication                @ Soup .
  strat  nested-mpr                  : MembraneSoup @ Soup .

  var MN : MembraneName .  var S : ObjSoup .  var TS  : TargetSoup .
  var MS : MembraneSoup .  var K : Nat .

  sd mpr := visit-mpr ; communication ; (dis !) .
  sd communication := (in | out | here) ! .

  sd visit-mpr :=
    (matchrew < MN | S MS > by S using handleMembrane(MN))
      ? matchrew < MN | S TS MS > by MS using try(nested-mpr(MS))
      : matchrew < MN | S MS >    by MS using nested-mpr(MS) .

  sd nested-mpr(empty) := fail .
  sd nested-mpr(M MS) := (matchrew M MS by M using visit-mpr)
       ? matchrew M' MS by MS using try(nested-mpr(MS))
       : matchrew M  MS by MS using nested-mpr(MS) .
```

The three phases of an evolution step (see Section 2) are concatenated in the main strategy `mpr`: (1) `visit-mpr` applies the evolution rules to the membranes, (2) `communication` transmits all targeted objects through the membranes, and (3) `dis !` dissolves them exhaustively. The `visit-mpr` strategy executes the parameter `handleMembrane` on the objects of the topmost membrane and then delegates the nested ones to `nested-mpr`. The requirement that at least an evolution rule must be applied in the whole system is enforced by the conditionals. When no rewrite is possible the strategies must fail, but if `handleMembrane` succeeds, the evolution of the nested membranes is allowed to fail by the `try`$(\alpha) \equiv \alpha$ `? idle : idle` combinator. `nested-mpr` receives the full nested membrane soup and recursively processes the membranes one at a time[2].

The `handleMembrane` strategy can be accommodated to different situations, depending on the rule priorities and their interpretation. The case without priorities or with a weak interpretation of them can be handled by the following strategy `inner-mpr`:

```
  strat inner-mpr : MembraneName @ Soup .
  sd inner-mpr(MN) := (matchrew S TS by S using membraneRules(MN))
                         ? inner-mpr(MN) : amatch TS s.t. TS =/= empty .
```

For a membrane name `MN`, it calls `membraneRules` repeatedly until it cannot be applied again. The matching pattern `S TS` separates the objects from the targets, and ensures that the products of the evolution rules are not used in the same step. Finally, **amatch** `TS` is executed to check whether there is a target in the term, i.e. if an evolution rule has been applied. With `membraneRules` being a disjunction of rules, as above, the `mpr` strategy implements a maximal parallel step.

Priorities in the weak sense can also be handled by adding a lattice of `or-else` combinators to the `membraneRules` definition. Assuming that $P \subseteq R \times R$ is a generator set for this priority relation, a recursive procedure is able to generate a strategy respecting the priorities at each application: (1) consider the

[2]The definition of the `nested-mpr` strategy is not efficient since all possible matches of the set-like argument will be tried at each call, hence processing the membranes in all possible orderings. This can be avoided at the expense of clarity.

disjunction of the minimal elements in P, (2) replace each such r by $(r_1 \mid \cdots \mid r_n)$ **or—else** r where $r_1, \ldots, r_n$ are all rules satisfying $(r_i, r) \in P$, (3) iterate this procedure up to the maximal elements. To optimize the algorithm on the fly, we can transform α **or—else** $\beta \mid \alpha$ **or—else** γ to α **or—else** $(\beta \mid \gamma)$. The correctness of this procedure follows from the fact that a rule will only be applied when its predecessors in the order have failed, due to the semantics of the `or-else` combinator, and that all rules appear in the expression because they must be reachable from the minimal elements of the order relation. The size of the strategy is bounded by the number of rules and the relation pairs in P.

When the priority of the rules is understood in the strong sense, the skeleton of `inner-mpr` cannot be exploited since it executes each rule independently, and respecting strong priorities requires knowing which rules have been previously applied. However, a variation of the previous procedure can be used. The parameter `handleMembrane` is defined using a new strategy `strong-mpr` that receives a set of labels of rules that have already been applied:

