Available Membrane Computing Software

Miguel Angel Gutiérrez-Naranjo, Mario J. Pérez-Jiménez,
Agustín Riscos-Núñez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
{magutier,marper,ariscosn}@us.es

Summary. The simulation of a P system with current computers is a quite com-plex task. P systems are intrinsically nondeterministic computational devices and therefore their computation trees are difficult to store and handle with computers with one processor (or a bounded number of processors). Nevertheless, there exists a first generation of simulators which can be successfully used for pedagogical pur-poses and as assistant tools for researchers. This chapter summarizes some of these simulators, presenting the state of the art of the available software for simulating (different variants of) cell-like membrane systems.

1 Introduction

In the few years since membrane computing was initiated [39] as a new branch of natural computing, a large number of variants have been considered, concerning both the syntax and the semantics of the model.

In many of these variants, P systems are seen as devices of a *generative* nature, that is, from a given initial configuration several distinct computa-tions may be developed (in a nondeterministic manner) and produce different outputs.

There are other approaches where P systems perform *computing tasks*. For example, if a certain number, n, is encoded somehow in the initial configuration and we consider the cardinality of the output multiset as the result of a successful computation, then we can interpret that to mean that the sys-tem *computes* a partial function from natural numbers onto sets of natural numbers.

Finally, membrane systems can also be used to deal with decision problems. In this case, special objects *yes* and *no* are included in the working alphabet, and thus the system is able to produce a boolean output (accepting or rejecting the input) in a *confluent* manner.

In all these approaches, we get the *output* of the computation from a final configuration, looking at the contents of the output membrane or, in the case of the *external output*, considering the objects that have been sent out of the system during the computation.

Unfortunately, for a machine-oriented model of computation as a P system is, it is usually a complex task to predict or to guess how a P system will behave when we are designing a cellular solution to a problem. Moreover, as there do not exist, up to now, implementations in laboratories (neither *in vitro* or *in vivo* nor in any electronical medium), it seems natural to look for software tools that can be used as assistants that are able to simulate computations of P systems.

This is the initial motivation for programming simulators. It is clear that such software tools are very useful when trying to understand how a cellular system works (both for pedagogical purposes and as an assistant tool for researchers). Another important point is that the formal verifications of the cellular solutions designed in this framework are specially hard, and having a simulator at hand allows us to quickly and easily get information about the evolution of P systems that can be used as starting point for a formal verification, maybe suggesting invariants that can be useful for the proofs. Finally, several of the existing P systems simulators were essentially used in the bio-applications of membrane computing (examples can be seen also in the first few chapters of this book).

The chapter is organized as follows. In the next section, some general considerations about the processes of the design and development of simulators of P systems are given. Section 3 is devoted to the simulators that work with transition P systems and run on sequential machines. Section 4 deals with parallel and distributed simulators (also simulating transition P systems) and Section 5 presents simulators for P systems with active membranes, including a session of one of them. The chapter ends with a section devoted to other software and some conclusions.

2 Preliminaries

The simulation of P systems with current computers is quite a complex task, but there have been several attempts in this direction in the last few years. We shall try to summarize some of them, presenting the state of the art of the available software for simulating (different variants of) cell-like membrane systems.

Generically speaking, the design and development processes for a P system simulator can be structured as follows:

2.1 Formal Definition of the Model

First of all, one has to choose which variant of membrane systems is going to be simulated, stating precisely the syntax and semantics of the model to avoid ambiguous interpretations. From a technical point of view, these models can be classified into two categories: the models of P systems where the number of membranes is bounded by the number of membranes in the initial configuration (i.e., this number does not change during the computation or decrease with the dissolution of membranes) and the models where the number of membranes can increase during the computation, via membrane creation or division.

The basic variant introduced in [39] is known as *transition P systems*. The rules in this model are of the form $u \to v$, where u is a string over the alphabet V and $v = v'$ or $v = v'\delta$, where v' is a string over

$$(V \times \{here, out\} \cup \{V \times \{in_j \,|\, 1 \leq j \leq n\})$$

and δ is a special symbol not in V. Besides, priority relations are considered among rules. These rules are applied in a maximally parallel way, that is, all objects which can evolve in one step must evolve (keeping in mind the priority restrictions).

This basic variant can be modified in many ways, for example, by restricting the model to non-cooperative rules or not allowing priority relations among rules, considering strings instead of multisets, or even substituting the classical tree-like membrane structure with tissue-like arrangements.

A specially relevant variant, namely, *P systems with active membranes* [41], is obtained by including rules for membrane division. Let us recall that the rules in this model are of the form

(a) $[x \to y]_h^\alpha$, for $h \in H$, $\alpha \in \{+, -, 0\}$, $x \in V$, $y \in V^*$ (*Object evolution rule*). This is an internal rule, associated with a membrane labeled h and depending on the polarity α of that membrane, but not directly involving the membrane.
(b) $x[\;]_h^{\alpha_1} \to [y]_h^{\alpha_2}$, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $x, y \in V$ (*Send-in communication rule*). An object from the region immediately outside a membrane labeled h is introduced in this membrane, is possibly transformed into another object, and, simultaneously, the polarity of the membrane can be changed.
(c) $[x]_h^{\alpha_1} \to [\;]_h^{\alpha_2} y$, for $h \in H$, $\alpha_1, \alpha_2 \in \{+, -, 0\}$, $x, y \in V$ (*Send-out communication rule*). An object is sent out from a membrane labeled h to the region immediately outside, is possibly transformed into another object, and, simultaneously, the polarity of the membrane can be changed.
(d) $[x]_h^\alpha \to y$, for $h \in H$, $\alpha \in \{+, -, 0\}$, $x, y \in V$ (*Dissolution rule*). A membrane labeled h is dissolved in reaction with an object. The skin is never dissolved.
(e) $[x]_h^{\alpha_1} \to [y]_h^{\alpha_2}[z]_h^{\alpha_3}$, for $h \in H$, $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$, $x, y, z \in V$ (*Division rule*). An elementary membrane can be divided into two membranes with the same label but possibly different polarities. The skin cannot divide.

Note that this variant of P systems uses 2-division but no cooperation or priorities. The rules are applied according to the following principles (informal semantics of P systems with active membranes):

- The rules are used as usual in the framework of membrane computing; that is, in a maximally parallel way. In one step, each object in a membrane can be used *only by one* rule (nondeterministically chosen in case there are several possibilities), but any object which can evolve by a rule of any type should evolve.
- If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external membrane (more precisely, of the closest predecessor which is not dissolved).
- All elements which are not specified in any of the operations to apply remain unchanged.
- A division rule can be applied to a membrane and, at the same time, some evolution rules can be applied to some objects inside that membrane. In this case, we can suppose that "first" the evolution rules are used, changing the objects, and "after that" the division takes place, introducing copies of the results of the evolutions in the two newly generated membranes (keeping in mind that all these processes take place in the same step of computation).
- The rules associated with label i are used for all membranes with this label. At one step, different rules can be applied to different membranes with the same label, but one membrane can be the subject of *at most* one rule of types (b) to (e).

These two models (transition and with active membranes) are widely considered in the existing simulators.

