A Simulator for Confluent P Systems

Miguel A. 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
E-mail: {magutier,marper,ariscosn}@us.es

Summary. Software simulators for P system are nowadays the main tool to carry out experiments in the field of Membrane Computing. Although the simulation of a P system is a quite complex task, current simulators have been successfully used for pedagogical purposes and also as assistant tools for researchers. Up to now, simulators have always been designed to deal with a specific model of P systems. In this paper we present a new simulator which is not oriented to only one model of systems, but it allows the researcher to experiment with many existing models or even to create new ones.

1 Introduction

Since Gh. Păun initiated Membrane Computing [21] as a new branch of Natural Computing, a large number of research lines have arisen in the field, both concerning the syntax and the semantics of the model. For example, we can consider P systems from a *generative* point of view, where starting from a fixed initial configuration, the system may generate nondeterministically a set of different outputs (that can be strings over some alphabet or also natural numbers); we can also choose an *accepting* approach, where the system accepts or not an input that is introduced at the beginning of the computation (this input can be a number, a word from a certain language, or an instance of a decision problem); furthermore, we can interpret the evolution of the P system as a *computing* process where some function is computed (given an input n, the output of the system will be $f(n)$); or we can think of any other interpretation of the behavior of the P systems adequate to our purposes.

Irrespectively of the selected approach, it is usually a complex task to predict or to guess how a P system will behave. Moreover, as there do not exist, up to now, implementations in laboratories (neither *in vitro* nor *in vivo* nor in any electronic medium), it seems natural to look for software tools that can be used as assistants that are able to simulate computations of P systems.

In the literature, several simulators can be found, but all of them have been designed to make experiments within a specific model of P systems, that is, both the set of possible types of rules and the way such rules are applied are fixed. Thus, it becomes a difficult task to compare several solutions to a problem designed in different models, and these types of software cannot be used to investigate new possibilities beyond the usual models.

In this paper we propose a new tool which is born with the hope of becoming a tool for the creativity in Membrane Computing. It deals with several of the current models for P systems, also allowing to mix them in order to explore new possibilities. Indeed, its main feature is its modularity and the ability of embedding new P system models in future releases with a minimal modification of the code.

For example, in the current version, we have not included rules of endocytosis and exocytosis (mainly because, to the best of our knowledge, these have not been used yet in the solution of problems). Nevertheless, these rules can be included in future versions and coexist with the current ones. We will discuss more about the modularity of our simulator in Section 3.

The paper is organized as follows. In the next section, an overview of the current simulators is given. Section 3 presents the *simulator for confluent P systems (SCPS)*. Section 4 is devoted to discuss the improvements of this tool with respect to other previous simulators and the paper ends with some final comments and conclusions.

2 Other Software

The first simulators appeared in 2000, only two years after Păun's foundational paper [21] was presented. In a few years, more than ten software simulators have been presented (see [16]), each one of them designed to simulate a specific model (or a small number of variants). Their common purposes is the better understanding of the computational process of P systems, pedagogical purposes, as well as assistant for researchers.

Among the first simulators for *transition* P systems, we can find Maliţa's simulator [19], written in LPA-Prolog or Suzuki-Tanaka's one [27], written in Lisp. In [3], the Natural Computing Group of the Technical University of Madrid [29], presented a simulator for transition P systems written in Haskell, based on some previous theoretical formalizations (see [1, 2, 4, 5, 6]). Other simulators for transition P systems were Balbontín-Pérez-Sancho's simulator [7], written in MzScheme, and Nepomuceno's one [20].

Other remarkable software is Ciobanu-Wenyuan's simulator [10], which is a parallel implementation of *transition P systems*, and Syropoulos-Mamatas-Allilomes-Sotiriades's one, presented in [28].

We would like also to mention the web-enabled simulator recently presented in the Third Brainstorming Week on Membrane Computing by C. Izbasa, C. Bonchis, C. Garboni and G. Ciobanu [17].

Another group of software tools are devoted to simulate *P systems with active membranes* (see [22, 23]). The main interesting point of these is that polynomial-time cellular solutions to **NP**-complete problems 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.

