

Rewriting P Systems in Maude

Oana ANDREI, Gabriel CIOBANU, Dorel LUCANU

“A.I.Cuza” University of Iași, Faculty of Computer Science

E-mail:{oandrei, gabriel, dlucanu}@info.uaic.ro

Keywords: P systems, rewriting systems, specifications, Maude, verification.

Abstract

We present the P systems as executable specifications in Maude, a software system supporting rewriting and equational logic. We use META-LEVEL module in Maude to define the P system maximal parallel evolution rules, and then provide an operational semantics of the P systems. Using the tower of reflection levels in Maude, we can verify the properties expressed as linear temporal logic formulas by using the model checker implemented in Maude.

1 Introduction

Membrane computing is a branch of natural computing which studies distributed and parallel computing models abstracted from the living cell structure and functioning. Membrane computing is based on *membrane systems* or *P systems*, a new class of computing devices introduced in [7]. The approach is based on hierarchical systems: finite cell-structures consisting of membranes embedded in a main membrane. The membranes determine regions where objects and evolution rules can be placed. The objects evolve according to the rules associated with each region. A computation starts from an initial configuration of the system, and terminates when no further rule can be applied. P systems are inspired by biological systems, but they are based on the theory of automata and formal languages. Since their introduction, many results of universality for P systems were proved, and several problems were solved with the help of formal languages. The field is very active; new properties are discovered, as well as connections with already known concepts. It is desirable to find more connections with the applied computer science, including implementations and executable specifications. A sequential software simulator of membrane systems is presented in [1], and a parallel simulator is presented in [2]. This paper presents *executable specifications* for P systems. Such a specification is executed by the sequential rewriting software tool called Maude. Forced to execute parallel steps on a sequential machine, we emphasize a new dimension of the membrane computing, and provide an algorithmic description of the rather awkward “nondeterministic maximal parallel” evolution in P systems. Maude allows the verification of some properties expressed as linear temporal logic formulas by using a model checker.

The paper is organized as follows. Section 2 briefly presents the P systems, as well as the mathematical specification system called Maude. We emphasize on the power given by reflection in Maude. Section 3 presents the specifications of the P systems in Maude. An example and the execution of the Maude specifications are described in Section 4. Model checking some temporal properties of the P systems are discussed in Section 5.

2 P Systems and Maude

A detailed description of the P systems can be found in [8]. A **P system** consists of several membranes that do not intersect, and a *skin membrane*, surrounding them all. The membranes delimit *regions*, and contain multisets of *objects*, as well as *evolution rules*. Only rules in a region delimited by a membrane act on the objects in that region. Moreover, the rules can contain target-indications, specifying the membrane where an object should be sent after applying the rule. The objects can pass through membranes, in two directions: they can be sent *out* the membrane which delimits a region from outside, or can be sent *in* one of the membranes which delimit a region from inside, precisely identified by its label. In a step, an object can pass only through a membrane. The membranes can be *dissolved*. When such an action takes place, all the objects of the dissolved membrane remain free in the membrane placed immediately outside, but the evolution rules of the dissolved membranes are lost. The skin membrane is never dissolved. The application of evolution rules is done in parallel and it is eventually regulated by *priority* relationships between rules.

We can identify a membrane structure with a tree (with skin as its root), or a string of correctly matching parentheses, placed in a unique pair of matching parentheses; each pair of matching parentheses corresponds to a membrane. Graphically, a membrane structure is represented by a Venn diagram in which two sets can be either disjoint, or one the subset of the other. The membranes are labelled in a one-to-one manner. A membrane without any other membrane inside is said to be *elementary*. The space outside the skin membrane is called the *outer region*. More formally, a P system is a structure $\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_o)$, where:

- (i) O is an alphabet of objects;
- (ii) μ is a membrane structure consisting of labelled membranes;
- (iii) w_i are multisets over O associated with the regions defined by μ ;
- (iv) R_i are finite sets of evolution rules over O associated with the membranes, of typical form $ab \rightarrow a(c, in_2)(c, out)$;
- (v) i_o is either a number between 1 and m specifying the *output* membrane of Π , or it is equal to 0 indicating that the output is the outer region.

These are the general P systems; many other variants and classes were introduced. In our approach, we prefer to define the P systems architecture by starting from elementary membranes, and then defining composed membranes.

An *elementary membrane* is denoted by $M = (R_M, init)$, where R_M represents the finite set of evolution rules, and $init$ is the initial configuration. A transition between configurations is defined as

$$C \Rightarrow C' \text{ iff } \exists C_1, C'_1, \text{ and } l \rightarrow r \in R_M \text{ such that } C = l C_1, C' = r C'_1, \\ C'_1 = C_1 \text{ if } C_1 \text{ is irreducible, or } C_1 \Rightarrow C'_1 \text{ otherwise.}$$

C is irreducible whenever there does not exist C' such that $C \Rightarrow C'$.

