

Specification and Execution of P Systems with Symport/Antiport Rules Using Rewriting Logic

Zhengwei QI¹, Cheng FU², Dongyu SHI², Jinyuan YOU²

Department of Computer Science and Engineering
Shanghai Jiao Tong University, Shanghai 200030, P.R. China
E-mail:¹qizhwei@sjtu.edu.cn, ²{fucheng,shi-dy,you-jy}@cs.sjtu.edu.cn

Abstract

Rewriting logic is a unified model of concurrency which provides a formal common framework of well-known models of concurrent systems. This paper proposes the specification and execution of P systems with symport/antiport rules using rewriting logic. We use the powerful tool Maude 2.0 to implement this specification and solve the problem of the infinite copies of objects in the environment. In order to present the general ideas in a concrete case study, we have represented how to specify and execute a simple and classical example from the literature. In this way, it is easy to extend this method to general P systems and we can borrow concepts and tools in rewriting logic to study P systems.

1 Introduction

Rewriting logic [1, 2] was introduced in [3] as a unified model of concurrency which provides a formal common framework of well-known models of concurrent systems. In [2], it is shown that rewriting logic can be used as a logical and semantic framework to represent many other logics, in a natural and direct way. In general, every formal system has two parts: (1) formulas or proof-theoretic structures, which can be presented as terms in an order-sorted equational data type whose equations express structural axioms natural to the logic in question; (2) the rules of deduction of a logic transforming certain patterns of formulas into other patterns, which can be represented by rewriting rules. In this way, an inference rule of the form $S_1, \dots, S_n \rightarrow S_0$ rewrites *multisets* of judgements S_i to S_0 . The above techniques can be used to specify and prototype a wide variety of inference systems. For instance, the following models of computation have been naturally expressed in rewriting logic [2]: (1) Petri nets, including place/transition nets, contextual nets, colored nets, and algebraic nets; (2) Process Algebra, including λ -calculi, CCS, and π -calculus.

P systems have been introduced by Gheorghe Păun in 1998 [11] as a distributed and parallel computability model based on biological membranes. The study of P systems is very active and has already been well developed at the theoretical level. Many variants of P systems have been researched through introducing the basic models from DNA computing, biochemistry, etc. A special variant has been proposed in [5], which introduces in P systems a form of communication based on a

biochemical transport of objects called symport/antiport mechanism. P systems with symport/antiport rules of a minimal size (only one object passes in any direction in a communication step) have been recently proved to be computationally universal. How to study other properties of this kind of P systems? The first step is to specify and execute them. In [12, 13, 14], the formalization of P systems has been discussed in detail. Petri Nets play an important role in the modeling, analysis and verification of parallel systems, hence it is natural to formalize P systems by Petri Nets [4]. Petri Nets can be represented by rewriting logic as we discussed above. This paper continues this direction of research and specifies and executes P systems with symport/antiport rules using rewriting logic. The reason why we use this kind of P systems is based on two considerations: (1) P systems with symport/antiport rules are relatively simple and many articles are devoted to them. (2) This kind of P systems raise a special problem, i.e., the existence of infinitely many copies of objects in environment is difficult to be formalized in current formal methods.

The rest of this paper is organized as follows. In Section 2, we provide the background and preliminaries about P systems and rewriting logic. In Section 3, we use Maude [6], a famous executable tool of rewriting logic, to describe the specification of P systems with symport/antiport rules. Finally, we conclude our work and present future research directions in the last section.

2 Preliminaries

2.1 Membership Equational Logic and Rewriting Logic

Membership Equational Logic (MEL) [7] is a many-sorted logic with subsorts and overloading of function symbols. In accordance with the terminology we refer to the *types* in the logic as *kinds*, and we view the sorts for each kind as unary predicates. The atomic sentences are equalities $M = N$ for terms M, N of the same kind, and memberships $M : s$ for M a term and s a sort, both of the same kind. Sentences of MEL are universally quantified Horn clauses on the atoms. In contrast to an entirely loose or entirely initial semantics of membership equational theories, in practice a mixed specification style is used, where certain subtheories are intended to be equipped with initial interpretations, or they are interpreted freely over their parameter specifications. To make such restrictions on the models explicit in the specification we enrich a membership equational theory by initiality and freeness constraints, and refer to this enriched theory as a *membership equational logic specification* (MES).