2.2 The Choice of a Programming Language

Each programming language has its own advantages and disadvantages and, up to now, there is no objective criterion to decide which is the most suitable one for simulating the evolution of a membrane system. Indeed, a large number of different languages such as Haskell, Prolog, Java, C, LISP, Visual C++, CLIPS or Scheme have been chosen by authors in the literature. The language chosen has to be able to carry out the evolution of the P system and to interact with the user in a friendly way.

It is also possible to design the interface separately from the engine that performs the evolution, using two different programming languages that are able to communicate with each other. For example, declarative languages can be appropriated for programming the inference engine, because an evolution step of a P system is nearer to a production system based upon rules than to a list of instructions to be executed in a sequential way.

2.3 A Good Way to Represent the Knowledge

The choice of a suitable data structure is a key problem in all fields of Computer Science (in particular, when dealing with the simulation of P systems). This decision is of course related to the programming language used, as specific techniques related to it have to be applied. A good representation allows a quick transition between configurations, and therefore speeds up the simulation.

There are also some designs that use two different knowledge representations, one for communicating with the user, whose goal is to implement an easy way to input the data describing the P system and to present the output in a natural way, so that the simulator can provide a better understanding of the evolution of a P system even to users who are not familiar with the programming language, and another for handling configurations and rules in order to perform the evolution steps (an efficient internal representation of P systems).

If two different representations are used, it becomes very useful to have at our disposal a parser (able to analyze syntactically the input introduced by the user) and a compiler (that translates the analyzed input into the internal grammar). Note that in some cases the internal representation is the same grammar used to input the data, so no compiler is needed.

2.4 Design of an Inference Engine to Carry out the Computation

There exists a basic difficulty intrinsic to the simulation of a P system in a conventional computer: the main power of P systems, concerning the execution of computations, is their massive parallelism. Furthermore, there are two levels of parallelism: all objects inside a membrane can be transformed simultaneously, and this process occurs in all membranes at the same time. Therefore, in one time unit (cellular step), many atomic transformations can be carried out. However, sequential conventional computers have only one processor. This means that, regardless of the programming language and the design chosen for the simulator, only one atomic transformation can be performed in each time unit (processor step).

The second feature which makes hard the design of a simulator is the intrinsic nondeterminism of P systems. If there is a large number of branches in the computation tree, the storage of the information can exceed the capacity of the computer and therefore, from a practical point of view, the simulation in this case is not feasible.

Keeping in mind these two difficulties, that is, since current computers are not able to deal with all the information related to the maximal parallelism and the nondeterminism of (relatively large) P systems, different authors have imposed several restrictions on their simulators. These constraints can be to bound the number of membranes or the cardinality of the multisets, to develop the computation tree until a prefixed depth, or to follow only one branch in the computation tree.

Usually, in the first generation of simulators, the codes are balanced between efficiency and explicitness in the following sense: the purpose of designing a simulator is to get information about the evolution of the system that is simulated and, therefore, we are interested in a software able to describe the intermediate steps and configurations. In many cases, authors have preferred to write the first versions of their simulators in code where clarity is enforced over efficiency, leaving the latter for further versions.

In spite of these limitations, the success of the first generation of simulators of P systems is beyond doubt. They are useful tools for teachers and researchers. On the one hand, one of the main utilities of this software is its use for a better understanding of membrane computing, so it is a pedagogical tool of first choice. On the other hand, it has proved to be a useful assistant tool for the design and verification of complex P systems which solve problems, relieving researchers of calculations by hand.

3 Simulators of Transition P Systems

The first simulators appeared in 2000, less than two years after Păun's foundational paper [39] was presented. All of them were focused on the basic model of transition P systems, and they pointed out one feature that has been followed by newer simulators: the balance between understandability and efficiency.

3.1 Maliţa's Simulator (2000)

In the *Workshop on Multiset Processing* which was held in Curtea de Argeş, Romania, in 2000, Mihaela Maliţa presented one of the first simulators for membrane systems [30]. It is a program written in LPA-Prolog for simulating transition P systems.

A configuration is represented as a list of labeled nested lists where objects are represented together with their multiplicities. There are also flags `x` or `y`, to distinguish between objects that can and cannot be processed.

The rules are represented by expressions explicitly mentioning four fields: the membrane (region) where the rule can be applied, the ordinal of the rule in its membrane, the initial multiset, and a multiset of products with target indicators ($here, in(j)$, or out) or, eventually, with the flag *dissolve*.

This simulator applies a restricted parallelism in the following sense: in each step, *for each* membrane, the simulator selects *only one* rule, and then this rule is applied as many times as possible.

In this simulator Maliţa pointed out one of the general ideas of the first generation of simulators: the transparency of the code in order to follow the features of membrane computing paradigm. She did not try to make programming shortcuts or tricks that might have given an optimal program. Her intention was to write a program so transparent that anyone who knows Prolog could understand how a P system works, and any person having some familiarity with membrane systems could read and understand the Prolog code of the simulator.

The simulator behaves as follows. It receives as input the configuration of a system together with a set of rules, and a parameter specifying the desired number of evolution steps. The output of the simulator shows the configurations of one branch of the computation tree until reaching the desired number of evolutions.

3.2 Suzuki and Tanaka's Simulator (2000)

In the same year, Yasuhiro Suzuki and Hiroshi Tanaka presented in [50] a program written in LISP for simulating transition P systems without membrane division and, therefore, with the number of membranes and complexity of membrane structure limited by the initial configuration, since membranes can only be dissolved.

They consider a class of P systems, which they call Artificial Cell Systems (ACSs), consisting of a membrane structure, multisets of symbols placed in its regions, and a set of rewriting rules acting in all the regions.

As we pointed out above, different authors have imposed some constraints to the design of their simulators. The specific feature from this one is to bound the size of each multiset.

This simulator has been successfully used to simulate realistic situations, such as the Brusselator model (the model of a chemical oscillation related to the Belousov-Zabotinski reaction), and in modeling and analyzing ecological systems (see [51] for more examples of applications).

3.3 Natural Computing Group from Madrid (2002)

In several papers (see [5, 6, 9, 11, 10]) some members of the Natural Computing Group of the Technical University of Madrid [54] proposed frameworks and data structures suitable for P systems, but in an abstract rather than practical context.

In [7], based on previous theoretical formalizations, they present a simulator for transition P systems written in Haskell. They consider two *layers* in a P system: on the one hand, there is a static structure, composed of the membranes and objects of the system; on the other hand, there is a dynamic structure, which refers to the set of rules of the system.

They present several specific modules (*Abstract Data Types*) to transfer to the software the concepts of multiset, rule, region, membrane, etc.

The Haskell interpreter chosen has been Hugs98 for Microsoft Windows[1]. The source code can be downloaded from the P system Web page [55].

[1] The interpreter for several operating systems can be downloaded from `http://cvs.haskell.org/Hugs/pages/downloading.htm`.

The simulator behaves as follows. It receives as input a file encoding a system (configuration and rules in each region) and produces another file encoding a system (configuration and rules in each region), obtained by the application of *one* step of the computation (via a maximal multiset of rules *randomly* selected).