A *composed membrane*, that is a membrane with other membranes M_1, \dots, M_k inside it, is denoted by $(M_1, \dots, M_k, R_M, init)$, where each $M_i (1 \leq i \leq k)$ is an elementary or a composed membrane. R_M represents the finite set of evolution rules of M , and $init$ is its initial configuration. The configurations of a composed membrane are of form $(C, (C_1, \dots, C_k))$, where C_i is the configuration of the corresponding membrane M_i . A transition between these configurations is defined as

$$(C, (C_1, \dots, C_k)) \Rightarrow (C', (C'_1, \dots, C'_k)) \text{ iff} \\ C \Rightarrow C', C_1 \Rightarrow C'_1, \dots, C_k \Rightarrow C'_k$$

In this way, the system passes from a configuration to another by using the evolution rules in parallel: all objects of the membranes which can be the subject of local evolution rules evolve simultaneously.

A sequence of transitions steps $C_0 \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \dots \Rightarrow C_n \Rightarrow \dots$ is a computation. A computation is successful if the computation is finite, that is, there is no rule applicable to the objects present in the last configuration. In a final configuration, the result of a successful computation is the total number of objects present in the membrane considered as the output membrane. We simplify this procedure; no internal membrane is specified as an output membrane, and so the result is given by the number of objects in the skin membrane.

Maude is a software system developed around the Maude language. Core Maude is the Maude interpreter implemented in C++; it provides the Maude's basic functionality. Full Maude is an extension written in Maude itself, allowing combination of various Maude modules to build more complex modules. Maude can be used for many applications with competitive performance and advantages over the conventional code. The current Maude implementation can execute syntactic rewriting with speeds from half a million to several million rewrites per second, depending on the particular application and machine. It is able to work well with multisets having millions of elements.

The Maude system is available free of charge, under the terms of the GNU General Public License, at the Maude home page <http://maude.cs.uiuc.edu>. Many useful materials are available, and Maude binaries are provided for selected architectures and operating systems, together with installation instructions.

Maude is essentially a mathematical language. The OBJ theory and languages [6] have influenced the Maude design and philosophy. The basic programming statements are equations, membership assertions, and rules. Their rewriting semantics is given by the fact that instances of the lefthand side pattern are replaced by corresponding instances of the righthand side. A Maude program containing only equations and membership assertions is called a *functional module*. The equations are used as rules (equational rewriting), and the replacement of equals for equals is performed only from left to right.

A Maude program containing both equations and rules is called a *system module*. Rules are not equations, they are local transition rules in a possibly concurrent system. Unlike for equations, there is no assumption that all rewriting sequences will lead to the same final result, and for some systems there may not be any final states. The functional modules define a functional sublanguage of Maude, and the system modules extend the purely functional semantics of equations to the concurrent rewriting semantics of rules.

If we consider, for example, a membrane system in which we have the objects floating in a soup (that is, a multiset of objects), then the objects can interact in this soup, and can work locally according to specific rewriting rules. These rules are the local transition rules of the system, and they can be applied concurrently to different membranes of the system. The rewriting performed for membranes is a multiset rewriting. In Maude this is specified in the equational part of the program (system module) by declaring that the multiset union operator satisfies the associativity and commutativity equations, and has also an identity. This is done simply by using attributes, and this information is used to generate a multiset matching algorithm. Further expressiveness is gained by various features as equational pattern matching, user-definable syntax and data, generic types and modules, support for objects, reflection.

Regarding both expressiveness and performance, we can mention the *evaluation strategies* and use of *reflection property*. Evaluation strategies control the positions in which equations can be applied, giving the user the possibility of indicating which arguments to evaluate before simplifying a given operator with the equations. Reflective computations allow the link between meta-level and the object level, whenever possible. A typical meta-level computation may perform efficiently millions of rewrites at the object level, paying a reasonable linear cost in changing the representations from the meta-level to the object level and back, only at the beginning and at the end of the computation.

Maude may also be viewed as a metalanguage, through which many different domain-specific languages can be implemented. It is possible to develop and use various tools of mathematics and logic. In this paper we actually implement the theory of P systems in Maude.

Essentially, a Maude program is a logical theory, and a Maude computation is a logical deduction using the axioms specified in the program. The foundations of Maude is given by membership equational logic and rewriting logic. A functional module specifies a theory in membership equational logic. Mathematically, we can view such a theory as a pair $(\Sigma, E \cup A)$, where Σ is the signature and specifies the type structure, E is the collection of equations and memberships declared in the functional module, and A is the collection of equational attributes (e.g., assoc, comm) declared for different operators.

Similarly, a system module specifies a rewriting theory, that is, a theory in rewriting logic. A *signature* in rewriting logic is an equational theory (Σ, E) , where Σ is an equational signature and E is a set of Σ -equations and it describes a particular structure for the state of a system (for instance, for string rewriting systems E consists of the associativity axiom, for multiset rewriting systems E consists of the associativity and commutativity axioms, and for term rewriting systems E is empty).