```
strat strong-mpr : MembraneName QidSet @ Soup .
sd handleMembrane(MN) := strong-mpr(MN, empty) .
```

The definition of `strong-mpr(`M`, AR)` for a strategy M whose priority is generated by P is given recursively as:

1. Take the disjunction of $\alpha_r := r$; `strong-mpr(`M`, (`r`, AP))` for every minimal element of P. The recursive call adds r to the comma-separated set `AP` of applied rules.

2. Replace each term in the disjunction by

   $(\alpha_{r_1} \mid \cdots \mid \alpha_{r_n})$ **or—else**
   (**match** `S` **s.t.** $\{r' \in R_i \mid r' >_{\rho_i} r\}$ `intersect AP = empty ;` α_r)

   where `S` is a variable of sort `Soup`, and $r_1, \ldots, r_n$ are all the elements that satisfy $(r_i, r) \in P$.

3. Iterate this procedure on the left-hand side of the **or—else** up to the maximal elements.

4. To ensure that `strong-mpr` fails iff no rule has been applied and that no rules are applied inside targets, complete the generated expression, say γ, to

   (**matchrew** `S TS` **by** `S` **using** γ) **or—else** **match** `S` **s.t.** `AP =/= empty`

The previous construction guarantees that rules are executed only if no rule with higher precedence has been applied.

Finally, executions to irreducible configurations are described by the following strategy `comp`. Note that we omit the trivial executions in which no step has been applied.

```
strats mcomp mcomp2 @ Soup .
sd mcomp := mpr ; mcomp2 .
sd mcomp2 := mpr ? mcomp2 : idle .
```

Executions of a bounded number of steps can also be specified with definitions **sd** `mcomp2(0) := idle .` and **sd** `mcomp2(s(K)) := mpr ? mcomp2(K) : idle ..`

Fortunately, the interactive environment in Section 6 will make unnecessary to instantiate these strategies manually: `membraneRules` and `strong-mpr` are contructed equationally following the above procedures from the membrane programs read from file.

5 Model checking

Model checking [8] is an automated verification technique based on the exhaustive exploration of the executions of a system model to check properties of its intended behavior. Multiple variants and algorithms exist, but traditionally the model is represented as a state and transition system. Formally, this is a Kripke structure $\mathscr{K} = (S, \to, AP, I, \ell)$ with a set of states S, a binary relation $(\to) \subseteq S \times S$, a finite set of atomic propositions AP, a finite set of initial states $I \subseteq S$, and a labeling function $\ell : S \to \mathscr{P}(AP)$. Properties are expressed in terms of these labels, usually in some temporal logics providing operators to specify how they occur in time[3]. Well-known examples are CTL* and its sublogics LTL (Linear Temporal Logic) and CTL (Computational Tree Logic).

For rewriting systems, a natural model is considering terms as states and single-step rule rewrites as transitions; and for P systems, the natural model has membrane configurations as states and evolution steps as transitions. Mapping membrane configuration to terms is straightforward, as we have seen in Section 4, but matching evolution steps and rewrite rules is not. Strategies and the possibility to consider them as atomic transitions in our strategy-aware model checker is a solution to this problem.

Maude includes since its 2.0 version an on-the-fly LTL model checker [10] that we have recently extended to support systems controlled by strategies [15]. Looking at strategies as subsets E of *admitted* executions of a model $\mathscr{K}$, the satisfaction of a linear-time property $(\mathscr{K}, E) \vDash \varphi$ can be understood as $\mathscr{K}, \pi \vDash \varphi$ for all $\pi \in E$, since linear properties refer to individual executions[4]. By providing a nondeterministic small-step semantics for the execution of strategies, whose steps correspond to rule rewrites of the underlying terms, we determine which are the executions allowed by a Maude strategy expression. In practice, the graph determined by the strategy semantics gives the standard Kripke structure where to apply the standard model-checking algorithms.

Users should specify the atomic propositions as regular operators of a predefined sort `Prop` and its satisfaction relation using a predefined symbol `_|=_` (see [15] or [9, §12] for details). For membrane systems, we provide a predefined set of properties, among others:

```
mod P-SYSTEM-PREDS is
  protecting P-SYSTEM-CONFIGURATION .
  including SATISFACTION .     protecting EXT-BOOL .

  subsort Soup < State .

  op isAlive  : MembraneName          -> Prop [ctor] .
  op contains : MembraneName Soup     -> Prop [ctor] .

  op {_}      : BoolExpr              -> Prop [ctor] .
  op _=_      : NatExpr NatExpr       -> BoolExpr [ctor] .
  op count    : MembraneName Soup     -> NatExpr [ctor] .
```

For example, the property `{ count(M1, a) = 2 * count(M2, b) }` says that the number of `a`s in `M1` doubles the number of `b`s in `M2`. The satisfaction of these propositions is defined equationally in the same module. Finally, the predefined `STRATEGY-MODEL-CHECKER` module declares a special `modelCheck` operator that gives access to the strategy-aware model checker.

[3]For model checking, it is usually assumed that all executions are non-terminating, and so finite executions are *stutter-extended* by repeating their final state forever, like in Spin [12] and other verification tools.

[4]Branching-time properties have also been studied, and CTL* and μ-calculus can be checked on Maude specifications controlled by strategies, including these membrane systems, via an external tool [16].