3.4 Balbontín et al. Simulator (2002)

Two years after the *Workshop on Multiset Processing*, also in Curtea de Argeş, D. Balbontín-Noval, M.J. Pérez-Jiménez, and F. Sancho-Caparrini presented during the *Workshop on Membrane Computing 2002* a simulator [12] for transition P systems written in MzScheme. A library of procedures was developed for working in two stages:

(1) first a parser analyzes the input and checks if it is syntactically correct, and if so, a compiler rewrites the input introduced by the user into an internal grammar;
(2) then, the simulation is carried out up to a prefixed level (number of evolution steps) in all branches of the computation tree.

The simulator behaves as follows. It receives as input the initial configuration of a system including the set of rules and a parameter specifying the desired number of evolution steps, and it outputs the computation tree of the P system, step by step, until reaching the desired number of evolutions.

The inference engine that actually implements the evolution steps follows the formalization from [47]. That is, first of all it checks which are the applicable rules, according to the priority relations; then it calculates the applicability vectors for each membrane (that is, the multisets of applicable rules satisfying the maximal parallelism condition); and finally it combines such vectors (one vector for each region) to get the applicability matrices for the system. The simulator uses this procedure to follow *all* the possible nondeterministic choices of the computation. The expansion of the computation tree is made in a progressive way, level by level (*breadth expansion*), to a prefixed depth.

3.5 Ardelean and Cavaliere's Simulator (2003)

A very interesting tool for modeling biological processes was presented in [4]. It can be thought as a transition P system simulator because the number of membranes does not change during the computation. More precisely, the software deals with a special variant of P systems: the allowed rules are both rewriting and symport/antiport rules. This variant of P systems has been proposed in [18] and its motivations are rooted in the idea to separate the evolutive mechanism of the cell from the communicative mechanism.

The authors try to bridge the mathematical model and biological reality, indicating how one can use the P system framework to model very important processes that happen in cells.

The simulator takes as input the rules of a system, its membrane structure (which can be any graph, not only a tree) and the multisets of objects associated with the regions. The software assigns to each rule two kinds of probabilities: *probability of being available* and *probability of winning a conflict*. The simulation takes place in the following way: at each step, the simulator decides which are the available rules in that step, and this decision is taken using the above mentioned probability. Then, the available rules are applied in the maximally parallel mode by using the *weak priority* approach. This can be seen as a competition of the rules for each single occurrence of the objects.

Several biological processes have been simulated illustrating the usefulness of this software (see [19] and Chapter 4 of the present book).

3.6 Nepomuceno's Simulator (2004)

In [37], we can find a software application, *SimCM*, written in Java. This tool is a friendly application which allows us to follow the evolution of a transition P system in a visual way. Essentially, we handle transition P systems by means of three basic operations: **Create** an initial membrane system (the simulator includes a debug mode in order to avoid user errors); **Load** and **Save** previously defined membrane systems; and **Carry out** a simulation of the P system evolution. This simulation can be made in three different ways: showing the computation tree to a given maximal depth, level by level, or guided.
The main screen is divided into four basic panels:

- **Computation tree**: this panel shows the tree of configurations after the simulation is finished or during its development.
- **Current cell**: initially, this panel contains a sketch in tree form of the membrane system to be studied (the program represents the membrane structures of P systems as trees). Once the simulation is finished or when it is in development, this panel will represent the state of the membrane system according to the configuration chosen by the user in the computation tree panel. In order to select a configuration, it suffices to simply click on the chosen node in the computation tree panel.
- **Rules**: in this panel the rules associated with each membrane are listed.
- **Applicable rules**: this panel shows the applicability multiset associated with the configuration selected by the user in the computation tree.

The simulator can be downloaded from the Web page of the Research Group on Natural Computing at the University of Seville [57].

4 Parallel and Distributed Simulators

As we already said in Section 2, one of the main difficulties in the simulation of P systems in current computers is that the computational power of these devices lies in their intrinsic massive parallelism. Several authors have implemented the first versions of simulators based on parallel and distributed architectures, which is close to the membrane computing paradigm.

4.1 Ciobanu and Wenyuan's Simulator (2003)

G. Ciobanu and G. Wenyuan presented in [24] a parallel implementation of transition P systems[2]. The implementation was designed for a cluster of computers. It is written in C++ and 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 implementation is object-oriented and involves three components:

- class *Membrane*, which describes the attributes and behavior of a membrane,
- class *Rule*, which stores information about a particular rule, and
- *Main method*, which acts as central controller.

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[3] are associated with the thread. At the beginning of each round, the barrier that the rule thread is waiting for 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 modeled 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 randomly picks one among the candidates.

With respect to synchronization and communication, the main communication for each membrane is done by sending and receiving messages to and from its parent and children at the end of every round. With respect to termination, when the system is no longer active there is no rule applicable in any membrane. When this happens, the designated output membrane prints out the result and the whole system halts.

In order to detect if the P system halts, each membrane must inform the other membranes about its inactivity. It can do so by sending messages to other membranes (these membranes can be *normal* or *inactive*) and by using a termination detection algorithm (see [8]).

[2] A preliminary version of this paper can be found in [23].

[3] A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread, and non-signaled when it is owned.

4.2 Syropoulos et al. Simulator (2003)

Syropoulos, Mamatas, Allilomes, and Sotiriades presented in [52] a purely distributive simulation of P systems. It is implemented using Java's *Remote Methods Invocation* to connect a number of computers that interchange data. As the authors pointed out, the idea of designing a distributed simulator for a network of computers, instead of doing so for a cluster architecture, avoids the problem of limited hardware compatibility. The class of P systems that the simulator can accept is a subset of the $NOP_2(coo, tar)$ family of systems, which have the computational power of Turing machines. This variant restricts the number of membranes to two, allows cooperation, and the symbol tar indicates that the communication rules use target indicators of the type in_j.

Initially, a copy of the simulator is installed on a number of different computers. Randomly, we choose a computer and assign to it the role of the external compartment, while the others play the role of the internal compartments. Upon starting, a *Membrane* object is ready to participate in the network on each computer. Threads are an essential aspect of the implementation. In particular, each membrane class runs in its own thread, which, in turn, operates on a different machine. When the system starts, the computer that plays the role of the external compartment reads the specification of a P system from an external file and stores the data.

An artificial parameter is introduced in order to prevent the system from going into an infinite loop. When the simulator has successfully parsed the P system's specification, the main computer decides whether there are enough resources or not. If the available resources match the requirements set by the description of the P system, then the simulator starts the computation. Otherwise, it aborts the execution. In order to be able to make this decision, the simulator has been designed in such a way that all membrane objects send multicast UDP packets to a well known multicast address. Each packet contains the IP address of each sender, and multicast packets are received by all objects participating in the network. Thus, each computer knows which computers are *alive* at any time. In this way, the main computer has all the necessary information to decide whether there are sufficient resources to start the computation. A universal clock is owned by the object that has the role of the external compartment. This object signals each clock tick by the time the previous macrostep is completed (i.e., when, for a given macrostep, all remote objects have finished their computation).

The source code of the system, a `jar` file, which can be used to install the simulator, as well as the documentation of the simulator, can be downloaded from [53].