A *sentence* over the signature (Σ, E) is an expression of the form $(\forall X)[t]_E \rightarrow [t']_E$, where t and t' are $\Sigma(X)$ -terms and $[t]_E$ denotes the equivalence class of t modulo the equations E . A sentence describes the possible transitions from the states described by $[t]$ to the corresponding states described by $[t']$. If $X = \{x_1, \dots, x_n\}$, then we denote a sentence by $[t(\bar{x})]_E \rightarrow [t'(\bar{x})]_E$, where \bar{x} is the sequence x_1, \dots, x_n . If E and X are understood from the context, we simply write $[t(\bar{x})] \rightarrow [t'(\bar{x})]$ or $[t] \rightarrow [t']$. A *rewriting specification* \mathcal{R} is a 4-tuple $\mathcal{R} = (\Sigma, E, L, R)$ where (Σ, E) is a rewriting logic signature, L is a set whose elements are called *labels*, and R is a set of labelled rewriting rules (sentences) written as $r : [t(\bar{x})]_E \rightarrow [t'(\bar{x})]_E$. The inference rules of rewriting logic allow to deduce general (concurrent) transitions which are possible in a system satisfying \mathcal{R} . We say that \mathcal{R} *entails the sentence* $[t] \rightarrow [t']$ and write $\mathcal{R} \vdash [t] \rightarrow [t']$ iff $[t] \rightarrow [t']$ can be obtained by finite application of the following *inference rules*:

- (1) *Reflexivity*. For each $\Sigma(X)$ term t ,

$$\overline{[t] \rightarrow [t]}$$

- (2) *Congruence*. For each operation symbol $f \in \Sigma$,

$$\frac{[t_1] \rightarrow [t'_1], \dots, [t_n] \rightarrow [t'_n]}{[f(t_1, \dots, t_n)] \rightarrow [f(t'_1, \dots, t'_n)]}$$

- (3) *Unconditional replace*. For each $r : [t(\bar{x})] \rightarrow [t'(\bar{x})]$ in R ,

$$\frac{[u_1] \rightarrow [v_1], \dots, [u_n] \rightarrow [v_n]}{[t(\bar{u}/\bar{x})] \rightarrow [t(\bar{v}/\bar{x})]}$$

- (4) *Transitivity*.

$$\frac{[t_1] \rightarrow [t_2], [t_2] \rightarrow [t_3]}{[t_1] \rightarrow [t_3]}$$

The general theory of the rewriting logic allows *conditional sentences* and *conditional rewriting rules*. The interested reader is invited to read, e.g., [3].

Rewriting logic is *reflective*, i.e. there is a (finitely presented) *universal rewriting specification* \mathcal{U} such that for any (finitely presented) rewriting specification \mathcal{R} (including \mathcal{U} itself), we have the following equivalence:

$$\mathcal{R} \vdash [t] \rightarrow [t'] \text{ iff } \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \bar{t}' \rangle,$$

where $\overline{\mathcal{R}}$ and \bar{t} are terms representing \mathcal{R} and t as data elements of \mathcal{U} . Since \mathcal{U} is representable in itself, it is possible to achieve a “reflective tower” with an arbitrary number of reflection levels:

$$\mathcal{R} \vdash [t] \rightarrow [t'] \text{ iff } \mathcal{U} \vdash \langle \overline{\mathcal{R}}, \bar{t} \rangle \rightarrow \langle \overline{\mathcal{R}}, \bar{t}' \rangle \text{ iff } \mathcal{U} \vdash \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \bar{t} \rangle} \rangle \rightarrow \langle \overline{\mathcal{U}}, \overline{\langle \overline{\mathcal{R}}, \bar{t}' \rangle} \rangle \dots$$

This interesting and powerful concept is supported by Maude through a built-in module called `META-LEVEL`. This module has sorts `Term` and `Module` such that the

representation \bar{t} of a term t is of sort **Term** and the representation \overline{SP} of a specification SP is of sort **Module**. There are also functions like `metaReduce(\overline{SP} , \bar{t})` which returns the representation of the reduced form of a term t using the equations in the module SP .

META-LEVEL module can be extended by the user to specify strategies of controlling the rewriting process. We use **META-LEVEL** in order to define the “maximal parallel rewriting” strategy. Forced to execute parallel steps on a sequential machine, we discover a new dimension (given by **META-LEVEL**) of the membrane computing, and provide an algorithmic description (given by `maxParRew`) of the rather awkward and ambiguous “nondeterministic and maximal parallel” applications of evolution rules in P system. Using `maxParRew` as a transition step between meta-level configurations, we then provide an operational semantics of the P systems. Using the power given by the tower of reflection levels in Maude, we define operations over modules and strategies to guide the deduction process, and finally use meta-metalevel to analyze and verify the properties of the P systems.

For our sequential thinking, such a clarifying and conceptual description of the P systems is more than helpful, and it could become a useful framework for further investigations.

3 P Systems Specification in Maude

Each P system Π is naturally represented as a collection of Maude system modules such that each membrane M_i is represented by a Maude system module denoted also by M_i . We use the sort **Obj** for the object names, the sort **Soup** for the multisets of objects, and the sort **Config** for the states of a P system. We have $\mathbf{Obj} < \mathbf{Soup}$ (each object defines a particular multiset). The complete meaning of these sorts is given by the following operations:

$$\begin{aligned} _ _ &: \mathbf{Soup} \times \mathbf{Soup} \rightarrow \mathbf{Soup} \\ \langle _ \mid _ \rangle &: \mathbf{Qid} \times \mathbf{Soup} \rightarrow \mathbf{Config} \\ _ , _ &: \mathbf{Config} \times \mathbf{Config} \rightarrow \mathbf{Config} \\ \langle _ \mid _ ; \langle _ \rangle \rangle &: \mathbf{Qid} \times \mathbf{Soup} \times \mathbf{Config} \rightarrow \mathbf{Config} \end{aligned}$$

The operation $_ _$ is declared to satisfy the structural laws of associativity, commutativity, and it has an identity `empty`. The operation $_ , _$ is declared to satisfy only the structural laws of associativity and commutativity. An expression of the form $\langle M \mid S \rangle$ represents a configuration corresponding to an elementary membrane M in state S and an expression of the form $\langle M \mid S; C_1, \dots, C_n \rangle$ represents a configuration corresponding to a composite membrane M in state S and with the component i having the configuration C_i .