The general rewriting logic is based on MEL [2].

Definition 2.1 *A Rewrite Logic is a 4-tuple, $\mathfrak{R} = (\Sigma, E, \Phi, R)$, where:*

- (i) (Σ, E) is a membership equational theory, with kinds K , sorts S and operations Σ ;
- (ii) $\Phi : \Sigma \longrightarrow \mathcal{P}_{fin}(N)$ is a $K^* \times K$ -indexed family of functions assigning to each $f : k_1 \dots k_n \rightarrow k$ in Σ the finite set $\Phi(f) \subseteq \{1, \dots, n\}$ of its *frozen* argument positions;

- (iii) R is a set of (universally quantified) labelled conditional rewrite rules of the form (with t, t' and w_k, w'_k pairs of terms of same kind)

$$l : t \longrightarrow t' \Leftarrow \left(\bigwedge_i u_i = u'_i \right) \wedge \left(\bigwedge_j v_j : s_j \right) \wedge \left(\bigwedge_k w_k \rightarrow w'_k \right) \quad (1)$$

A more detailed definition can be found in [8].

2.2 P systems with Symport/Antiport rules

The basic elements of a P system (or membrane system) consist of several membranes which form the membrane structure (a tree-like structure of several membranes embedded in a main membrane called the skin membrane). In every membrane, there exist some objects and some evolving rules on these objects.

In general, the evolution rules allow the system to modify the objects in the various regions of the system and communicate them among the compartments. A special variant has been proposed in [5], which introduces in P systems a form of communication based on a biochemical transport of objects called symport/antiport mechanism. When two objects can pass through a membrane only together, in the same direction, the process is called symport; when the two chemicals can pass only with the help of each other, but in opposite directions, we say that we have an antiport process (see [10]). Such a mechanism has been formalized and generalized in P systems by considering rules of the form $(x, in), (x, out)$ (symport rules), and $(x, out; y, in)$ (antiport rules), where x, y are multisets of arbitrary size.

3 Specification and Execution of P systems with Symport/Antiport

3.1 The Maude 2.0 System

Rewriting logic has been proved to have good properties as a semantic and logic framework. The Maude 2.0 (<http://maude.cs.uiuc.edu>) [6] supports both equational and rewriting logical computation with high generality and expressiveness, yet without compromising performance. There are two kinds of modules in Maude. The first is the functional module which represents membership equational theories. The second is the system module, which is a general rewrite theories whose rules are equations, memberships and rewrites in their conditions, and where some operator arguments can be *frozen* to block undesired rewrites. Furthermore, because rewriting logic is suitable for specifying concurrent systems, Maude 2.0 supports efficient explicit-state model checking of Linear Temporal Logic (LTL) properties satisfied by finite-state rewrite theories. The Maude LTL Model Checker supports two different levels of specification: (1) a system specification level, in which the concurrent system to be analyzed is formalized; and (2) a property specification level, in which the properties to be model checked are specified. Because of above advantages, Maude is very suitable to specify and execute P systems. In Maude formal environment, we have a set of powerful tools to study the properties of complex P systems. For

example, there are many useful tools, such as (1) an inductive theorem prover; and (2) Church-Rosser, coherence, and termination checkers [2].

3.2 Specification of P Systems with Symport/Antiport in Maude

Firstly, the formal definition of P systems with symport/antiport is given as follows [10].

Definition 3.1 *A P system (of degree $m \geq 1$) with symport/antiport rules is a construct*

$$\Pi = (V, \mu, w_1, w_2, \dots, w_m, E, R_1, \dots, R_m, i_O),$$

where:

- (i) V is a finite alphabet of symbols called objects,
- (ii) μ is a membrane structure consisting of m membranes that are labelled in a one-to-one manner by $1, \dots, m$,
- (iii) $w_i \in V^*$, for each $1 \leq i \leq m$ is a multiset of objects,
- (iv) $E \subseteq V$ is the set of objects that appear in the environment,
- (v) R_i , for each $1 \leq i \leq m$, is a finite set of symport/antiport rules associated with the region i that are of the form (x, in) , (x, out) , $(x, out; y, in)$, with $x, y \in V^*$,
- (vi) i_O is the label of an elementary membrane of μ that identifies the corresponding output region.