Among the simulators which can handle P systems with active membranes, we can consider Ciobanu-Paraschiv's simulator [9] which provides a graphical simulation for two variants of P systems, the initial version of catalytic hierarchical cell systems and P systems with active membranes, or Pérez-Romero's simulator [24], written in CLIPS, which also deals with P systems with active membranes.

In [12] and [13], a new simulator written in Prolog was presented. It is pretty different from Maliţa's simulator ([19]) in the implementation and, besides, it works with P systems with active membranes. This simulator has been successfully used as assistant in the design of recognizer P systems with active membranes to solve **NP**-complete decision problems, for instance, SAT, VALIDITY, Subset Sum, Knapsack and Partition problems (see [11, 12, 13, 15, 25, 26]).

The simulator that we present takes some ideas from this one, but its algorithm and implementation are quite different (and more *efficient*).

3 Introducing the SCPS Simulator

As pointed out in [16], the design and development processes for a P system simulator can be structured in several stages. Next we give a short overview of the formal definition of the P systems that can be simulated by our software, we comment the way in which the information is expressed and handled in Prolog during the simulation, and we explain the functioning of the inference engine, including some final comments and examples.

3.1 Formal definition of the model

First of all, one has to choose which variant of membrane systems to simulate, stating precisely the syntax and semantics of the model to avoid ambiguous interpretations that could produce an incorrect functioning of the software.

From a technical point of view, P system models can be classified into two categories: the models of P systems where the number of membranes along the computations is bounded by the number of membranes in the initial configuration (i.e., it does not change along the computation or it decreases by dissolution of membranes) and the models where the number of membranes can increase along the computation, via membrane creation or division.

Up to now, the simulators only can deal with one or two variants of P systems. One of the main features of the SCPS simulator is the ability of simulate a wide range of models of P systems. In fact, it is able to develop simulations of new models of P systems (for example, the model obtained by allowing both division

and merging of membranes). This is possible due to its modularity in which the simulator can handle several kinds of rules in any possible combination.

It is also able to handle electrical charges (usually 2 or 3 polarizations are used, but the simulator can handle any number of polarizations, no polarizations, or even the non-explored case in which some membranes can be electrically charged and some other not). The simulator can also handle rules that change membrane labels, or a combination of both, changing labels and polarizations.

More precisely, the simulator is able to work with any combination of the following types of rules (where H is the set of labels and E is the set of electrical charges):

(a) $[x \to u]_h^\alpha$, for $h \in H$, $\alpha \in E$, $x \in V$, $u \in V^*$ (*Object evolution rule*). This is an internal rule, associated with a membrane labelled by h and depending on the polarity of that membrane (if it exists), but not directly involving the membrane.

(b) $x[\ ]_{h_1}^{\alpha_1} \to [y]_{h_2}^{\alpha_2}$, for $h_1, h_2 \in H$, $\alpha_1, \alpha_2 \in E$, $x, y \in V$ (*Send-in communication rule*). An object from the region immediately outside a membrane labelled by h is introduced in this membrane, possibly transformed into another object and, simultaneously, the polarity and the label of the membrane can be changed.

(c) $[x]_{h_1}^{\alpha_1} \to [\ ]_{h_2}^{\alpha_2} y$, for $h_1, h_2 \in H$, $\alpha_1, \alpha_2 \in E$, $x, y \in V$ (*Send-out communication rule*). An object is sent out from a membrane labelled by h to the region immediately outside, possibly transformed into another object and, simultaneously, the polarity and the label of the membrane can be changed.

(d) $[x]_h^\alpha \to y$, for $h \in H$, $\alpha \in E$, $x, y \in V$ (*Dissolution rule*). A membrane labelled by h is dissolved in reaction with an object. The skin is never dissolved.

(e) $[x]_{h_1}^{\alpha_1} \to [y]_{h_2}^{\alpha_2} [z]_{h_3}^{\alpha_3}$, for $h_1, h_2, h_3 \in H$, $\alpha_1, \alpha_2, \alpha_3 \in E$, $x, y, z \in V$ (*Division rule*). A membrane can be divided into two membranes with possibly different labels and polarizations, simultaneously transforming some objects. The skin cannot divide.