We have not in Maude a built-in relation able to describe the operational semantics for P systems. Therefore we have to supply ourselves a definition for such a relation. Such a definition is depending on the rewriting rules included in the Maude description of the P system and therefore it must be defined at meta-level (see Figure 1). We consider the following maximal parallel application of rules: in a transition step, the rules of each membrane are used against its resources until no more rules can

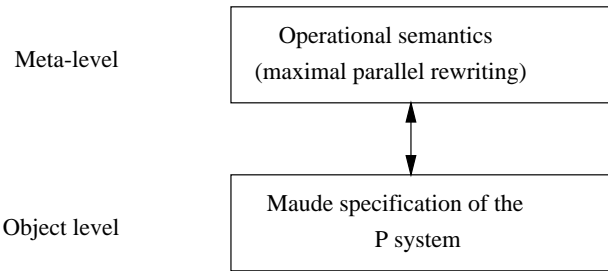


Figure 1: Operational semantics is defined at meta-level

be applied. In this way it is not guaranteed that we obtain the maximal number of rules over the existent multiset because of the nondeterministic choice of rules realized in Maude. Therefore “maximal” means in fact “until no more rules can be applied”. The one step maximal parallel rewriting corresponding to an elementary membrane M_i is given by means of a function:

$$\text{maxParRewS} : \text{RuleSet} \times \text{Term} \rightarrow [\text{Term}]$$

representing at meta-level the function maxParRewS ($\text{maxParRewS} = \overline{\text{maxParRewS}}$) given by:

$$\begin{aligned}
 \text{maxParRewS}(\emptyset, S) &= S \\
 \text{maxParRewS}(R, S) &= r \text{maxParRewS}(R, S_1) \\
 &\quad \text{if } (\ell \rightarrow r) == \text{choose}(R) \wedge (\exists S_1) S == \ell S_1 \\
 \text{maxParRewS}(R, S) &= \text{maxParRewS}(R \setminus \{\ell \rightarrow R\}, S) \\
 &\quad \text{if } (\ell \rightarrow r) == \text{choose}(R) \wedge \neg(\exists S_1) S == \ell S_1
 \end{aligned}$$

where $\text{choose}(R)$ returns an arbitrary rule depending on priorities (if any). It is easy to see that $S \Rightarrow \text{maxParRewS}(R, S)$ provided that R is the set of rules of the membrane described by M_i . The Maude description of maxParRewS is given by:

```

eq maxParRewS(none, T) = T .
cq maxParRewS(RS, T) = if(MP :: MatchPair)
    then '_,_[Y, maxParRewS(RS, getContext(MP))]
    else maxParRewS(RS1, T)
    fi
if (r1 X => Y [label(Q)] .) RS1 := choose(RS) RS1 ^
MP := metaXmatch(up(Mi), X, T) .
  
```

where $::$ is the membership predicate, and $:=$ is the matching operator. The function $\text{metaXmatch}(\text{up}(M_i), X, T)$ used in the conditional part checks if X matches T in the module M_i . In our case, MP is of sort MatchPair iff the lhs of the rule is a subterm of T , and in that case it is a pair consisting of the empty substitution (lhs has no variables) and the context. The function $\text{getContext}(MP)$ extracts the context from MP . Since we are working over multisets, the subterm to be processed by maxParRewS is the context after removing the placeholder $[_]$. There is a function called $\text{getRls}(\text{up}(M_i))$ which returns the set of rules included in a module M . $\text{up}(M_i)$ is the representation of the module M_i at metalevel, i.e., $\text{up}(M_i)$

= $\overline{M_i}$. In order to find out a successor of a multiset S in M_i with respect to the maximal parallel rewriting, we compute at meta-level the normal form of the term $\text{maxParRewS}(\text{getRls}(\text{up}(M_i)), \text{up}(M_i, S))$.

We extend maxParRewS to maxParRew defined over configurations:

$$\begin{aligned} \text{maxParRew}\langle M|S \rangle &= \langle M|\text{maxParRewS}(\text{rules}(M), S) \rangle \\ \text{maxParRew}\langle M|S; C \rangle &= \langle M|\text{maxParRewS}(\text{rules}(M), S); \text{maxParRewS}(C) \rangle \\ \text{maxParRew}\langle C_1, C_2 \rangle &= \text{maxParRewS}(C_1), \text{maxParRewS}(C_2) \end{aligned}$$

The Maude description of the function maxParRew is given at meta-level by

`maxParRew : Term -> [Term]`

defined as follows:

```

cq maxParRew('<_|_>[X , Y]) = '<_|_>[X, maxParRewS(getRls(up(Mi), Y)]
   if getType(metaReduce(m, Y)) == 'Soup .
cq maxParRew('<_|_>[X , Y]) = '<_|_>[X, maxParRewS(getRls(up(Mi), Y)]
   if getType(Y) == 'Obj .
eq maxParRew('<_<_>[_[X , Y]) = '<_<_>[_[maxParRew(X), maxParRew(Y)] .
eq maxParRew('<_|_>[_>[X , Y, Z]) =
   '<_|_>[_>[X, maxParRewS(getRls(up(Mi), Y), maxParRew(Z)] .

