International Journal of Computers, Communications & Control
Vol. I (2006), No. 3, pp. 13-24

A Programming Perspective of the Membrane Systems

Gabriel Ciobanu

Abstract: We present an operational semantics of the membrane systems, using an appropriate notion of configurations and sets of inference rules corresponding to the three stages of an evolution step in membrane systems: maximal parallel rewriting step, parallel communication of objects through membranes, and parallel membrane dissolving.
We define various arithmetical operations over multisets in the framework of membrane systems, indicating their complexity and presenting the membrane systems which implement the arithmetic operations.
Finally we discuss and compare various sequential and parallel software simulators of the membrane systems, emphasizing their specific features.
Keywords: membrane systems, operational semantics, arithmetical operations over multisets.

1 Membrane Systems

Membrane systems represent a computational model inspired by cell compartments and molecular membranes. Essentially, such a system is composed of various compartments, each compartment with a different task, and all of them working simultaneously to accomplish a more general task of the whole system. A detailed description of the membrane systems (also called P systems) can be found in [17]. A *membrane system* consists of a hierarchy of membranes that do not intersect, with a distinguishable membrane, called the *skin membrane*, surrounding them all. The membranes produce a delimitation between *regions*. For each membrane there is a unique associated region. Regions contain multisets of *objects*, *evolution rules* and possibly other membranes. Only rules in a region delimited by a membrane act on the objects in that region. The multiset of objects from a region corresponds to the "chemicals swimming in the solution in the cell compartment", while the rules correspond to the "chemical reactions possible in the same compartment". Graphically, a membrane structure is represented by a Venn diagram in which two sets can be either disjoint, or one is a subset of the other. More details (concepts, results) and several variants of membrane systems are presented in [17].

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*. Each membrane has a unique associated region. The space outside the skin membrane is called the *outer region* (or the environment). Because of the one-to-one correspondence between the membranes and the regions, we usually use the word membrane instead of region. Only rules in a region delimited by a membrane act on the objects in that region. Moreover, the rules must contain target indications, specifying the membrane where objects are sent after applying the rule. The objects can either remain in the same region, or pass through membranes in two directions: they can be sent *out* of 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. The membranes can also 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. A *P system* of degree m is a structure $\Pi = (O, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m), i_o)$, where:

(i) O is an alphabet of objects, and μ is a membrane structure;

Selected from ICCCC 2006 (invited paper)

(ii) w_i are the initial multisets over O associated with the regions defined by μ;

(iii) R_i are finite sets of evolution rules over O associated with the membranes, of typical form $u \rightarrow v$, with u a multiset over O and v a multiset containing paired symbols (messages) of the form $(c, here)$, (c, in_j), (c, out) and the dissolving symbol δ;

(iv) ρ_i is a partial order relation over R_i, specifying a *priority* relation among the rules: $(r_1, r_2) \in \rho_i$ iff $r_1 > r_2$ (i.e., r_1 has a higher priority than r_2);

(v) i_0 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.

Since the skin is not allowed to be dissolved, we consider that the rules of the skin do not involve δ. These are the *general P systems*, or *transition P systems*; many other variants and classes were introduced [17].

The existing results regarding the P systems refer mainly to their computation power and complexity, namely to their characterization of Turing computability (universality is obtained even with a small number of membranes, and with rather simple rules), and the polynomial solutions to NP-complete problems by using an exponential workspace created in a "biological way" (e.g., membrane division, string replication). Other types of formal results are given by normal forms, hierarchies, connections with various formalisms.

In this paper we refer to some "programming" aspects of the membrane systems. We first present an operational semantics of the P systems, together with some correctness results. Then we define several arithmetical operations in membrane systems using a natural encoding of numbers. Finally some software simulators of the membrane systems are presented.

2 Structural Operational Semantics

Membrane systems provide an abstract model for parallel systems, and a suitable framework for distributed and parallel algorithms [7]. For each abstract model, theory of programming introduces various paradigms and uses different notions of computations. Turing machines and register machines are related to imperative programming, and λ-calculus is related to functional programming. It is natural to look at the membrane systems from the point of view of programming theory. This means that we define an abstract syntax, and an operational semantics of the membranes systems. The operational semantics of the membrane systems is given in a big-step style, each step representing the collection of parallel steps due to the maximal parallelism principle. A computation is regarded as a sequence of parallel application of rules in various membranes, followed by a communication step and a dissolving step.