5 Simulators of P Systems with Active Membranes

The third group of software tools are devoted to simulating P systems with active membranes. Polynomial time solutions to **NP**-complete problems via P systems can be reached by trading time with space. This is done by producing (via membrane division) an exponential number of membranes that can work in parallel.

The simulation of these P systems has to deal with the potential growth of the membrane structure and adapt dynamically the topology of the configurations depending on whether some membranes are added or deleted. Due to the obvious limitations of computational resources, the P systems which can be simulated are of a small size.

5.1 Ciobanu and Paraschiv's Simulator (2002)

In [21] G. Ciobanu and D. Paraschiv presented a software application which provides a graphical simulation for two variants of P systems: for the initial version of catalytic hierarchical cell systems and for P systems with active membranes (see [40, 41]). Its main functions are:

- interactive definition of a membrane system,
- visualization of a defined membrane system,
- a graphical representation of the computation and final result, and
- save and (re)load of a defined membrane system.

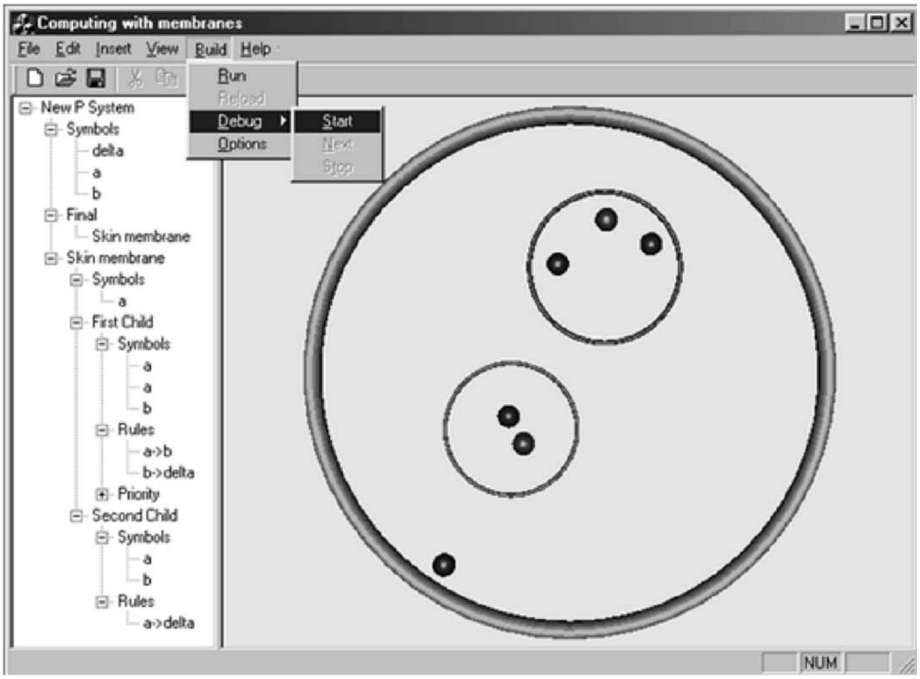

Fig. 1. Main screen of Ciobanu and Paraschiv's simulator.

The application was implemented in Microsoft Visual C++ using MFC classes. For a scalable graphical representation, Microsoft DirectX technology was 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 means of the functions *Start*, *Next*, and *Stop*, the user can observe the system evolution step by step.

The following two simulators have been developed as assistant tools for the design and formal verification of cellular solutions to **NP**-complete problems via recognizer P systems [45, 49]. In this case, as we work with confluent P systems, it suffices to follow one branch of the computation tree.

5.2 Pérez and Romero's Simulator (2004)

In this case ([45]) the simulator, written in CLIPS, deals with P systems with active membranes. The design is based on representing P systems through the *production systems* programming paradigm. Generally speaking, a production system can be structured into three components:

- *Working Memory*: A set of "facts" consisting of positive literals defining what is known to be true about the world.
- *Rules*: An unordered set of user-defined "if–then" rules of the form:

  $$\textbf{if } P_1 \wedge \ldots \wedge P_m \textbf{ then } Action_1, \ldots, Action_n$$

  where the P_is are facts that determine the conditions when the rule is applicable. Each *Action* adds or deletes a fact from the Working Memory.
- *Inference Engine*: Procedure for inferring changes (additions and deletions) to Working Memory.

Configurations are represented as a set of unordered facts using the following *template*:

```
(deftemplate membrane (slot number)
   (slot father)     (multislot children)
   (slot evolved)    (slot label)
   (slot polarity-0) (slot polarity-1)
   (slot multiset-0) (slot multiset-1))
```

The slots *number*, *father* and *children* are used to represent the membrane structure. The slots *label*, *polarity-0*, *polarity-1*, *multiset-0*, and *multiset-1* represent respectively the label, polarity of the membrane (current and next) and multiset (current and next) associated with each membrane.

The simulator transforms the rules of the P system into CLIPS rules as follows:

(a) $[x \to y]_h^\alpha$

```
(defrule evolution
   ?membrane <- (membrane (label h) (polarity-0 α )
                             (multiset-0 $?b0 , x , $?f0)
                             (multiset-1 $?b1 , x , $?f1))
  =>
   (modify ?membrane (multiset-0 $?b0 $?f0)
                       (multiset-1 $?b1 , y , $?f1)))
```

(b) $x[\ ]_h^{\alpha_1} \to [y]_h^{\alpha_2}$

```
(defrule send-in
   ?child <- (membrane (father ?f) (evolved 0)
                        (label h) (polarity-0 α1 )
                        (multiset-1 $?content))
   ?father <- (membrane (number ?f) (multiset-0 $?b0 , x , $?f0)
                         (multiset-1 $?b1 , x , $?f1))
  =>
   (modify ?child (evolved 1) (polarity-1 α2)
                   (multiset-1 $?content , y ,))
   (modify ?father (multiset-0 $?b0 $?f0) (multiset-1 $?b1 $?f1)))
```

(c) $[x]_h^{\alpha_1} \to [\ ]_h^{\alpha_2} y$

```
(defrule send-out
   ?child <- (membrane (father ?f) (evolved 0)
                        (label h) (polarity-0 α1 )
                        (multiset-0 $?b0 , x , $?f0)
                        (multiset-1 $?b1 , x , $?f1))
   ?father <- (membrane (number ?f) (multiset-1 $?content))
  =>
   (modify ?child (evolved 1) (polarity-1 α2)
                   (multiset-0 $?b0 $?f0) (multiset-1 $?b1 $?f1))
   (modify ?father (multiset-1 $?content , y ,)))
```

(d) $[x]_h^\alpha \to y$

```
(defrule dissolve
   ?child <- (membrane (number ?n) (evolved 0)
                        (father ?f) (children $?ch)
                        (label h) (polarity-0 α1 )
                        (multiset-0 $?b0 , x , $?f0)
```

```
                            (multiset-1 $?b1 , x , $?f1))
     ?father <- (membrane (number ?f) (children $?ch0 ?n $?ch1)
                            (multiset-1 $?content))
    =>
     (retract ?child)
     (assert (restructure (father ?f) (children $?ch)))
     (modify ?father (children $?ch0 $?ch $?ch1)
                            (multiset-1 $?content $?b1 , y , $?f1 )))
```