```

where `'Soup` is the representation at meta-level of `Soup` and so on. The function $\text{getType}(\overline{S})$ returns the sort of S represented at meta-level. Note that the prefix notation is used at meta-level, and the parameters are written between square brackets.

The Maude operational semantics of a P system Π is the rewriting specification (Σ, E, R) , where (Σ, E) is $\overline{M}(\Pi)$ together with the definitions of functions `choose`, `maxParRew()`, `maxParRew()`, and R consists of the following rewriting rule:

`cr1 rwf(X) => rwf(maxParRew(X)) if (X :: Term)`

where `rwf : Term -> Term` is an auxiliary operation used to avoid infinite rewritings of the form $X \Rightarrow \text{maxParRew}(X) \Rightarrow \text{maxParRew}(\text{maxParRew}(X)) \Rightarrow \dots$. Since the result sort of `maxParRewS` is `[Term]`, a special supersort associated with the sort `Term` called *kind*, the condition allows to apply the rule only if the strategy described by `maxParRew` was completely applied.

We can use the Maude commands like `rew` or `search` to verify local properties concerning the behaviour of a P system. Sometimes we need more than that. For instance, we have to read the “result” from a configuration obtained after a number of computation steps. This can be done using meta-commands like `getTerm` and `metaRewrite`. This means that we have to work at meta-metalevel (see Figure 2). This will be exemplified in the next section.

If Π is nondeterministic, then the definition of the maximal parallel rewriting is strongly dependent on how the function `choose()` is defined. Moreover, the function `choose()` is suitable for simulations, but it is not suitable for verification when the whole state space must be generated. An alternative to avoid the use of `choose()`

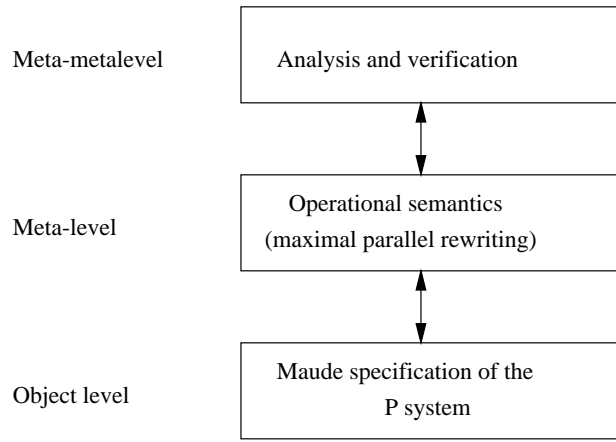


Figure 2: Analysis and verification can be done at meta-metalevel

is by replacing the equational definition by a conditional rewriting system:

$$\begin{aligned}
 [\text{ps1}] : \text{maxParRewS}(R, S) &\rightarrow r \text{ maxParRewS}(R, S, \ell \rightarrow r) \\
 &\text{ if } (\ell \rightarrow r) \text{ is a rule in } R \text{ applicable to } S \\
 [\text{ps2}] : \text{maxParRewS}(R, S, \ell \rightarrow r) &\rightarrow \text{maxParRewS}(R, r S_1) \\
 &\text{ if } S == \ell S_1 \\
 [\text{ps3}] : \text{maxParRewS}(R, S) &\rightarrow S \text{ if } S \text{ is irreducible}
 \end{aligned}$$

The first rule plays the role of function `choose()`, but now the search commands and the model checker will generate all the successors of a multiset. The three rules above are implemented in Maude at meta-level as follows:

```

crl maxParRewS(RS, T) =>
  maxParRewS(RS, (rl X => Y [label(Q)] .), MP, T)
  if (RS1 rl X => Y [label(Q)] .) := RS /\
    MP := metaXmatch(m, getTerm(metaReduce(m, X)),
      getTerm(metaReduce(m,T)), nil, 0, unbounded, 0) /\
    MP :: MatchPair .
crl maxParRewS(RS, R, MP, T) =>
  getTerm(metaReduce(m, '__[Y, maxParRewS(RS, toTerm(getContext(MP)))]))
  if (rl X => Y [label(Q)] .) := R .
crl maxParRewS(RS, T) => maxParRewS(RS1 RS2, T)
  if (RS1 rl X => Y [label(Q)] . RS2) := RS /\
    MP := metaXmatch(m, getTerm(metaReduce(m, X)),
      getTerm(metaReduce(m,T)), nil, 0, unbounded, 0) /\
    not (MP :: MatchPair) .
rl maxParRewS(none, T) => T .
  