The membrane structure and the multisets in Π determine a configuration of the system. We can pass from a configuration to another one by using the evolution rules. This is done in parallel: all objects, from all membranes, which can be the subject of local evolution rules, as prescribed by the priority relation, should evolve concurrently. Since the right hand side of a rule consists only of messages, an object introduced by a rule cannot evolve at the same step by means of another rule. The use of a rule $u \rightarrow v$ in a region with a multiset w means to subtract the multiset identified by u from w, and then to add the objects of v according to the form of the rule. If an object appears in v in the form $(c, here)$, then it remains in the same region. If we have (c, in_j), then c is introduced in the child membrane with the label j; if a child membrane with the label j does not exist, then the rule cannot be applied. If we have (c, out), then c is introduced in the membrane placed immediately outside the region of the rule $u \rightarrow v$. If the special symbol δ appears in v, then the membrane which delimits the region is dissolved; in this way, all the objects in this region become elements of the region placed immediately outside, while the rules of the dissolved membrane are removed.

Let O be a finite alphabet of objects organized as a free commutative monoid O_c^*, whose elements are called *multisets*. Formally, the set of *membranes for a system* Π, denoted by $\mathcal{M}(\Pi)$, and *the membrane structure* are inductively defined as follows:

- if L is a label, and w is a multiset over $O \cup (O_c^* \times \{here\}) \cup (O_c^+ \times \{out\}) \cup \{\delta\}$, then $\langle L \,|\, w \rangle \in \mathcal{M}(\Pi)$; $\langle L \,|\, w \rangle$ is called *simple (or elementary) membrane*, and it has the structure $\langle\rangle$;
- if $M_1, \ldots, M_n \in \mathcal{M}(\Pi)$ with $n \geq 1$, the structure of M_i is μ_i for all $i \in [n]$, L is a label, w is a multiset over $O \cup (O_c^* \times \{here\}) \cup (O_c^+ \times \{out\}) \cup (O_c^+ \times \{in_{L(M_j)} | j \in [n]\}) \cup \{\delta\}$, then $\langle L \,|\, w; M_1, \ldots, M_n \rangle \in \mathcal{M}(\Pi)$; $\langle L \,|\, w; M_1, \ldots, M_n \rangle$ is called *a composite membrane*, and it has the structure $\langle \mu_1, \ldots, \mu_n \rangle$.

A finite set of membranes is usually written as $M_1, \ldots, M_n$. We denote by $\mathcal{M}^+(\Pi)$ the set of non-empty finite sets of membranes. The union of two multisets of membranes $M_+ = M_1, \ldots, M_m$ and $N_+ = N_1, \ldots, N_n$ is written as $M_+, N_+ = M_1, \ldots, M_m, N_1, \ldots, N_n$. An element from $\mathcal{M}^+(\Pi)$ is either a membrane, or a set of sibling membranes.

A *committed configuration* for a membrane system Π is a skin membrane which has no messages and no dissolving symbol δ, i.e., the multisets of all regions are elements in O_c^*. We denote by $\mathcal{C}(\Pi)$ the set of committed configurations for Π, and it is a proper subset of $\mathcal{M}^+(\Pi)$. We have $C \in \mathcal{C}(\Pi)$ iff C is a skin membrane of Π and $w(M)$ is a multiset over O for each membrane M in $\mathcal{C}$.

An *intermediate configuration* is a skin membrane in which we have messages or the dissolving symbol δ. The set of intermediate configurations is denoted by $\mathcal{C}^{\#}(\Pi)$. We have $C \in \mathcal{C}^{\#}(\Pi)$ iff C is a skin membrane of Π such that there is a membrane M in C with $w(M) = w'w''$, $w' \in (Msg(O) \cup \{\delta\})_c^+$, and $w'' \in O_c^*$. By $Msg(O)$ we denote the set $(O^* \times \{here\}) \cup (O^+ \times \{out\}) \cup (O^+ \times \{in_L(M)\})$.

A *configuration* is either a committed configuration or an intermediate configuration. Each membrane system has an initial committed configuration which is characterized by the initial multiset of objects for each membrane and the initial membrane structure of the system.