(e) $[x]_h^{\alpha_1} \rightarrow [y]_h^{\alpha_2}[z]_h^{\alpha_3}$

```
  (defrule division
     ?child <- (membrane (number ?n) (evolved 0)
                            (father ?f) (label h) (polarity-0 α₁ )
                            (multiset-0 $?b0 , x , $?f0)
                            (multiset-1 $?b1 , x , $?f1))
     ?father <- (membrane (number ?f) (children $?ch0 ?n $?ch1)
    =>
     (retract ?child)
     (assert (membrane (number ?*number*) (evolved 1) (father ?f)
                          (label h) (polarity-1 α₂)
                          (multiset-0 $?b0 $?f0)
                          (multiset-1 $?b1 , y , $?f1)))
               (membrane (number (+ ?*number* 1)) (evolved 1)
                          (father ?f) (label h) (polarity-1 α₂)
                          (multiset-0 $?b0 $?f0)
                          (multiset-1 $?b1 , z , $?f1)))
     (modify ?father (children $?ch0 ?*number* (+ ?*number* 1 ) $?ch1))
     (bind ?*number* (+ ?*number* 2)))
```

In order to carry out one step of the computation, the simulator performs first an initialization step where the rules are translated into CLIPS rules and the application of the rules is then simulated.

The simulator behaves as follows. It receives as input the initial configuration of a system and a set of rules, and it simulates only one branch of the computation tree, and several options are provided to choose the degree of verbosity of the output: show all the configurations of the evolution, or show only a concrete step, or run and show only the final answer (external output) of the system. Besides, the user can also decide if the rules applied are displayed for each step or not.

This simulator has been proven useful for designing and debugging families of P systems solving strongly **NP**-complete problems like BINPACKING and the Common Algorithmic Decision Problem (CADP). Currently, variants of this simulator which provide symport-antiport rules, catalysts, and a Java interface are being developed.

5.3 Cordón-Franco et al. Simulator (2004)

In [27] and [26], a new simulator written in Prolog was presented. It is pretty different from Maliţa's simulator ([30]) in implementation, and it works with P systems with active membranes [41] instead of with transition P systems.

This simulator has been successfully used as assistant in the design of P systems with active membranes to solve **NP**-complete problems, for instance, SAT, VALIDITY, Subset Sum, Knapsack, and Partition problems (see [25, 26, 27, 28, 46, 49]).

Similarly to other programs, the simulator stores and handles the information related to the P system and tries to show the process to the user in a friendly way. One of the main features of this simulator is that both tasks (computation and relation with the user) are made in the same language. For that, one exploits the ability of Prolog to define ad hoc symbols in order to imitate natural language.

In order to give a formal representation in Prolog of the basic structures of P systems with active membranes using 2-division, the following representation is considered. A given membrane structure is expressed by means of a labeled tree, where:

1. $< >$ is the *position* to denote the root of the tree and it is associated with the skin;
2. if $< i_1, \ldots, i_n >$ is the position of a membrane h, then $< i, i_1, \ldots, i_n >$ denotes the *position* of the ith membrane placed inside membrane h.

Let us remember that to give a configuration of a P system with active membranes consists of making explicit the membrane structure and the contents of all membranes.

In this model each configuration is represented as a set of one-literal clauses, each of them representing a membrane. Hence, in this representation each clause shows label, position, polarity, multiset of objects, and current step of the computation, as well as the P system the membrane belongs to. In this way, the set of clauses gives information about the contents of the membranes and the membrane structure (by means of the position of each one).

More precisely, to denote that in the step t of its evolution the P system P has a membrane at position $[pos]$ with label h, polarity α, and m as multiset, we write

$$P \;\texttt{::}\; h \;\texttt{ec}\; \alpha \;\texttt{at}\; [pos] \;\texttt{with}\; m \;\texttt{at\_time}\; t$$

Note that we use the user-friendly representation of a Prolog literal instead of the functional representation.

By means of some new function symbols, the rules are also represented as literals, in the following way:

(a) $[x \to y]_h^\alpha$

$P \;\texttt{rule}\; x \;\texttt{evolves\_to}\; [y] \;\texttt{in}\; h \;\texttt{ec}\; \alpha$

(b) $x[\,]_h^{\alpha_1} \rightarrow [y]_h^{\alpha_2}$

P `rule` x `out_of` h `ec` α_1 `sends_in` y `of` h `ec` α_2

(c) $[x]_h^{\alpha_1} \rightarrow [\,]_h^{\alpha_2} y$

P `rule` x `inside_of` h `ec` α_1 `sends_out` y `of` h `ec` α_2

(d) $[x]_h^{\alpha} \rightarrow y$

P `rule` x `inside_of` h `ec` α `dissolves_and_sends_out` y

(e) $[x]_h^{\alpha_1} \rightarrow [y]_h^{\alpha_2}[z]_h^{\alpha_3}$

P `rule` x `inside_of` h `ec` α_1 `divides_into` y `inside_of` h `ec` α_2
`and` z `inside_of` h `ec` α_3

The simulator behaves as follows. The input of the program is the initial configuration of the system (which is represented as a set of literals with predicate symbol `::`, all of them `at_time 0`) and a set of rules. The Prolog algorithm to carry out the evolution of a P system works in a natural way, as explained below. It is worth mentioning that only one branch of the computation tree is simulated, and therefore the result of the simulation is faithful only in the cases of *confluent* membrane systems (that is, systems that on the same input produce the same output).

- **Step 1: Initialization**. At the beginning of each computation step, all the membranes are set to *applicable* and their objects are split into two multisets: one **usable** multiset, containing all the objects of the initial membrane, and one **used** multiset which is empty.
- **Step 2: Transition**. If there exists an applicable membrane satisfying the condition of a rule, then the rule is applied in the following way:
  - **(a) step:** At this stage, only rules of type (a) are checked. The object which triggers the rule is removed from **usable** and the resulting multiset by the application of the rule is added to **used** to prevent the use of the same object by two different rules at the same step. After that, the membrane remains *applicable*, and new evolution rules can be applied. This stage ends when no more rules of type (a) can be applied.
  - **Non-(a) step**: At this stage, only one rule of the other types (not (a)) can be applied, and Prolog selects one from the existing possibilities (remember that this simulation works only with *confluent* P systems). The action depends on the kind of rule to apply:
    - **Send out rule**: The element which triggers the rule is removed from the **usable** multiset and the new one is added to the **used** multiset of the parent membrane. The membrane changes to *not applicable* mode. If the element is sent out of the skin, it is marked with the property **outside**.
    - **Send in rule**: This is the converse of the previous action. The element which triggers the rule is removed from **usable** in the parent membrane and the new one is added to the **used** multiset. The membrane changes to *not applicable* mode.