```
op modelCheck : State Formula Qid QidList Bool ~> ModelCheckResult [...] .
```

Its arguments are the initial state, the LTL formula to be checked, and the name of a strategy in the module to control the system, plus two other optional arguments. The fourth one is very useful in this case: a list of named strategies whose executions must be considered as atomic steps of the verified system. Like this, the executions of an `mpr` step can be automatically seen as the steps of the model, making the *next* operator of the temporal logics work as expected and hiding the intermediate states in which the rules that are supposed to be executed in parallel are being applied. Thus, issuing

```
red modelCheck(< M1 | a b < M2 | a > >, [] contains(M1, a), 'comp, 'mpr) .
```

will check the property that `M1` always contain one `a` in all membrane executions from the given initial one. Again, the interactive environment automatically handles this behind the scenes.

6 The membrane system environment

After downloading the membrane example from `http://maude.ucm.es/strategies` and loading the `memparse.maude` file into Maude, we can execute the interactive environment with

```
Maude> erew initREPL(repl) < repl : MemREPL | none > .

      ** Membrane system environment in Maude **

Membrane>
```

The environment offers different commands that are listed by typing `help`. The first command is `load` followed by a filename, which loads a membrane specification from a file and runs the commands in it. `load divisor.memb` loads the membrane system of Fig. 1. Its membrane `M2` is there expressed in the format of [4] as follows:

```
membrane M2 is
  ev r21 : d a -> c .  ev r23 : tic -> tac .      ev r25 : d tac -> d .
  ev r22 : c   -> d .  ev r24 : a tac -> a tic .  ev r26 : tac -> delta .
  pr r24 > r26 .       pr r25 > r26 .
end
```

When loading the file, the strategies described in Section 4 are generated for later use by the following commands. The names of the loaded membranes are shown by the `show membranes` command, and their complete definition can be listed with `show` followed by a membrane name.

Two commands, `trans` and `compute`, allow simulating evolution steps and computations. The first one executes a single step, indicating the multiset of rules applied for each membrane.

```
Membrane> trans < M1 | a a a tic < M2 | d tac > > .
Solution 1 with r11 r12 r13 in M1, r25 in M2 :
        < M1 | c c c < M2 | a a a d d tic > >
Solution 2 with r12 r12 r12 r13 in M1, r25 in M2 :
        < M1 | c c c < M2 | a a a d tic > >
No more solutions.
```

The `compute` command shows all irreducible states that can be found by successive transitions.

```
compute < M1 | a a a a a a a a tic < M2 | empty > > .
Solution 1:     < M1 | d d d d >
Solution 2:     < M1 | < M2 | d d d > >
Solution 3:     < M1 | d d >
Solution 4:     < M1 | d >
No more solutions.
```

The interpretation of rule priorities, either `weak` or `strong`, can be set globally with the `set priority` command that causes strategies to be recompiled accordingly. By default, strong priorities are used.

Moreover, we can check properties of the membrane executions by using the `check` command. The properties are expressed in LTL for the predefined language of atomic propositions described in Section 5, which can anyhow be extended by modifying the environment source code.

```
Membrane> check < M1 | a a a a a a a a a a a a < M2 | a b b > >
   satisfies [] ({ count(M1, d) = 0 } \/ { count(M1, d) divides 12 }) .
The property is satisfied.
```

When the property is not satisfied, the output shows a counterexample describing the intermediate steps and the rules that have been applied.