Each P system has an initial configuration which is characterized by the initial multiset of objects for each membrane and the initial membrane structure of the system. For two configurations C_1 and C_2 of Π, we say that there is a *transition* from C_1 to C_2, and write $C_1 \Rightarrow C_2$, if the following *steps* are executed in the given order:

1. *maximal parallel rewriting step*, consisting of non-deterministically assigning objects to evolution rules in every membrane and executing the rules in a maximal parallel manner;

2. *parallel communication of objects through membranes*, consisting in sending existing messages;

3. *parallel membrane dissolving*, consisting in dissolving the membranes containing δ.

The last two steps take place only if there are messages or δ symbols resulted from the first step, respectively. If the first step is not possible, consequently neither the other two steps, then we say that the system has reached a *halting configuration*. An operational semantics of the P systems, considering each of the three steps, is presented in [2]. We mention here the main results.

We can pass from a configuration to another one by using the evolution rules. This is done in parallel: all objects from all membranes evolve simultaneously according to the evolution rules and their priority relation. The rules of a membrane are using its current objects as much as this is possible in a parallel and non-deterministic way. However, an object produced by a rule cannot evolve at the same step as source of another rule. The use of a rule $u \to v$ in a region with a multiset w has as effect the subtraction of the multiset identified by u from w, followed by the addition of the multiset identified by v.

We denote the *maximal parallel rewriting* on membranes by $\stackrel{mpr}{\Longrightarrow}$ and by $\stackrel{mpr}{\Longrightarrow}_L$ the maximal parallel rewriting over the multisets of objects of the membrane labelled by L (we omit the label whenever it is clear from the context). The rules defining the maximal parallel rewriting use two predicates regarding mpr-irreducibility and (L, w)-consistency.

Proposition 1. Let Π be a membrane system. If $C \in \mathcal{C}(\Pi)$ and $C' \in \mathcal{C}^{\#}(\Pi)$ such that $C \stackrel{mpr}{\Longrightarrow} C'$, then C' is mpr-irreducible.

We denote the *parallel communication relation* by $\stackrel{tar}{\Longrightarrow}$. The rules defining the parallel communication relation use a predicate expressing tar-irreducibility.

Proposition 2. Let Π be a P system. If $C \in \mathcal{C}^{\#}(\Pi)$ with messages and $C \stackrel{tar}{\Longrightarrow} C'$, then C' is tar-irreducible.

We denote the *parallel dissolving relation* by $\stackrel{\delta}{\Longrightarrow}$. The rules defining the parallel dissolving relation use a predicate expressing δ-irreducibility. We note that $C \in \mathcal{C}(\Pi)$ iff C is tar-irreducible and δ-irreducible.

Proposition 3. Let Π be a P system. If $C \in \mathcal{C}^{\#}(\Pi)$ is tar-irreducible and $C \stackrel{\delta}{\Longrightarrow} C'$, then C' is δ-irreducible.

According to the standard description in membrane computing, a *transition step* between two configurations $C, C' \in \mathcal{C}(\Pi)$ is given by: $C \Rightarrow C'$ iff C and C' are related by one of the following relations:

either $C \stackrel{mpr}{\Longrightarrow}; \stackrel{tar}{\Longrightarrow} C'$, or $C \stackrel{mpr}{\Longrightarrow}; \stackrel{\delta}{\Longrightarrow} C'$, or $C \stackrel{mpr}{\Longrightarrow}; \stackrel{tar}{\Longrightarrow}; \stackrel{\delta}{\Longrightarrow} C'$.

The three alternatives in defining $C \Rightarrow C'$ are given by the existence of messages and dissolving symbols along the system evolution. Starting from a configuration without messages and dissolving symbols, we apply the "mpr" rules and get an intermediate configuration which is mpr-irreducible; if we have messages, then we apply the "tar" rules and get an intermediate configuration which is tar-irreducible; if we have dissolving symbols, then we apply the dissolving rules and get a configuration which is δ-irreducible. If the final configuration has no messages or dissolving symbols, then we say that the transition relation $\Rightarrow$ is well-defined as an evolution between the initial and final configurations.

Proposition 4. The relation $\Rightarrow$ is well-defined over the entire set $\mathcal{C}(\Pi)$ of configurations.

Examples of inference trees, as well as the proofs of the results are presented in [2].

Operational semantics provides us with a formal way to find out which transitions are possible for the current configuration of a membrane system. Given an operational semantics, we can derive easily an interpreter for membrane systems, as well as the basis for the definition of certain equivalences and congruences between membrane systems. Moreover, given an operational semantics, we can reason about the rules defining the semantics. A notion of bisimulation can be defined (see [2]), and the bisimulation relation allows to compare the evolution behaviour of two membrane systems.