In order to illustrate our specification of P systems, we use a simple P system from [10] as an example.

$$\Pi = (\{a, b\}, [1[2]2]_1, a, \lambda, \{b\}, R_1, R_2, 2),$$

where $R_1 = \{(a, out; b, in), (a, in)\}$, $R_2 = \{(a, in), (b, in)\}$.

This is a simple P system which is able to generate the whole set of natural numbers (zero excluded). Firstly, the system module of this P system called MEMBRANE-SYSTEM is defined as follows.

```

mod MEMBRANE-SYSTEM is
  protecting INT .
  protecting QID .
  sorts Obj Rule Mem .
  subsorts Obj Rule < Mem .
  subsorts Qid < Obj .
  op nul : -> Obj .
  op __ : Obj Obj -> Obj [assoc comm id: nul] .
  op none : -> Rule .

```

```

op |_| : Rule Rule -> Rule [assoc comm id: none] .
op <_,in> : Obj -> Rule .
op <_,out> : Obj -> Rule .
op <_,in;_,out> : Obj Obj -> Rule .
op 0 : -> Mem .
op __ : Mem Mem -> Mem [assoc comm id: nul] .
op [_:_,_,_] : Int Obj Rule Mem -> Mem .
vars O1 O2 O3 O4 : Obj .
vars I J K : Int .
vars R1 R2 R3 : Rule .
vars M1 M2 M3 : Mem .
crl [out] :
  [ I : O2, R1, [ J : O3 O1, < O3,out> |
    R2, M1] M2 ]
    => [ I : O3 O2, R1, [ J : O1, < O3 ,out> |
      R2, M1] M2 ]
  if I >= 1 .
crl [in] :
  [ I : O2 O3, R1, [ J : O1, < O3,in> |
    R2, M1] M2 ]
    => [ I : O2, R1, [ J : O1 O3, < O3,in> |
      R2, M1] M2 ]
  if I >= 1 .
crl [inout] :
  [ I : O2 O3, R1, [ J : O1 O4, < O3,in; O4,out> |
    R2, M1] M2 ]
    => [ I : O2 O4, R1, [ J : O1 O3,
      < O3,in; O4,out> | R2, M1] M2 ]
  if I >= 1 .
*** The rules in the skin membrane
***and environment are defined as follows.
rl [ein] :
  [ 0 : 'a O3, none, [ 1 : O1, < 'a,in> | R2, M1]]
  => [ 0 : O3, none, [ 1 : O1 'a, < 'a,in> | R2,M1]].
rl [einout] :
  [ 0 : 'b O3, none, [ 1 : 'a O4, < 'b,in; 'a,out> |
    R2, M1]]
  => [ 0 : 'b 'a O4, none, [ 1 : 'b O3,
    < 'b,in; 'a,out> | R2, M1]] .
endm

```

From this specification, INT and QID are predefined data modules in Maude, which represent the sort *integer* and *identifer*. In the MEMBRANE-SYSTEM module, the sort *Obj* denotes the objects in P system, i.e., 'a, 'b, and 'c are objects in P systems, and *nul* denotes the empty object. The operator (denoted by *op*) *--* is used to combine the multiset of objects, i.e., 'b'b'c is a multiset of objects. The sort

Rule denotes the rules in P systems, and *none* is the empty rule. The operator $_|_$ is used to combine the multiset of rules. The sort *Mem* denotes the structure of P systems, and *O* is the empty membrane. Like the sort *Obj*, the operator $_ _$ denotes the multiset of Membranes.

A membrane is constructed as follows: the operator $[- : _ , _ , _]$ denotes that a membrane has a label, a multiset of Objects, a multiset of Rules, and submembranes. For the above example, the membrane structure is

$$[0 : 'b, none, [1 : 'a, <'b, in; 'a, out > | <'a, in >, [2 : nul, <'a, in > | <'b, in >, O]]]$$

In this notation, the label 0 denotes the environment, Membrane 0 does not have rules, and its submembrane is the skin membrane (denoted by label 1).

The membrane rules have three types. The first is *in* rules.

```

cr1 [in] :
  [ I : O2 O3, R1, [ J : O1, < O3, in > |
    R2, M1] M2 ]
  => [ I : O2, R1, [ J : O1 O3, < O3, in > |
    R2, M1] M2 ]
  if I >= 1 .