  - **Dissolution rule**: The element which triggers the rule is removed from **usable** and the new element obtained, together with the rest of the elements of the membrane, is added to the **used** multiset of the parent membrane. The dissolved membrane is removed, and the membranes inside it become children of the parent, and their positions are arranged to be correct.
  - **Division rule**: The element which triggers the rule is removed from **usable** and the division creates two new membranes in *not applicable* mode. One of them keeps the original position and the second one gets a position which has not been occupied by any membrane.
- **End**: When no more rules can be applied to membranes in *applicable* mode, a new configuration (with `at_time` incremented by 1) is stored. At this time no membrane has *applicable* or *not applicable* state. These modes have validity only during the evolution. At this point, the P system is ready for a new evolution step.

- **Step 3: End of computation**. If there are no rules to be applied, then the evolution finishes (the P system *halts*).

As we said before, the information provided by the simulator can be helpful in the processes of designing cellular solutions to some problems. Furthermore, this simulator has been also used in [29] as a tool to study the descriptive complexity of P systems. The complexity of a computation of a P system can be described for example by a table showing the number of times that the rules of the system are applied at each step. Such tables are known as *Sevilla carpets*, and were presented in [22].

The data needed to graphically represent the Sevilla carpet associated with a P system can be extracted from the output produced when simulating the system. To illustrate this, we present in Fig. 2 the carpet associated with a P system solving the Partition problem (the parameters of the instance are $N = 5$, $w_1 = 5, w_2 = 4, w_3 = 1, w_4 = 8$, and $w_5 = 6$). For further details we refer the readers to [29] or [49].

The user can choose from several possible outputs. We illustrate them by showing a session as an example.

In order to launch the simulator, we open a Prolog interpreter and load the main file `simulator.pl`. Suppose that we now want to solve an instance of the Subset Sum problem. The simulator includes a tool that is able, given an instance of the problem, to automatically generate the files containing the set of rules and the initial configuration of a recognizer P system with active membranes which solves the instance.

```
?- generate(subs).
% subs_file.pl compiled 0.00 sec, 6,876 bytes

Welcome to the program to generate the used files
for a P system to solve the SUBSET SUM problem
```

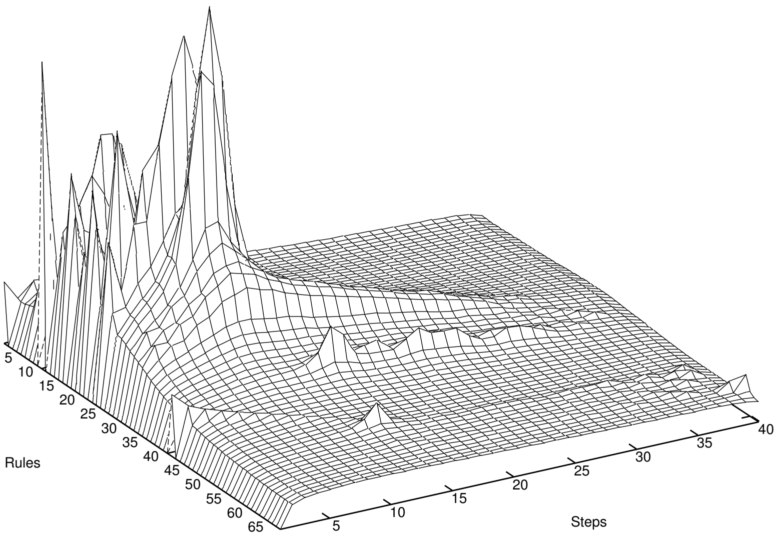

Fig. 2. Sevilla Carpet of a solution of the Partition problem through P systems with active membranes.

```
The Subset Sum problem is the following one: Given a finite
set A of N elements, a weight function w defined over it, and
a constant K determine whether or not there exists a subset of
A such that its weight is exactly k.
Please, introduce the name of the P system:
and one point (.) to end (e.g.: p1.)
Name: p1.

Next, introduce the parameters:
and one point (.) to end (e.g.: 5.)
Value of N = 5.
Value of K = 8.

The set of rules has been successfully generated
and stored in the file rules_subs_p1_5_8.pl
Do you want to load it now? (y./n.): y.

% rules_subs_p1_5_8.pl compiled 0.01 sec, 13,500 bytes

Ok, file loaded.
Next, we are going to build the initial configuration.
```

```
We need the specific INPUT for a concrete instance of
the SUBSET SUM problem
Introduce the list of weigths (e.g. [4,5,2,1].)
List: [5,2,7,9,2].

Please, write the name of the file to store the
initial configuration and one point (.) to finish
File : init_subs_p1_58_52792.
Ok, the initial configuration has been stored in the
file init_subs_p1_58_52792.pl

Do you want to load it now? (y./n.): y.

% init_subs_p1_58_52792.pl compiled 0.00 sec, 1,024 bytes

Ok, file loaded.
 Have a nice computation!
```

The current version of the program includes auxiliary files `subs_file.pl`, `knp_file.pl`, and `part_file.pl` to deal with the Subset Sum, Knapsack, and Partition problems, respectively. It is important to note that the generation process can be skipped if we perform further simulations of the same instances; it suffices then to load the corresponding files.

The user can select different types of outputs for the simulation. One can ask for a configuration in a concrete step, or to let the simulator run internally, getting only information about the number of cellular steps of the computation and the output of the P system.

```
?- go(p1).
The P system p1 stops at step 32 and returns NO
```

One can also ask the simulator to show the configurations step by step until a given point in the computation. Given a configuration of a P system `p1` at time `t`, the Prolog instruction that simulates one computation step is `evolve(p1,t)`.

```
?- evolve(p1,0).

p1 :: s ec 0 at [] with [z1-1] at_time 1
p1 :: e ec -1 at [1] with [a_-8, q-1, x1-5, x2-2, x3-7, x4-9, x5-2]
    at_time 1
p1 :: e ec 1 at [2] with [a_-8, e0-1, x1-5, x2-2, x3-7, x4-9, x5-2]
    at_time 1

Used rules in the step 0:
  * The rule 1 has been used only once
  * The rule 51 has been used only once
```

Note that the output displayed includes not only the next configuration but also information related to the rules used. Besides, the simulator informs us if any objects have been sent out to the environment.

```
?- evolve(p1,31).

p1 :: s ec 0 at [] with [# -508] at_time 32
p1 :: e ec -1 at [1] with [a-8, x1-5, x2-2, x3-7, x4-9, x5-2]
                                                          at_time 32
p1 :: e ec -1 at [2] with [a-3, x1-2, x2-7, x3-9, x4-2] at_time 32
p1 :: e ec -1 at [3] with [a-6, x1-7, x2-9, x3-2] at_time 32
p1 :: e ec -1 at [4] with [a-1, x1-7, x2-9, x3-2] at_time 32
.
.
.
p1 :: e ec 1 at [64] with [a0_-25, a_-8, e5-1] at_time 32

Used rules in the step 31:
  * The rule 83 has been used only once
The P-system has sent out d1 at step 29
The P-system has sent out no at step 31

?- evolve(p1,32).
No more evolution!
The P system p1 has already reached a halting configuration
at step 32
```

Currently, a graphical interface of this simulator is being developed using the Prolog/XPCE object-oriented library.

6 Other Software

We can also find in the literature other approaches that do not exactly fit in the previous sections.

For instance, although it is not exactly a simulator, we would like to note the work that Nicolau Jr., Solana, Fulga, and Nicolau published in *Fundamenta Informaticae* in 2002.