3 Arithmetical Operations in Membrane Systems

The problem of number encoding using multisets is interesting and complex. The first paper on the encodings and arithmetical operations in membrane systems is [5]. In [5] we present several combinatorial results and some encodings of numbers using multisets. Here we present some arithmetical operations over numbers encoded by a simple and natural encoding (each object of a membrane represents a unit, and we use n objects to represent the number n). We indicate the complexity of some arithmetical operations, and build the membrane systems which implement the arithmetic operations over the encoded numbers.

Addition
Time complexity: $O(1)$

$$\begin{aligned}\Pi &= (V,\mu,w_0,(R_0,\emptyset),0),\\ V &= \{a,b\},\\ \mu &= [_0]_0,\\ w_0 &= a^n b^m,\\ R_0 &= \{b \to a\}.\end{aligned}$$

Addition is trivial; we consider n objects a and m objects b. The rule $b \to a$ says that an object b is transformed in one object a. Such a rule is applied in parallel as many times as possible. Consequently, all objects b are erased. The remaining number of objects a represents the addition $n+m$.

Subtraction
Time complexity: $O(1)$

$$\begin{aligned}\Pi &= (V,\mu,w_0,(R_0,\emptyset),0),\\ V &= \{a,b\},\\ \mu &= [_0]_0,\\ w_0 &= a^n b^m,\\ R_0 &= \{ab \to \lambda\}.\end{aligned}$$

Subtraction is described in the following way: given n objects a and m objects b, a rule $ab \to \lambda$ says that one object a and one object b are deleted (this is represented by the empty symbol λ). Consequently, all the pairs ab are erased. The remaining number of objects represents the difference between n and m.

Multiplication without promoters
Time complexity: $O(n \cdot m)$

The object is a promoter for a rule if the rule can be applied only in the presence of object. Figure 1 presents a P system Π_1 without promoters for multiplication of n (objects a) by m (objects b), the result being the number of objects d in membrane 0. In this P system we use the priority relation between rules; for instance $bv \to dev$ has a higher priority than $av \to u$, meaning the second rule is applied only when the first one cannot be applied anymore. Initially only the rule $au \to v$ can be applied, generating an object v which activates the rule $bv \to dev$ m times, and then $av \to u$. Now $eu \to dbu$ is applied m times, followed by $au \to v$. The procedure is repeated until no object a is present within the membrane. We note that each time when one object a is consumed, then m objects d are generated.

$$\begin{aligned}\Pi_1 &= (V,\mu,w_0,(R_0,\rho_0),0),\\ V &= \{a,b,e,v,u\},\\ \mu &= [_0]_0,\\ w_0 &= a^n b^m u,\\ R_0 &= \{r_1 : au \to v, r_2 : bv \to dev, r_3 : av \to u, r_4 : eu \to dbu\},\\ \rho_0 &= \{r_2 > r_1, r_4 > r_3\}.\end{aligned}$$

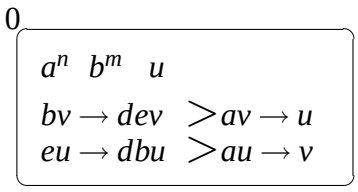

Figure 1: Multiplier without promoters

Multiplication with promoters
Time complexity: $O(n)$

Figure 2 presents a P system Π_2 with promoters for multiplication of n (objects a) by m (objects b), the result being the number of objects d in membrane 0. In this P system we use rules with priority and with promoters. The object a is a promoter in the rule $b \to bd|_a$, i.e., this rule can only be applied in the presence of object a. The available m objects b are used in order to apply m times the rule $b \to bd|_a$ in parallel; based on the priority relation and the availability of a objects (except one a as promoter), the rule $au \to u$ is applied in the same time. The priority relation is motivated because the promoter a is a resource for which the rules $b \to bd|_a$and $au \to u$ are competing. The procedure is repeated until no object a is present within the membrane. We note that each time when one object a is consumed, then m objects d are generated.

$$\begin{aligned}
\Pi_2 &= (V, \mu, w_0, (R_0, \rho_0), 0),\\
V &= \{a, b, u\},\\
\mu &= [_0]_0,\\
w_0 &= a^n b^m u,\\
R_0 &= \{r_1 : b \to bd|_a, r_2 : au \to u\},\\
\rho_0 &= \{r_1 > r_2\}.
\end{aligned}$$