```

The definition of *maxParRew* is modified in a similar way:

$$\begin{aligned}
 [\text{ps4}] : \text{rwf}(C) &\rightarrow \text{rwf}(\text{maxParRew}(C)) \\
 [\text{ps5}] : \text{maxParRew}(\langle M|S \rangle) &\rightarrow \langle M|\text{maxParRewS}(\text{rules}(M), S) \rangle \\
 [\text{ps6}] : \text{maxParRew}(\langle M|S; C \rangle) &\rightarrow \langle M|\text{maxParRewS}(\text{rules}(M), S); \text{maxParRew}(C) \rangle \\
 [\text{ps7}] : \text{maxParRew}(C_1, C_2) &\rightarrow \text{maxParRew}(C_1), \text{maxParRew}(C_2)
 \end{aligned}$$

and its Maude implementation at meta-level is:

```

cr1 rwf(X) => rwf(maxParRew(X)) if (X :: Term) .
cr1 maxParRew('<_|_>[X , Y]) =>
  '<_|_>[X, maxParRewS(getQRls(getRls(m), X), Y)]
  if getType(metaReduce(m, Y)) == 'Soup or
    getType(metaReduce(m, Y)) == 'Obj .
r1 maxParRew('<_<_>[_ , Y]) => '<_<_>[_ , maxParRew(X), maxParRew(Y)] .
r1 maxParRew('<_|_>[_ , Y, Z]) =>
  '<_|_>[_ , maxParRewS(getQRls(getRls(m), X), Y), maxParRew(Z)] .

```

A maximal parallel rewriting is defined now by $\text{rwf}(X : \text{Term}) \Rightarrow^+ \text{rwf}(Y : \text{Term})$, where the intermediate terms are of kind `[Term]` and consequently, ignored in the definition of the operational semantics for P systems. Note that \Rightarrow^+ is the transitive closure of \Rightarrow .

In order to cope with priorities and dissolutions we have to add some ingredients in the soup, namely two subsorts `Priority` and `Dissolve` of the sort `Soup`. We also consider a partial order $<$ over the sort `Priority` with the least element \perp . We define a function `dissolve()` which finds the dissolving objects and dissolve the corresponding subconfiguration into the surrounding configuration:

$$\begin{aligned}
\text{dissolve}(\langle M_i | S_i ; \langle M_j | S_j \delta \rangle, C \rangle) &= \langle M_i | S_i S_j ; \text{dissolve}(C) \rangle \\
\text{dissolve}(\langle M_i | S_i ; \langle M_j | S_j \rangle, C \rangle) &= \langle M_i | S_i ; \langle M_j | S_j \rangle, \text{dissolve}(C) \rangle \\
&\text{if } \delta \text{ does not occur in } S_j \\
\text{dissolve}(\langle M_i | S_i ; \langle M_j | S_j \delta ; C_1 \rangle, C_2 \rangle) &= \langle M_i | S_i S_j ; \text{dissolve}(C_1), \text{dissolve}(C_2) \rangle \\
\text{dissolve}(\langle M_i | S_i ; \langle M_j | S_j ; C_1 \rangle, C_2 \rangle) &= \langle M_i | S_i ; \langle M_j | S_j ; \text{dissolve}(C_1) \rangle, \text{dissolve}(C_2) \rangle \\
&\text{if } \delta \text{ does not occur in } S_j
\end{aligned}$$

Note that if S_j contains more δ objects, then all these objects must be removed. This will be done a function called `clean()`. We give as example the Maude implementation of the first rule:

```

ceq dissolve('<_|_>[_ , Y, '<_|_>[_ , V ]]) =
  if (MP :: MatchPair)
    then '<_|_>[_ , getTerm(metaReduce(m,
      '__[_ , clean(toTerm(getContext(MP)))))]
    else '<_|_>[_ , Y, '<_|_>[_ , V ]]
  fi
if MP := metaXmatch(m, 'delta.Dissolve, getTerm(metaReduce(m, V)),
  nil, 0, unbounded, 0) .

```

The priorities are handled with the mean of the predicate $\text{isMaxPriority}(R, \ell \rightarrow r, S)$ which returns `true` iff there is no a rule $(\ell' \rightarrow r') \neq (\ell \rightarrow r)$ in the set R which can be applied to the multiset S and has the priority greater than the rule $\ell \rightarrow r$. The definition of this predicate is:

$$\begin{aligned}
\text{isMaxPriority}(\emptyset, \ell \rightarrow r, T) &= \text{false} \\
\text{isMaxPriority}(\{\ell' \rightarrow r'\}, \ell \rightarrow r, T) &= \text{true} \\
&\text{if } \text{pri}(r') \leq \text{pri}(r) \vee \ell' \rightarrow r' \text{ is not applicable to } S \\
\text{isMaxPriority}(R \cup \{\ell' \rightarrow r'\}, \ell \rightarrow r, T) &= \text{false}
\end{aligned}$$

if $\ell' \rightarrow r'$ is applicable to $S \wedge \text{pri}(r) < \text{pri}(r')$
 $\text{isMaxPriority}(R \cup \{\ell' \rightarrow r'\}, \ell \rightarrow r, T) = \text{isMaxPriority}(R, \ell \rightarrow r, T)$
 otherwise

where $\text{pri}(r)$ returns the priority ingredient from r (if any); otherwise it returns \perp .

The second and the fourth rules from the rewriting system defining the operational semantics of P systems are modified as follows:

$[\text{ps2}] : \text{maxParRewS}(R, S, \ell \rightarrow r) \rightarrow \text{maxParRewS}(R, \text{clean}(r) S_1)$
 if $S == \ell S_1$
 $[\text{ps4}] : \text{rwf}(C) \rightarrow \text{rwf}(\text{dissolve}(\text{maxParRew}(C)))$

The other rules remains unchanged.