In [38], D.V. Nicolau Jr. et al. presented an ANSI C library developed to facilitate the implementation and simulation of P systems. Using the library proposed in this paper a user can specify an initial configuration (membrane structure and its contents) and perform actions on the objects or on the membranes. In fact, with this library membranes can be altered by dissolve, divide, and create actions. This library represents an intermediate step toward a practical implementation of P systems *in silico*.

The authors describe membrane structures as trees. A membrane is represented by a node in the tree, and the contents (symbols or strings) of that membrane are associated with the node by means of some auxiliary data structure such as an array or a list. The immediate "children" of this membrane are also included in the node contents. This is done by using a recursive *Abstract Data Type*. From a theoretical point of view, there is no limit on the number of membranes in a P system, and this represents a problem for simulating *in silico* P systems which allow division or creation of membranes. To avoid memory space violations, this software fixes an upper limit on the number of children in each membrane.

The information in the data structure also include the *name* of the membrane (its label), the number of children, and the name of its parent membrane.

Rules are not implemented explicitly as data structures. Instead, the user is supposed to write a *function* for each membrane reflecting the *program* of that membrane. In this way, the user is given complete flexibility over the way in which the rules are defined and applied in each membrane, including priority relations, and so on.

In September 2003, Alexandros Georgiou from University of Sheffield presented a simulator called SubLP-Studio. It is a software simulator for the Sub LP-Systems model, a variant of L systems and P systems. It optionally interfaces with CPFG, thus producing plant graphics using the turtle interpreter. It is available from the P systems Web page [55].

The *Group for Models of Natural Computing* [56] in Verona has developed a P systems simulator[4] based on the implementation of the metabolic algorithm introduced in [14]. The algorithm is inspired by the Law of Mass Action. This law states that the driving force of a chemical reaction is directly proportional to the active masses of all the reactants.

They propose regarding a rule $r : A_1A_2 \rightarrow B_1B_2$ as a *chemical reaction*; then the left objects A and B play the role of *reactants* while those of the right are *products*. Following this chemical interpretation, they propose regarding rules as descriptors of the changes in concentration of the reactants into products.

The simulator which implements these ideas is written in Java and the input is provided to the simulator as an XML file.

We would also like to note the implementation of catalytic P systems presented by Binder et al. in [15] and Alhazov's simulator for maximally parallel multiset-rewriting systems with promoters/inhibitors [1]. The latter was used as an engine of the communicative P systems simulator by Vladimir Rogozhin to check the theorems in [2, 32].

Although it is beyond the scope of the present chapter, we consider Petreska and Teuscher's implementation [48] interesting. Instead of developing software, they have presented a hardware-based parallel implementation that allows us to run a certain class of P systems in a highly efficient manner. The source code of the implementation and more information are available from [58].

[4] The simulator is described in [13].

It is also worth mentioning the fact that Holger Hoos from the University of British Columbia teaches a course on *Algorithms for Bioinformatics*[5] and one of the assignments for his students is to implement a P system simulator for a restricted version of transition P systems.

7 Conclusions

In this chapter we have briefly presented some programs from what we consider to be the first generation of P systems simulators. In a few years, more than a dozen software simulators have been presented. As we pointed out above, the common purpose of all of them is better understanding of the computational process of P systems for pedagogical purposes, as assistants for researchers, and for use (mainly) in biological applications. One of the most extended features is the balance between efficiency and explicitness of the code.

We are at the beginning of a new generation of simulators, whose properties have been already pointed out by some of the simulators mentioned above.

For example, it is necessary that the simulator have a friendly and intuitive graphical interface. This is not a trivial task, because problems such as division or dissolution of membranes need dynamical solutions in order to update the graphical representation of all membranes *simultaneously*.

Another important point to address is the way in which the P system is provided to the simulator. Future simulators will need parsers to check the information provided by the user and store it appropriately. Likewise, the use of tools is necessary to handle information of P systems when the number of rules, membranes, or objects in a configuration is large.

It is also desirable that the simulator be able to interact with the user by providing detailed information about the computation, for example, about the number of rules used in each step and intermediate configurations or objects sent to the environment (if any) in order to make statistical studies of the computations (see, e.g., [19], [29], [31] or [51]). Indeed, biologically inspired variants of membrane systems are not interested in looking for halting configurations, but in the evolution process itself.

Then, the simulators have to be tested when approaching new problems both with computational interest (such as solving new **NP**-complete problems) and related to applications in biology. The development of more complex simulators will also require the use of tools for their verification.

The next generation of simulators may be oriented to solve (at least partially) the problems of storage of information and massive parallelism by using parallel language programming or by using multiprocessor computers. In this framework, the emergent generation of simulators based on parallel or distributed architectures could lead to an *efficient* simulation of P systems *in silico.*

[5] http://www.cs.ubc.ca/labs/beta/Courses/CPSC545-03/

In some sense, the current P systems simulators represent a first step toward an implementation of such cellular models in electronic media. However, we can note an important limitation: the problem of finding an *efficient* implementation of P systems with active membranes (i.e., a software able to simulate computations with a polynomial number of cellular steps in polynomial processor time) is as hard as proving that **P**=**NP**.

Acknowledgement

The support for this research through project TIC2002-04220-C03-01 of the Ministerio de Ciencia y Tecnología of Spain, cofinanced by FEDER funds, is gratefully acknowledged.

References

1. A. Alhazov: Maximally Parallel Multiset-Rewriting Systems: Browsing the Configurations. *Proc. Third Brainstorming Week on Membrane Computing*, Sevilla, 2005, RGNC Report 01/2005, 1-10.
2. A. Alhazov, M. Margenstern, V. Rogozhin, Yu. Rogozhin, S. Verlan: Communicative P systems with Minimal Cooperation. In [35].
3. A. Alhazov, C. Martín-Vide, Gh. Păun, eds.: *Pre-Proceedings of the Workshop on Membrane Computing*, Tarragona, Spain, 2003, Report RGML 28/03.
4. I.I. Ardelean, M. Cavaliere: Modelling Biological Processes by Using a Probabilistic P System Software. *Natural Computing*, 2, 2 (2003), 173–197.
5. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. de Mingo: A Recursive Algorithm for Describing Evolution in Transition P Systems. In [33], 19–30.
6. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. de Mingo: Structures and Bio-Language to Simulate Transition P Systems on Digital Computers. In [17], 1–16.
7. F. Arroyo, C. Luengo, A.V. Baranda, L.F. de Mingo: A Software Simulation of Transition P Systems in Haskell. In [43], 19–32.
8. H. Attiya, J. Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 2000.
9. A.V. Baranda, J. Castellanos, F. Arroyo, R. Gonzalo: Data Structures for Implementing P Systems in Silico. In [16], 21–34.
10. A.V. Baranda, J. Castellanos, F. Arroyo, R. Gonzalo: Towards an Electronic Implementation of Membrane Computing: A Formal Description of Nondeterministic Evolution in Transition P Systems. In *Proceedings of DNA-Based Computers, Tampa, Florida, 2002* (N. Jonoska, N.C. Seeman, eds.), LNCS 2340, Springer, Berlin, 2002, 350–359.
11. A.V. Baranda, J. Castellanos, R. Gonzalo, F. Arroyo, L.F. de Mingo: Data Structures for Implementing Transition P Systems in Silico. *Romanian Journal of Information Science and Technology*, 4, 1-2, (2001), 21–32.
12. D. Balbontín-Noval, M.J. Pérez-Jiménez, F. Sancho-Caparrini: A MzScheme Implementation of Transition P Systems. In [43], 58–73.
13. L. Bianco: A P System Simulator: Introduction to Psim. Unpublished manuscript, 2004.