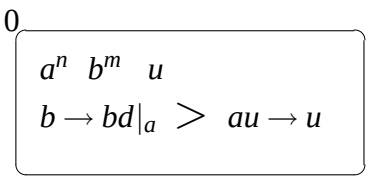

Figure 2: Multiplier with promoters

The membrane systems for multiplication differ from others presented in the literature [17] because they do not have exponential space complexity, and do not require active membranes. As a particular case, it would be quite easy to compute n^2 by just placing the same number n of objects a and b. Another interesting feature is that the computation may continue after reaching a certain result, and so the system acts as a P transducer [12]. Thus if initially there are n (objects a) and m (objects b), the system evolves and produces $n \cdot m$ objects d. Afterwards, the user can inject more objects a and the system continues the computation obtaining the same result as if the objects a are present from the beginning. For example, if the user wishes to compute $(n+k) \cdot m$, it is enough to inject k objects a at any point of the computation. Therefore this example emphasizes the asynchronous feature and a certain degree of reusability and robustness.

Division

We implement division as repeated subtraction. We compute the quotient and the remainder of n_2 (objects a in membrane 1) divided by n_1 (objects a in membrane 0) in the same P system evolution. The evolution starts in the outer membrane by applying the rule $a \rightarrow b(v, in_1)$. The (v, in_1) notation means that the object v is injected into the child membrane 1. Therefore the rule $a \rightarrow b(v, in_1)$ is applied $n_1$times converting the objects a into objects b, and object v is injected in the inner membrane 1. The evolution continues with a subtraction step in the inner membrane, with the rule $av \rightarrow e$ applied n_1 times whenever possible.

$$
\begin{aligned}
\Pi &= (V, \mu, w_0, w_1, (R_0, \rho_0), (R_1, \rho_1), 0), \\
V &= \{a, b, b', c, s, u, v\}, \\
\mu &= [_0[_1]_1]_0, \\
w_0 &= a^{n_1}s, \\
w_1 &= a^{n_2}s, \\
R_0 &= \{a \rightarrow b(v, in_1), b' \rightarrow a, r_1 : bu \rightarrow b'|_{\neg v}, r_2 : u \rightarrow \delta|_{\neg v}, r_3 : csu \rightarrow u|_v\}, \\
\rho_0 &= \{r_1 > r_2, r_2 > r_3\}, \\
R_1 &= \{r_1 : av \rightarrow e, r_2 : v \rightarrow (v, out), \\
&\quad\ r_3 : es \rightarrow s(u, out)(c, out), r_4 : e \rightarrow (u, out)\}, \\
\rho_1 &= \{r_1 > r_2, r_2 > r_3, r_3 > r_4\}.
\end{aligned}
$$

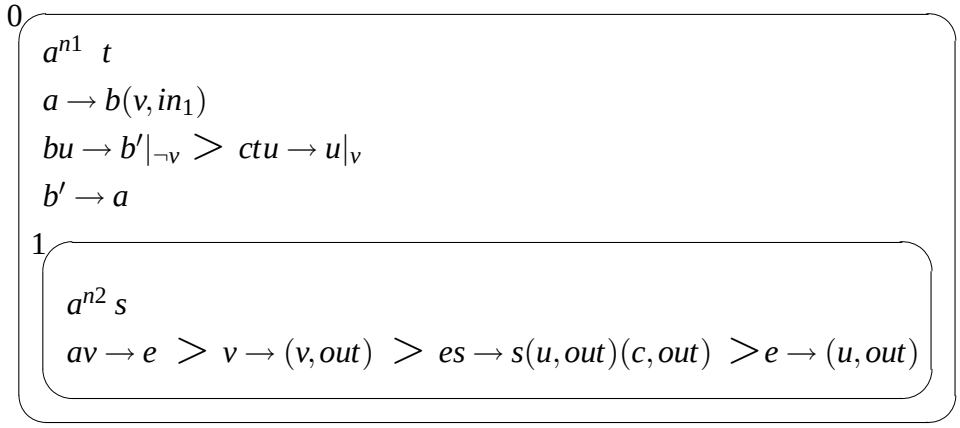

Figure 3: P system for division

Two cases are distinguished in the inner membrane:

- If there are more objects a than objects v, only the rules $es \rightarrow s(u, out)(c, out)$ and $e \rightarrow (u, out)$ are applicable. Rule $es \rightarrow s(u, out)(c, out)$ sends out to membrane 0 a single c (restricted by the existence of a single s into this membrane) for each subtraction step. The number of objects c represents the quotient. On the other hand, both rules send out n_1 objects u (equal to the number of objects e). The evolution continues in the outer membrane by applying $bu \rightarrow b'|_{\neg v}$ of $n_1$times, meaning the objects b are converted into objects b' by consuming the objects u only in the absence of v ($|_{\neg v}$ denotes an inhibitor having an effect opposite to that of a promoter). Then the rule $b' \rightarrow a$ produces the necessary objects a to repeat the entire procedure.