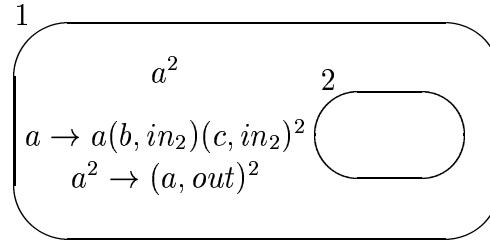
4 Example of a P System Specified in Maude

In this section we consider a simple example of P system, and then describe and execute its Maude specification.

Example: Consider a P system generating numbers of the form $6n$, $n \geq 0$:

$$\begin{aligned}
 \Pi_1 &= (O, \mu, w_1, w_2, R_1, R_2, i_o), \\
 O &= \{a, b, c\}, \\
 \mu &= [{}_1[{}_2]_2]_1, \\
 w_1 &= a^2, \\
 w_2 &= \lambda, \\
 R_1 &= \{a \rightarrow a(b, \text{in}_2)(c, \text{in}_2)^2, a^2 \rightarrow (a, \text{out})^2\}, \\
 R_2 &= \emptyset, \\
 i_o &= 2.
 \end{aligned}$$

The initial configuration is:



In Maude, we firstly define the modules for the object names and configurations:

```

(fmod OBJ is
  sorts Out Obj .
  subsort Out < Obj .
  op a : -> Obj .
  ops b c : Out .
)

```

```

endfm)

(fmod CONFIG is
  inc OBJ .
  pr QID .
  sorts Soup Config .
  subsort XObj < Soup .
  op empty : -> Soup .
  op _ : Soup Soup -> Soup [assoc comm id: empty] .
  op <_|_> : Qid Soup -> Config .
  op <_|_;> : Qid Soup Config -> Config .

  op _',_ : Config Config -> Config [assoc] .
endfm)

```

Then we give the Maude descriptions of the membranes, each membrane being specified by an independent system module. Actually, we ignore the membrane labelled by 2 and consider that Π_1 consists only of the skin:

```

(mod SKIN is
  inc CONFIG .
  op init : -> Soup .
  eq init = a a .
  rl ['SKIN] : a => a b c c .
  rl ['SKIN] : a a => empty .
endm)

```

Generally speaking, the Maude specification of a P system consists of a system module importing the modules corresponding to the component membranes and defining the initial configuration. We note that the configuration includes also the structure of the system. The Maude module describing Π_1 is:

```

(mod PSYS is
  inc SKIN .
  op initConf : -> Config .
  eq initConf = < 'SKIN | init > .
endm)

```

The Maude module defining the operational semantics of P systems is called OPSEMPS. We can use various Maude commands in order to make experiments with the P system specification. For instance, we use the command `rew` to see the result obtained after ten steps of maximal parallel rewritings:

```

Maude> (select OPSEMPS .)
rewrites: 29 in 10ms cpu (0ms real) (2900 rewrites/second)

Maude> (down PSYS : rew [10] getTerm(metaReduce(up(PSYS),
                                                up(PSYS, initConf))) .)
rewrites: 5755 in 130ms cpu (160ms real) (44269 rewrites/second)
result Config :
< 'SKIN | a a b b b b b b b b b b b b b b b b b b c c c c
          c c c c c c c c c c c c c c c c c c c c c c c c
          c c c c c c c c c c >

```

We consider a module METAPS defining a filtering function `out`, which removes the non-output objects, and a counting function `#()`, which counts the occurrences of an object into a configuration. Since we wish to apply these functions to configurations obtained with `metaRewrite` command, the module METAPS is defined at

meta-metalevel. The filtering function simulate somehow the membrane labelled by 2. For instance, we can check that the number of objects *c* is double of the number of objects *b*:

```
(mod PROOF is
  inc METAPS .
  op ql : -> QidList .
  eq ql = out(...(metaRewrite(up(OPSEMPS),up(PSYS,initConf))),100))) .
endm)
Maude> (red #(ql, ''c) == 2 * #(ql, ''b) .)
rewrites: 29996 in 1170ms cpu (1170ms real) (25637 rewrites/second)
reduce in PROOF :
  #(ql, ''c) == 2 * #(ql, ''b)
result Bool :
  true
```

This example uses one hundred maximal parallel rewritings. In the next section we show how this property can be checked for all the configurations, using the temporal formulas and a model checker implemented in Maude.

5 Model Checking P Systems

A Maude module is essentially a mathematical object. Therefore, we can directly use all the tools of mathematics and logic, including automatic or semiautomatic tools, to reason about the correctness of a Maude module. The Maude interpreter is a high-performance logical engine to prove logical facts about our theories (programs). Moreover, Maude has a collection of formal tools supporting different forms of logical reasoning to verify program properties, including a model checker to verify temporal properties of finite-state system modules. Therefore, once we have a Maude description of the membrane systems, we may use the Maude implementation of Linear Temporal Logic (LTL) to verify various properties expressed in LTL [5]. The LTL model checker provides a powerful tool to detect subtle errors and to verify some desired temporal properties.