```

This rewrite rule will be applied only if $I \geq 1$, which means that this type of rule can be not applied in the environment. This rule says that if there is a rule $\langle O3, in \rangle$ in Membrane *J*, and its parent membrane has the objects *O3*, the objects *O3* will be moved into Membrane *J*. Like this specification, it is easy to present other two rules: *out* and *inout* rules.

Because the environment (Membrane 0) contains no objects in $V - E$ and an infinite number of copies of each object in *E*. In our example, *E* is $\{b\}$. That is, the rule $\langle 'b, in; 'a, out \rangle$ in Membrane 1 is denoted by the rewrite rule *einout*:

```

r1 [einout] :
  [ 0 : 'b O3, none, [ 1 : 'a O4, < 'b, in; 'a, out > |
    R2, M1]]
  => [ 0 : 'b 'a O4, none, [ 1 : 'b O3,
    < 'b, in; 'a, out > | R2, M1]]

```

This rewrite rule will not remove the object *'b* in the environment, which simulates an infinite number of copies of *'b* in *E*.

3.3 Execution of P Systems with Symport/Antiport in Maude

The system module MEMBRANE-SYSTEM provides a general specification of P systems. Now, we can build the run time module to execute the above example. We define another system module called RUN-MEMBRANE-SYSTEM as follows.

```

mod RUN-MEMBRANE-SYSTEM is
  inc MEMBRANE-SYSTEM .

```

```

op test : -> Mem .
eq test =
  [0 : 'b , none, [1 : 'a, < 'b,in; 'a,out> |
    < 'a,in>,[2 : nul, < 'a,in> | < 'b,in>,0]]] .
endm

```

The *test* denotes the above example in Maude 2.0. As already mentioned, this P system generates the whole set of natural numbers (zero excluded). Firstly, we can use the command *rewrite* (or *rew*) and the result is shown as follows.

```

Maude> rew [10] test .
rewrite [10] in RUN-MEMBRANE-SYSTEM : test .
rewrites: 17 in 0ms cpu (0ms real) (~ rewrites/second)
result Mem: [0 : 'b , none,[1 : 'a 'b 'b 'b 'b 'b,< 'b,in;
'a,out> | < 'a,in>,[2 : nul, < 'a,in> | < 'b,in>,0]]] .

```

Unlike *rewrite*, which uses a leftmost outermost strategy for applying rules and reduces the whole term with equations after each successful rule *rewrite*, *frewrite* (or *frew*) attempts to be position fair by making a number of depth-first traversals of the term, and on each traversal, each position that existed at the start of the traversal is entitled to at most number rule rewrites when its turn comes around. Then, if we use the command *frewrite*, we get a termination status, just only one rule $\langle 'a, in \rangle$ in Membrane 2 is applied, which enters the termination. This experiment proceeds as follows.

```

Maude> frew [10, 2] test .
frewrite [10, 2] in RUN-MEMBRANE-SYSTEM : test .
rewrites: 3 in 0ms cpu (0ms real) (~ rewrites/second)
result Mem: [0 : 'b , none,[1 : nul,< 'b,in; 'a,out> |
< 'a,in>,[2 : 'a, < 'a,in> | < 'b,in>,0]]] .

```

From the first experiments, we know that after 17 rewrites, the object *'b* in Membrane 1 has four copies. Of course, we can get any copy of this object. From the second experiment, we use fair rewrites to get a termination status. From the two experiments, we get two important conclusions: (1) this module can be accepted and executed by Maude; (2) the command *rew* and *frew* cannot get all termination results because the default strategy adopted by these two commands is not insufficient for the purposes of a given application. This drawback can be modified by the future version of Maude or the application of META-LEVEL [6] functions in Maude.

Now, our modules can be executed in Maude, and there are many interesting experiments which can be performed. For example, we can use the command *match* to perform straightforward matching in the given module; we can use command *search* to perform a breadth-first search for rewrite proofs starting at subjects to a final state that matches *pattern* and satisfies an optional condition. In addition, there are many useful tools, i.e., the inductive theorem prover, the Church-Rosser, coherence, and termination checkers which can be used in our modules. It is very easy to execute this commands and we ignore it in this paper.