- When there are less objects a than objects v in the inner membrane we get a division remainder. After applying the rule $av \rightarrow e$, the remaining objects v activate the rule $v \rightarrow (v, out)$. Therefore all these objects v are sent out to the parent membrane 0, and the rules $es \rightarrow s(u, out)(c, out)$ and $e \rightarrow (u, out)$ are applied. Due to the fact that we have objects v in membrane 0, the rule $bu \rightarrow b'|_{\neg v}$ cannot be applied. Since n_2 is not divisible by n_1, the number of the left objects u in membrane 0 represents the remainder of the division. A final cleanup is required in this case, because an object c is sent out even if we have not a "complete" subtraction step; the rule $ctu \rightarrow u|_v$ removes that extra c from membrane 0 in the presence of v. This rule is applied only once because we have a unique t in this membrane.

The natural encoding is easy to understand and work with. However it has the disadvantage that the membranes can contain a very large number of objects when working with very large numbers. We introduce and study the most compact encoding using two object types (binary case) in [5], where we present other P systems implementing the arithmetical operations on numbers encoded using the binary cases of the most compact encoding. We use a web-based simulator available at `http://psystems.ieat.ro` to implement the arithmetical operations, and test each P system.

4 Software Implementations

Several programming paradigms and programming languages have been selected for implementing membrane systems simulators: Lisp, Haskell, MzScheme (as functional programming languages) Prolog, CLIPS (as declarative languages), C, C++, Java (as imperative and object-oriented languages). The user interface can be designed separately from the engine performing the evolution, and it is possible to use different programming languages able to communicate with each other. Each programming paradigm, each programming language has advantages and disadvantages.

Transition membrane systems and deterministic membrane systems with active membranes are simulated in Prolog [14]; they are used to solve NP-complete problems as SAT, VALIDITY, Subset Sum, Knapsack, and partition problems. Sevilla carpets describing the complexity of a membrane system computation [11] are used as a graphical representation for a partition problem in [20].

Membrane systems with active membranes, input membrane and external output are simulated in CLIPS and used to solve NP-complete problems in [18]. The simulator presented in [18] allows to observe the evolution of the systems with active membranes based on production system techniques. The set of rules and the configurations in each step of the evolution are expressed as facts in a knowledge base.

Rewriting membrane systems and membrane systems with symport/antiport rules are described as executable specifications in MAUDE in [1]. The advantage of this approach is that it uses the existing tools of Maude, and it is used to verify the temporal properties of the membrane systems expressed in linear temporal logic.

A more complex simulator (written in Visual C++) for membrane systems with active membranes and catalytic membrane systems is presented in [10]. It provides a graphical simulator, interactive definition, visualization of a defined membrane system, a scalable graphical representation of the computation, and step-by-step observations of the membrane system behaviour. The simulation of these membrane systems has to deal with the potential growth of the membrane structure and adapt dynamically the topology of the configurations depending if some membranes are added or deleted. Polynomial-time solutions to **NP**-complete problems via membrane systems can be reached trading time by space. This is done by producing (via membrane division) an exponential amount of membranes that can work in parallel.

In [10] it is presented a software implementation which provides a graphical simulation for two variants of membrane systems: for the initial version of catalytic hierarchical cell systems, and for membrane

systems with active membranes. Its main functions are given by an interactive definition of a membrane system, a visualization of a defined membrane system, a graphical representation of the computation and final result, and saving and (re)loading a defined membrane system. The application is implemented in Microsoft Visual C++ using MFC classes. For a scalable graphical representation, the Microsoft DirectX technology is used. One of the main features of this technology is that the size of each component of the graphical representation is adjusted according to the number of membranes of the system. The system is presented to the user with a graphical interface where the main screen is divided into two windows: The left window gives a tree representation of the membrane system including objects and membranes. The right window provides a graphical representation of the membrane system given by Venn-like diagrams. A menu allows the specification of a membrane system for adding new objects, membranes, rules and priorities. By using the functions *Start*, *Next* and *Stop*, the users can observe the system evolution step-by-step.