```
Membrane> check < M2 | a a d d tic >
  satisfies [] (contains(M2, tac) -> O contains(M2, tic)) .
| < M2 | a a d d tic >
 with r21 r21 r23 in M2
| < M2 | c c tac >
 with r22 r22 r26 in M2
X < M2 | delta d d >
```

The `check` command admits bounded model checking on the number of objects in the configuration, where the bound may be indicated between brackets after the `check` keyword. This is useful for membrane systems that, unlike this example, have an unbounded configuration space.

7 Related work and conclusions

The transformation from membrane systems to rewrite theories controlled by strategies and the interactive prototype are inspired by [4], but here strategies are compiled statically from membrane specifications and evaluated in a much cleaner way using the Maude strategy language, now efficiently supported in Core Maude. That work pointed out the difficulty to apply analysis tools like the model checker, already used in their first work [1], to their strategy-based prototype that we have solved with the strategy-aware model checker and its *opaque strategies* feature. Although the strategy-free prototype of their first work was able to model check LTL properties, its discrete time steps do not correspond to evolution steps of the membrane system, and so properties like the second of the previous section cannot be expressed, and counterexamples are much longer and difficult to understand. Moreover, our prototype seems to be more efficient: the first property of the divisor's example is model checked 3.6 times faster in ours, and the square calculation example included in their article runs in 56% of the time, even though our prototype reads the membrane specification from an external file and starts an interactive interface before model checking. There are other examples of model-checking tools for membrane systems like kPWorkbench [11] for LTL and CTL properties of *kernel P systems*, but the interesting point of our approach is that model checking comes automatically as a consequence of specifying the system in rewriting logic and the Maude strategy language.

As future work, other variants and extensions of membrane systems not currently supported by the environment can be implemented, like promoters and inhibitors, membrane creation and duplication, nonintegral object multiplicities, etc. Moreover, the simulation and verification capabilities can be extended with a search command, or support for other temporal logics. The changes are expected to be done without much effort from the current version of the prototype.

References

[1] Oana Andrei, Gabriel Ciobanu & Dorel Lucanu (2005): *Executable Specifications of P Systems*. In Giancarlo Mauri, Gheorghe Păun, Mario J. Pérez-Jiménez, Grzegorz Rozenberg & Arto Salomaa, editors: *Membrane Computing, 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers*, *LNCS* 3365, Springer, pp. 126–145, doi:10.1007/978-3-540-31837-8_7.

[2] Oana Andrei, Gabriel Ciobanu & Dorel Lucanu (2006): *Expressing Control Mechanisms of Membranes by Rewriting Strategies*. In Hendrik Jan Hoogeboom, Gheorghe Puaun, Grzegorz Rozenberg & Arto Salomaa, editors: *Membrane Computing, 7th International Workshop, WMC 2006, Leiden, The Netherlands, July 17-21, 2006, Revised, Selected, and Invited Papers*, *LNCS* 4361, Springer, pp. 154–169, doi:10.1007/11963516_10.

[3] Oana Andrei, Gabriel Ciobanu & Dorel Lucanu (2007): *A rewriting logic framework for operational semantics of membrane systems*. *Theor. Comput. Sci.* 373(3), pp. 163–181, doi:10.1016/j.tcs.2006.12.016.

[4] Oana Andrei & Dorel Lucanu (2009): *Strategy-Based Proof Calculus for Membrane Systems*. In Grigore Roşu, editor: *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest, Hungary, March 29-30, 2008*, *ENTCS* 238(3), Elsevier, pp. 23–43, doi:10.1016/j.entcs.2009.05.011.

[5] 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.

[6] 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.

[7] Gabriel Ciobanu, Mario J. Pérez-Jiménez & Gheorghe Păun, editors (2006): *Applications of Membrane Computing*. Natural Computing Series, Springer, doi:10.1007/3-540-29937-8.

[8] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith & Roderick Bloem, editors (2018): *Handbook of Model Checking*. Springer, doi:10.1007/978-3-319-10575-8.

[9] 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`.

[10] Steven Eker, José Meseguer & Ambarish Sridharanarayanan (2004): *The Maude LTL Model Checker*. In Fabio Gadducci & Ugo Montanari, editors: *Proceedings of the Fourth International Workshop on Rewriting Logic and its Applications, WRLA 2002, Pisa, Italy, September 19-21, 2002*, *ENTCS* 71, Elsevier, pp. 162–187, doi:10.1016/S1571-0661(05)82534-4.

[11] Marian Gheorghe, Savas Konur, Florentin Ipate, Laurentiu Mierla, Mehmet E. Bakir & Mike Stannett (2015): *An Integrated Model Checking Toolset for Kernel P Systems*. In Grzegorz Rozenberg, Arto Salomaa, José M. Sempere & Claudio Zandron, editors: *Membrane Computing - 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*, *LNCS* 9504, Springer, pp. 153–170, doi:10.1007/978-3-319-28475-0_11.

[12] G.J. Holzmann et al.: *Spin - Formal Verification*. Available at `https://spinroot.com`.

[13] 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.

[14] Gheorghe Păun (2002): *Membrane Computing: An Introduction*. Natural computing series, Springer, doi:10.1007/978-3-642-56196-2.

[15] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita & Alberto Verdejo (2019): *Model checking strategy-controlled rewriting systems*. In Herman Geuvers, editor: *4th International Conference on Formal Structures for Computation and Deduction, 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.

[16] Rubén Rubio, Narciso Martí-Oliet, Isabel Pita & Alberto Verdejo (2020): *Strategies, model checking and branching-time properties in Maude*. In Santiago Escobar, editor: *Accepted in Rewriting Logic and Its Applications - 13th International Workshop, WRLA 2020*, LNCS, Springer, pp. 1–15.