(f) $[x \to [u]_{h_1}^{\alpha_1}\ ]_{h_2}^{\alpha_2}$, for $h_1, h_2 \in H$, $\alpha_1, \alpha_2 \in E$, $x \in V$, $u \in V^*$ (*Creation rule*). A new elementary membrane can be created inside an existing one from an object. In nature, this process is know as *autopoiesis*, see [18].

(g) $[x]_{h_1}^{\alpha_1} [y]_{h_2}^{\alpha_2} \to [z]_{h_3}^{\alpha_3}$, for $h_1, h_2, h_3 \in H$, $\alpha_1, \alpha_2, \alpha_3 \in E$, $x, y, z \in V$ (*Merging rule*). Two elementary membranes with the same father can be merged into a new membrane and simultaneously transforming some objects.

For rules of type (b), (c) and (e), if $h_1 = h_2 = h_3$, we have the usual communication and division rules of the P system model with active membranes.

Note that, with the exception of the *object evolution rules*, all the remaining types or rules are coupled: *Send-in* with *Send out*, *Division* with *Merging* and *Dissolution* with *Creation*.

These rules do not use cooperation or priorities. Due to the modularity of the simulator, these features can be easily added in new versions according with the idea of making a general simulator for all kinds of confluent P systems.

The rules are applied according to the following principles:

- The rules are used as customary in the framework of membrane computing, that is, in a maximally parallel way. In one step, each object in a membrane can *only* be used for *one* rule (non-deterministically 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 (but keeping in mind that all these processes take place in the same step of computation).
- The rules associated with a label h 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 only be the subject of *at most* one rule of types (b), (c), (d), (e) and (g).

The modularity of the simulator also affects the semantics. In the present version, communication rules of types (b) and (c) (*send-in* and *send-out* rules, respectively) can only be applied sequentially, in the same way as in P systems with active membranes. This means that only one object can cross each membrane in each step of the computation.

We have chosen this option because the model of P systems with active membranes is the main variant for which we are interested in running simulations (as there exist in the literature many cellular solutions to hard problems in this model). Nevertheless, this can be easily modified by the user, or even several types of rules can coexist, by simply specifying a flag in the syntax of the system.

3.2 Implementing P systems in Prolog

Each programming language has its own advantages and disadvantages and, up to now, there is no objective criterion to decide which is the more suitable one for simulating the evolution of a membrane system. We decided to use Prolog[1] because of two main features: it is expressive enough to handle symbolic knowledge in a natural way, and it has the ability of evolving the different configurations following a set of rules.

On one hand, the tree-based data structure and the use of infix operators defined *ad hoc* by the programmer allow us to *simulate* the natural language, and the user can follow the evolution of the system without any knowledge of

[1] A good starting point for Prolog can be [8] or [30].

Prolog. On the other hand, Prolog programs are sets of facts and rules, and basic mechanisms of Prolog are pattern matching and automatic backtracking, so the design of the inference engine to perform the evolutions has a natural treatment from a programmer point of view.

In a similar line to other simulators, the present 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, this simulator exploits the ability of Prolog to define *ad hoc* symbols in order to imitate natural language.

Finally, the main advantage of using Prolog for this simulator is its modularity. Our simulator has been thought as the starting point of an open project in which new models of P systems (with new rules and probably new semantics) can be studied. This lead us to consider a programming language which allows to easily add new features to the existing ones with a minimum number of changes.

Another important issue is the choice of a suitable data structure. This 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 with the programming language that is used, as specific techniques related with it have to be applied. A good representation allows a quick transition between configurations and therefore speeds up the simulation.

In order to express in Prolog the basic information of a P system (membrane structure, contents of the regions, and rules), the following formal representation is considered. A given membrane structure is expressed by means of a labelled tree, where:

1. $< >$ is the *position* to denote the root of the tree and it is associated to 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 i-th inner membrane to h.

Note that different membranes with the same label are individualized using their *positions*.

The configuration in one step of the evolution is represented as a set of one-literal clauses, each of them representing a membrane. Hence, in this representation each clause shows the position, identificator[2], multiset of objects and current step of the computation, as well as the P system this 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 t-th step of its evolution the P system, P, has a membrane at position $[pos]$ with identificator Id and m as multiset, we write

$$P \texttt{ :: } Id \texttt{ at } [pos] \texttt{ with } m \texttt{ at\_time } t$$

[2] The identificator is the label plus the polarity, if it exists.