By simulating parallelism and nondeterminism on a sequential machine one can lose the power and attractiveness of membrane system computing. Parallel and cluster implementation for transition membrane systems in C++ and MPI are reported in [8] and [9]. The rules are implemented as threads. At the initialization phase, one thread is created for each rule. Rule applications are performed in terms of rounds. To synchronize each thread (rule) within the system, two barriers implemented as mutexes are associated with the thread. At the beginning of each round, the barrier that the rule thread is waiting on is released by the primary controlling thread. After the rule application is done, the thread waits for the second barrier, and the primary thread locks the first barrier. Since each rule is modelled as a separate thread, it should have the ability to decide its own applicability in a particular round. Generally speaking, a rule can run when no other rule with higher priority is running, and the resources required are available. When more than one rule can be applied in the same conditions, the simulator picks randomly one among the candidates. With respect to the synchronization and communication, for every membrane, the main communication is done by sending and receiving messages to and from its father and children at the end of every round. With respect to the termination, when the system is no longer active, there is no rule in any membrane that is applicable. When this happens, the designated output membrane prints out the result and the whole system halts. In order to detect if the membrane system halts, each membrane must inform the other membranes about its inactivity. It can do so by sending messages to others, and by using a termination detection algorithm [4].

The implementation was designed for a cluster of computers. It is written in C++ and it makes use of *Message Passing Interface (MPI)* as its communication mechanism. MPI is a standard library developed for writing portable message passing applications, and it is implemented both on shared-memory and on distributed-memory parallel computers. The program was implemented and tested on a Linux cluster at the National University of Singapore; the cluster consisted of 64 dual processor nodes.

The above implementations represent the first generation of membrane systems simulators. The recent developments are related to biological applications, and to a new generation of Web-based simulators. WebPS is an open-source web-enabled simulator for membrane systems [6]. The simulator is based on CLIPS, and it is already available as a Web application. As any Web application, WebPS does not require an installation. It can be used from any machine anywhere in the world, without any previous preparation. A simple and easy to use interface allows the user to supply an XML input both as text and as a file. A friendly way of describing membrane systems is given by an interactive JavaScript-based membrane system designer. The interface provides a high degree of (re)usability during the development and simulation of the membrane systems. The initial screen offers an example, and the user may find useful documentation about the XML schema, the rules, and the query language. The query language helps the user to select the output of the simulation. The simulator is free software, and it offered at `http://psystems.ieat.ro` under the *GNU General Public License*. This allows anyone to contribute with enhancements and error corrections to the code, and possibly develop new interfaces for the C and CLIPS level APIs. These interfaces can be local (graphical or command-line), or yet other

Web-based ones.

In the same paper [6], the authors present an accelerator for parallelization of the existing sequential simulators. This accelerator is used to parallelize an existing CLIPS simulator [18]. The speedup and the efficiency of the resulting parallel implementation are surprisingly close to the ideal ones.

5 Conclusion and Related Work

Structural operational semantics is an approach originally introduced by Plotkin [19] in which the operational semantics of a programming language or a computational model is specified in a logical way, independent of a machine architecture or implementation details, by means of rules that provide an inductive definition based on the elementary structures of the language or model. Structural operational semantics is intuitive and flexible, and it becomes more attractive during the years by the developments presented by Kahn [15] and Milner [16]. Configurations are states of transition systems, and computations consists of sequences of transitions between configurations, and terminating (if it terminates) in a final configuration. We present a structural operational semantics of the membrane systems; the inference rules provide a big-step operational semantics due to the parallel nature of the model. A structural operational semantics of the systems emphasizes also the deductive nature of the membrane computing by describing the transition steps by using a set of inference rules. Considering a set $\mathcal{R}$ of inference rules, we can describe the computation of a membrane system as a deduction tree. In [3] we translate the big-step operational semantics of membrane systems into rewriting logic. By using the rewriting engine Maude [13], we obtain an interpreter for membrane systems, and verify various properties of these systems.

Looking at the membrane systems from the point of view of programming theory, we define an appropriate data representation for P systems, and make the first steps to define an arithmetic unit for these abstract machine inspired by cells. The natural encoding over multisets is very close to biology, and can help to understand some biological mechanisms, improving also some computational models inspired by biology.