14. L. Bianco, F. Fontana, G. Franco, V. Manca: P Systems for Biological Dynamics. In this volume.
15. A. Binder, R. Freund, G. Lojka, M. Oswald: Implementation of Catalytic P Systems. *Proceedings of CIAA 2004, Ninth International Conference on Implementation and Application of Automata*, Kingston, Canada, 2004, 24–33.
16. C.S. Calude, M.J. Dinneen, Gh. Păun, eds.: *Pre-Proceedings of Workshop on Multiset Processing*, Curtea de Argeş, Romania, CDMTCS TR 140, Univ. of Auckland, 2000.
17. C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *Multiset Processing. Mathematical, Computer Science and Molecular Computing Points of View*, LNCS 2235, Springer, Berlin, 2001.
18. M. Cavaliere: Evolution-Communication P Systems. In [43], 134–145.
19. M. Cavaliere, I.I. Ardelean: Modelling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria by Using a P System Simulator. In this volume.
20. M. Cavaliere, C. Martín-Vide, Gh. Păun, eds.: *Proceedings of the Brainstorming Week on Membrane Computing*, Tarragona, Spain, 2003, Report RGML 26/03.
21. G. Ciobanu, D. Paraschiv: P System Software Simulator. *Fundamenta Informaticae*, 49, 1-3 (2002), 61–66.
22. G. Ciobanu, Gh. Păun, Gh. Ştefănescu: Sevilla Carpets Associated with P Systems. In [20], 135–140.
23. G. Ciobanu, G. Wenyuan: A Parallel Implementation of Transition P Systems. In [3], 169–184.
24. G. Ciobanu, G. Wenyuan: P Systems Running on a Cluster of Computers. In [34], 123–139.
25. A. Cordón-Franco, M.A. Gutiérrez-Naranjo. M.J. Pérez-Jiménez, A. Riscos-Núñez, F. Sancho-Caparrini: Implementing in Prolog an Effective Cellular Solution to the Knapsack Problem. In [34], 140–152.
26. A. Cordón-Franco, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F. Sancho-Caparrini: Cellular Solutions of Some Numerical NP-Complete Problems: A Prolog Implementation. In *Molecular Computational Models: Unconventional Approaches* (M. Gheorghe, ed.), Idea Group, Inc., 2005, 115–149.
27. A. Cordón-Franco, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F. Sancho-Caparrini: A Prolog Simulator for Deterministic P Systems with Active Membranes. *New Generation Computing*, 22, 4 (2004), 349–364.
28. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez: A Fast P System for Finding Balanced 2-Partition. *Soft Computing*, 9 (2005).
29. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez: On Descriptive Complexity of P Systems. In [36], 245–255.
30. M. Maliţa: Membrane Computing in Prolog. In [16], 159–175.
31. V. Manca: On the Dynamics of P Systems. In [36], 29–43.
32. M. Margenstern, V. Rogozhin, Yu. Rogozhin, S. Verlan: About P Systems with Minimal Symport/Antiport Rules and Four Membranes. In [36], 283–294.
33. C. Martín-Vide, Gh. Păun, eds.: *Pre-Proceedings of Workshop on Membrane Computing*, Curtea de Argeş, Romania, August 2001. Technical Report GRLMC 17/01, Rovira i Virgili University, Tarragona, Spain, 2001.
34. C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *Membrane Computing. International Workshop WMC2003, Tarragona, Spain, 2003. Revised Papers*. LNCS 2933, Springer, Berlin, 2004.

35. G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa, eds.: *Membrane Computing. International Workshop WMC5, Milano, Italy, 2004. Revised Papers.* LNCS 3365, Springer, Berlin, 2005.
36. G. Mauri, Gh. Păun, C. Zandron, eds.: *Pre-Proceedings of the Workshop on Membrane Computing WMC5*, Universitá di Milano-Bicocca, Italy, 2004.
37. I.A. Nepomuceno-Chamorro: A Java Simulator for Basic Transition P Systems. In [42], 309–315.
38. D.V. Nicolau Jr., G. Solana, F. Fulga, D.V. Nicolau: A C Library for Simulating P Systems. *Fundamenta Informaticae*, 49, 1-3 (2002), 241–248.
39. Gh. Păun: Computing with Membranes. Turku Centre for Computer Science, TUCS Technical Report, Nr.208, 1998.
40. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
41. Gh. Păun: P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal of Automata, Languages and Combinatorics*, 6, 1 (2001), 75–90.
42. Gh. Păun, A. Riscos-Núñez, A. Romero-Jiménez, F. Sancho-Caparrini, eds.: *Proceedings of the Second Brainstorming Week on Membrane Computing*, Sevilla, Spain, Report RGNC 01/04, 2004.
43. Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.: *Membrane Computing, International Workshop WMC-CdeA 2002, Curtea de Argeş, Romania. Revised Papers.* LNCS 2597, Springer, Berlin, 2003.
44. Gh. Păun: *Membrane Computing. An Introduction.* Springer, Berlin, 2002.
45. M.J. Pérez-Jiménez, F. Romero-Campero: A CLIPS Simulator for Recognizer P Systems with Active Membranes. In [42], 387–413.
46. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini: Solving VALIDITY Problem by Active Membranes with Input. In [20], 279–290.
47. M.J. Pérez-Jiménez, F. Sancho-Caparrini: A Formalization of Transition P Systems. *Fundamenta Informaticae*, 49, 1-3 (2002), 261–272.
48. B. Petreska, C. Teuscher: A Reconfigurable Hardware Membrane System. In [34], 269–285.
49. A. Riscos-Núñez: *Cellular Programming: Efficient Resolution of Numerical NP-Complete Problems.* Ph.D. Thesis, University of Seville, 2004.
50. Y. Suzuki, H. Tanaka: On a LISP Implementation of a Class of P Systems. *Romanian Journal of Information Science and Technology*, 3, 2 (2000), 173–186.
51. Y. Suzuki, Y. Fujiwara, H. Tanaka, J. Takabayashi: Artificial Life Applications of a Class of P Systems: Abstract Rewriting Systems on Multisets. In [17], 299–346.
52. A. Syropoulos, E.G. Mamatas, P.C. Allilomes, K.T. Sotiriades: A Distributed Simulation of Transition P Systems. In [34], 357–368.
53. `http://research.araneous.com`
54. `http://www.lpsi.eui.upm.es/nncg/`
55. `http://psystems.disco.unimib.it/`
56. `http://www.di.univr.it`
57. `http://www.gcn.us.es`
58. `http://www.teuscher.ch/psystems`