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[3]:

(a) $[x \to u]_h^{\alpha}$; $[x \to u]_h$

P `rule` x `evolves_to` u `in` h `ec` α
P `rule` x `evolves_to` u `in` h

(b) $x[\ ]_{h_1}^{\alpha_1} \to [y]_{h_2}^{\alpha_2}$; $x[\ ]_{h_1} \to [y]_{h_2}$

P `rule` x `out_of` h_1 `ec` α_1 `sends_in` y `of` h_2 `ec` α_2
P `rule` x `out_of` h_1 `sends_in` y `of` h_2

(c) $[x]_{h_1}^{\alpha_1} \to [\ ]_{h_2}^{\alpha_2} y$; $[x]_{h_1} \to [\ ]_{h_2} y$

P `rule` x `inside_of` h_1 `ec` α_1 `sends_out` y `of` h_2 `ec` α_2
P `rule` x `inside_of` h_1 `sends_out` y `of` h_2

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

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

(e) $[x]_{h_1}^{\alpha_1} \to [y]_{h_2}^{\alpha_2}[z]_{h_3}^{\alpha_3}$; $[x]_{h_1} \to [y]_{h_2}[z]_{h_3}$

P `rule` x `inside_of` h_1 `ec` α_1 `divides_into` y `inside_of` h_2 `ec` α_2
`and` z `inside_of` h_3 `ec` α_3
P `rule` x `inside_of` h_1 `divides_into` y `inside_of` h_2
`and` z `inside_of` h_3

(f) $[x \to [u]_{h_1}^{\alpha_1}\ ]_{h_2}^{\alpha_2}$; $[x \to [u]_{h_1}\ ]_{h_2}$

P `rule` x `inside_of` h_2 `ec` α_2 `creates` u `inside_of` h_1 `ec` α_1
P `rule` x `inside_of` h_2 `creates` u `inside_of` h_1

(g) $[x]_{h_1}^{\alpha_1}[y]_{h_2}^{\alpha_2} \to [z]_{h_3}^{\alpha_3}$; $[x]_{h_1}[y]_{h_2} \to [z]_{h_3}$

P `rule` x `inside_of` h_1 `ec` α_1 `and` y `inside_of` h_2 `ec` α_2
`merge_into` z `inside_of` h_3 `ec` α_3
P `rule` x `inside_of` h_1 `and` y `inside_of` h_2
`merge_into` z `inside_of` h_3

3.3 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

[3] If the membrane has h as label and α as polarity, we will write the identificator as h `ec` α.

(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 non-determinism 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 non-determinism of (relatively large) P systems, different authors have imposed several restrictions to 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.

In this simulator we have tried to minimize these constraints, with the hope of covering a broad range of models of P systems. The limits on the number of membranes, the cardinality of the multisets or the number of computation steps are only imposed by the resources of the computer where the simulator is running, but not by the simulator itself. The only restriction imposed to the simulations is that, in case of non-determinism, the simulator only follows *one* of the possible computations[4]. This constraint gives name to the simulator (SCPS, Simulator for Confluent P Systems), since the simulator is thought for performing the simulation of *confluent* P systems (that is, systems where, for a given input, all the computations produce the same output). Let us remark that *all* recognizer P systems are confluent, regardless the P system model, so our simulator can be considered a powerful tool for studying cellular solutions of decision problems, in spite of this restriction.

In the same line of the first generation of simulators reported in [16], the purpose of designing this simulator is to get information about the evolution of the system that is simulated and, therefore, we are interested in describing the intermediate steps and configurations.

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 it is explained below.

- **Step 1: Initialization**. At the beginning of each computation step, all the membranes are set to *applicable* and their objects are split into three multisets: one **usable** multiset, containing all the objects of the initial membrane and two empty multisets: the **tried** multiset and the **used** multiset.

[4] Current studies are looking for heuristics to decide which is a "good branch" to follow in the computation tree; see, for example [14].