LTL was designed for expressing the temporal ordering of events. In computer science, temporal logics keep track of the systems states, changes of the variable values, and the order in which they occur. Intuitively, the system state is a snapshot of the system's execution. In this snapshot, every variable has some value. A particular execution of the system is represented by a sequence of system states, and obviously, time progresses during the execution, but there is no keeping track of how long the system is in any particular state. In LTL, formulas are evaluated with respect to a particular execution and a particular state in that execution. Time is totally ordered, usually bounded in the past and unbounded in the future. LTL formulas allows boolean connectives and modalities together with the operators **X** expressing “in the next state”, and **U** expressing that one property holds until another holds (Until). These formulas are evaluated on individual executions (computation paths). The modal path operators are \diamond and \square ; \diamond expressing “at some future time”, and \square expressing “at all future times”. They are also found as the letters **F** and **G**. More information on LTL, and its link to Maude can be found in [5].

In order to apply the model checker, the space of the reachable states of the P system must be finite. Unfortunately, this requirement is not fulfilled by almost all the P systems. Therefore we have to consider a finite subspace of reachable states. An example is the subspace including the states with the size less than, or equal to, a given value. We consider a function $\#(C)$ which returns the size of the configuration C . The rewriting system describing the operational semantics is modified as follows:

$$\begin{aligned} \text{rwf}(X) &\rightarrow \text{rwf}(\text{maxParRew}(X)) \text{ if } \#(X) < \text{givenSize} \\ \text{rwf}(X) &\rightarrow X \text{ if } \#(X) \geq \text{givenSize} \end{aligned}$$

For `givenSize = 20`, the `rewrite` command generates a finite number of rewritings (maximal parallel rewriting steps) for Π_1 :

```
Maude>(down PSYS : rew rwf(getTerm(metaReduce(up(PSYS),
                                                up(PSYS, initConf)))) .)
rewrites: 4246 in 110ms cpu (100ms real) (38600 rewrites/second)
result Config :
  < 'SKIN | a a b b b b b b c c c c c c c c c c c c c c c c >
Maude>
```

The function `out`, removing the non-output objects, and the function $\#(_,_)$, counting the number of occurrences of an object in a soup, are defined now at meta-level instead of meta-metalevel.

We wish to check if the number of objects `c` is double the number of the objects `b` for all the configurations with the size less than or equal to `givenSize` in Π_1 . We define an atomic proposition `isDouble` which is satisfied by a configuration iff the number of objects `c` is double the number of the objects `b` in it:

```
(mod PI1-PREDS is
  including OPSEMPS .
  including SATISFACTION .
  subsort Term < State .
  ops isDouble : -> Prop .
  var T : Term .
  ceq T |= isDouble = true if #(out(T), 'c) == 2 * #(out(T), 'b) .
endm)
```

`SATISFACTION` is a built-in module which includes the specifications for the atomic linear temporal formulas and the satisfaction relation between states and propositions. We then define the initial state against which we wish to check the temporal formula $\Box \text{isDouble}$:

```
mod PROOF is
  including CC-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  op init : -> Term .
  eq init = rwf('<_>[ 'SKIN.Qid, '[_['a.Obj, 'a.Obj]]) .
endm
```

The formula is verified by giving the `red` command:

```
red modelCheck(init, [] (isDouble) ) .
```

and Maude supplies the following output:

```
Maude> red modelCheck(init, [] (isDouble)) .
reduce in PROOF : modelCheck(init, (isSingle -> isWhite)) .
rewrites: 591 in 60ms cpu (70ms real) (~ rewrites/second)
result Bool: true
```

6 Conclusion

The contributions of this paper consist in providing executable specifications of P systems, using a complex software system based on rewriting. Moreover, it is presented for the first time the use of a software verification tool able to automatically check properties of a P system. The approach fully exploits the reflection property of the rewriting logic, property which allows a meta-level implementation of the P systems operational semantics. It is also presented an algorithmic description of the nondeterministic maximal parallel evolution rules in P systems.

The paper does not present the rich theoretical worlds of P systems and rewriting logic, respectively. It rather concentrates on fruitful bridge between these two worlds, emphasizing the use of Maude as a complex tool able to execute specifications of the P systems, and then to verify the desired properties of the specified P system. Since this is the first paper describing Maude executable specifications for the P systems, we have used a simple example.

References

- [1] G. Ciobanu, D. Paraschiv. P System Software Simulator. *Fundamenta Informaticae* vol.49, 61-66, 2002.
- [2] G. Ciobanu, W. Guo. P Systems Running on a Cluster of Computers. In Gh.Păun, G.Rozenberg, A.Salomaa (Eds.): *Membrane Computing*, LNCS vol.2933, Springer, 123-139, 2004.
- [3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, J.F. Quesada. Maude: Specification and Programming in Rewriting Logic. *Theoretical Computer Science*, vol.285(2), 187-243, 2002.
- [4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, C .Talcott. *Maude Manual* (Version 2.1). <http://maude.cs.uiuc.edu>, 2004.
- [5] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and Its Implementation. In T.Ball, S.K.Rajamani (Eds.): *Model Checking Software: 10th SPIN Workshop*, LNCS vol.2648, Springer, 230-234, 2003.
- [6] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud. Introducing OBJ. In *Software Engineering with OBJ: Algebraic Specification in Action*, 3-167, Kluwer, 2000.
- [7] Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences* vol.61, 108-143, 2000.
- [8] Gh. Păun. *Computing with Membranes: An Introduction*, Springer, 2002.