We have designed and implemented sequential and parallel software simulators; we present some of them, and compare with other software simulators of the P systems. A web-based implementation is presented in [6].

Acknowledgements

The contributions of this paper were obtained together with my colleagues. Many thanks to Oana Andrei and Dorel Lucanu for the joint work on the operational semantics of the membrane systems. Many thanks to Cosmin Bonchiş and Cornel Izbaşa for their contributions to the arithmetical operations over multisets in the framework of membrane systems, and to the software implementation WebPS.

References

[1] O. Andrei, G. Ciobanu, D. Lucanu. Executable Specifications of the P Systems. In *Membrane Computing WMC5*, LNCS vol.3365, Springer, 127-146, 2005.

[2] O. Andrei, G. Ciobanu, D. Lucanu. Structural Operational Semantics of P Systems. *Proceedings WMC6*, LNCS vol.3850, Springer, 32-49, 2006.

[3] O. Andrei, G. Ciobanu, D. Lucanu. Operational Semantics and Rewriting Logic in Membrane Computing. *Proceedings SOS Workshop* 2005, to appear in *ENTCS*.

[4] H. Attiya, J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 2000.

[5] C. Bonchiş, G. Ciobanu, C. Izbaşa. Encodings and Arithmetic Operations in Membrane Computing. In Jin-Yi Cai, S. Barry Cooper, Angsheng Li (Eds.): *Theory and Applications of Models of Computation*, LNCS 3959, Springer, 618–627, 2006.

[6] C. Bonchiş, G.Ciobanu, C. Izbaşa, D. Petcu. A Web-based P systems simulator and its parallelization. In C.Calude et al. (Eds.): *Unconventional Computing*, LNCS vol.3699, Springer, 58-69, 2005.

[7] G. Ciobanu. Distributed Algorithms over Communicating Membrane Systems. *Biosystems* vol.70, Elsevier, 123-133, 2003.

[8] G. Ciobanu, R. Desai, A. Kumar. Membrane Systems and Distributed Computing. In *Proceedings WMC3*, LNCS vol.2597, Springer, 187-202, 2003.

[9] G. Ciobanu, W. Guo. P Systems Running on a Cluster of Computers. In *Procedings 4th WMC*, Taragona, LNCS vol.2933, Springer, 123-139, 2004.

[10] G. Ciobanu, D. Paraschiv. P System Software Simulator. *Fundamenta Informaticae* 49, 61-66, 2002.

[11] G. Ciobanu, Gh. Păun, Gh.Ştefănescu. Sevilla Carpets Associated with P Systems. *Report 26/03 Rovira i Virgili University*, Tarragona, 135-140, 2003.

[12] G. Ciobanu, Gh. Păun, Gh. Ştefănescu. P Transducers. *New Generation Computing* **24**, 1–28, 2006.

[13] 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, 187-243, 2002.

[14] A. Cordon-Franco, M.A. Gutierrez-Naranjo, M.J. Perez-Jimenez, A. Riscos-Nunez, F. Sancho-Caparrini. Implementing in Prolog an Effective Cellular Solution for the Knapsack Problem. In *Proceedings 4th WMC*, Taragona, LNCS vol.2933, Springer, 140-152, 2004.

[15] G. Kahn. *Natural semantics*, Technical Report 601, INRIA Sophia Antipolis, 1987.

[16] R. Milner. Operational and algebraic semantics of concurrent processes. In J. van Leeuwen (Ed.), *Handbook of Theoretical Computer Science* vol.B, 1201-1242, Elsevier Science, 1990.

[17] Gh. Păun. *Membrane Computing. An Introduction*. Springer, 2002.

[18] M.J. Perez-Jimenez, F.J. Romero-Campero. A CLIPS Simulator for Recognizer P Systems with Active Membranes. In *Proceedings 2nd Brainstorming Week on Membrane Computing*, University of Sevilla Tech. Rep 01/2004, 387-413, 2004.

[19] G. Plotkin. Structural operational semantics. *Journal of Logic and Algebraic Programming* vol.60, 17-139, 2004.

[20] A. Riscos-Núñez, *Cellular Programming: Efficient Resolution of Numerical NP-Complete Problems*. PhD Thesis, University of Seville, 2004.

Gabriel Ciobanu
Romanian Academy
Institute of Computer Science
Address: Blvd. Carol I nr.8, Iaşi
E-mail: gabriel@iit.tuiasi.ro