4 Conclusions and Future Work

In this paper, we have shown how P systems with symport/antiport rules can be represented in rewriting logic and its implementation language Maude in a general and fully executable way. In order to present the general ideas in a concrete case study, we have represented the simple and classical example from the literature. We solve the problem of the infinite copies of objects in environment. The rewriting logic has represented many formal systems, such as Petri Nets [9], CCS, and π -calculus. P systems can be formalized in Petri Nets. This paper continues this way and provides the specification and execution of P systems with symport/antiport rules. The next step is to extend this specification for general P systems. Taking into account the power of Maude, it is easy to generalize our specification to many variants of P systems.

From our experiments, it is shown that the default rewrite strategy used by *rewrite* and *frewrite* can not get all desired statuses, especially in a non-terminating fashion. Maude enables the user to create his/her own strategy from within a module, using meta-level reflection and the descent functions. The study of meta-level reflection in our modules is very attractive.

An interesting direction of further work is extend our specification to the LTL model checking [15]. P systems are concurrent systems, and many properties, such as terminating, Church-Rosser, are not hold in such concurrent systems. For properties in non-terminating and concurrent systems, we can use some kind of temporal logic, i.e., LTL. Because of its intuitive appeal, widespread use, and well-developed proof methods in Maude, it is promising to introduce LTL model checking in P systems.

Acknowledgments

This paper is supported by the Shanghai Science and Technology Development Foundation project (No. 03DZ15027) and the State Natural Science Foundation project (No. 60173033).

References

- [1] J. Meseguer, Research Directions in Rewriting Logic, *Computational Logic*, NATO Advanced Study Institute, Marktoberdorf, Germany, 1997. Springer-Verlag, 1998.
- [2] N. Martí-Oliet, J. Meseguer, Rewriting logic: roadmap and bibliography, *Theor. Comput. Sci.* 285(2): 121-154, 2002.
- [3] J. Meseguer, Conditioned Rewriting Logic as a United Model of Concurrency, *Theor. Comput. Sci.* 96(1): 73-155, 1992.
- [4] Z. Qi, J. You, H. Mao, P Systems and Petri Nets, *Lecture Notes in Computer Science*, Vol. 2933, Springer, Berlin. 2003.

- [5] A. Păun, Gh. Păun, The Power of Communication: P Systems with Symport/Antiport, *New Generation Computing* 20: 295-305, 2002.
- [6] M. Clavel, F. Durán, S. Eker, et al., The Maude 2.0 System, In *Proc. Rewriting Techniques and Applications*, 2003. *Lecture Notes in Computer Science*, Vol. 2706, 2003.
- [7] A. Bouhoula, J.-P. Jouannaud, J. Meseguer, Specification and proof in membership equational logic, *Theor. Comput. Sci.* 236(1-2): 35-132, 2000.
- [8] R. Bruni, J. Meseguer, Generalized Rewrite Theories, *Lecture Notes in Computer Science*, Vol. 2719, Springer, Berlin.
- [9] M.-O. Stehr, J. Meseguer, P. C. Ölveczky, Rewriting Logic as a Unifying Framework for Petri Nets, In *Unifying Petri Nets 2001*: 250-303.
- [10] F. Bernardini, A. Paun, Universality of Minimal Symport/Antiport: Five Membranes Suffice, *Lecture Notes in Computer Science*, Vol. 2933, Springer, Berlin, 2004.
- [11] Gh. Păun, Computing with membranes, *J. Comput. System Sci.* 61(1): 108-143, 2000. (See also Turku Center for Computer Science-TUCS Report No. 208, 1998, www.tucs.fi).
- [12] M.J. Perez-Jimenez, F. Sancho-Caparrini, Verifying a P system generating squares, *Romanian J. Inform. Sci. Tech.* 5:181-191, 2000.
- [13] M.J. Perez-Jimenez, F. Sancho-Caparrini, A formalization of transition P systems, *Fund. Inform.* 49:261-272, 2002.
- [14] A.V. Baranda, F. Arroyo, J. Castellanos, and R. Gonzalo, Towards an electronic implementation of membrane computing: A formal description of non-deterministic evolution in transition P systems, *Lecture Notes in Computer Science*, Vol. 2340, Springer, Berlin.
- [15] S. Eker, J. Meseguer, A. Sridharanarayanan, The Maude LTL Model Checker and Its Implementation, In *SPIN 2003*: 230-234, 2003.