- **Step 2:** For each *applicable* membrane we perform the following steps:
  - **(2.a) Evolution:** For each object a in the **usable** multiset, we look for an evolution rule which can be triggered by a. If the rule exists, all the copies of the object are deleted from the **usable** multiset and the objects obtained by the application of the rule are added to the **used** multiset. If that rule does not exist, all the copies of a are added to the **tried** multiset. After this step, the membrane remains *applicable*, i.e., new rules can be applied to this membrane. If several evolution rules can be triggered by the same object, the simulator only considers the first rule found.
  - **(2.b) Creation:** The process is similar to that from step 2.a. For each object a in the **usable** multiset, we look for a creation rule which can be triggered by a. If the rule exists, all the copies of the object are deleted from the **usable** multiset and the new membranes are placed in the **used** multisets of the corresponding regions. If such rules do not exist, all the copies of a are added to **tried** multiset. After this step, the membrane remains *applicable*, i.e., new rules can be applied on this membrane. If several creation rules can be triggered by the same object, the simulator only considers the first rule found.
  - **(2.c) Send-in:** In the current version of the simulator, the semantics of this rule is quite different from the evolution and creation rules (steps 2.a and 2.b) since only one object is allowed to cross out the membrane. In this step we look for a send-in rule applicable to the membrane. If that rule exists, the object which triggers the rule is deleted from the father membrane, the new object is added to the **used** multiset, and the membrane turns into *no-applicable* mode. If such rules do not exist, we follow with step 2.d.
  - **(2.d) Dissolution, Send-out and Division rules:** In spite of these three rules are pretty different, they have common features and can be grouped in the algorithm: all of them can be applied only once, the multiset of the father membrane is not involved in the application (in the opposite way of the send-in rule), and they all turn the membrane into *no-applicable* mode. In this case we look for (a) a dissolution rule, (b) a send-out rule, and (c) a division rule, in that order. If a rule of such kind is found, a copy of the object which triggers the rule is removed from the **usable** multiset, the rule is applied (i.e., the membrane is dissolved, a new object is sent to the father membrane, or the membrane is divided) and the membrane (or membranes, if division) turns into *no-applicable* mode.
- **Step 3:** *(Merging rules)* When all the membranes have been examined, some of them may remain in the *applicable* mode (if no rule have been applied or only evolution or creation ones). Then, for each merging rule we look for membranes in *applicable* mode that can be merged. If such membranes are found, they will be merged. After this step, all the membranes, merged or not, turn into *no-applicable* mode.
- **Step 4:** After step 3, a cellular step is completed. The simulator takes all the membranes and store them as a new configuration (with `at_time` incremented

by 1). At this point, the P system is ready for a new evolution step, so we go again to step 1 and start a new cellular step.
- **End of computation**. If there are no rules to be applied to any membrane of a configuration, then the evolution finishes (the P system *halts*).

This is the basic algorithm. Obviously, the simulator has other technical features oriented to speed up the computation, but we will not get into details about the code here. We also provide different levels of verbosity with respect to the information supplied to the user, that are illustrated in the next subsection.

3.4 Interacting with the simulator

In order to launch the simulator, we open a Prolog interpreter and we load the main file `scps.pl`. Then, we have to load the information describing the P system that we want to simulate (set of rules and initial configuration, including input multiset if it is the case). This information is provided to the system in a text file containing the corresponding Prolog literals. Users can easily create such files following the guidelines hinted in Subsection 3.2, but the simulator also includes a tool that is able to generate automatically the files containing the set of rules and the initial configuration of recognizer P systems with active membranes which solve several **NP**-complete problems (in the current distribution, the available options are Subset Sum, Knapsack and Partition).

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 initial files. Also, if we are interested in running a simulation for a different instance with the same *size*, the file containing the set of rules can be used again, and we only need to generate the initial configuration for the new input multiset via the instruction `generate_initial`.

Once the P system is loaded, we can choose between several possible commands that provide information about the evolution of the system. For example, we can let the simulator run internally, getting only information about the number of cellular steps of the computation and the output of the P system. This is done using the command `go`.

```
?- go(p1).
The P system p1 stops after the step 56 and returns no.
```

Also, one can ask the simulator to show the configurations step-by-step until a given instant of the computation. If we already went until the configuration `at_time t`, the Prolog instruction that shows the next configuration is `evolve(p1,t)`, where `p1` is the name of the P system.

Using this instruction, the simulator only stores in memory the last configuration, for the sake of efficiency. There are two variants of this instruction: `secure_evolve` stores all the configurations already simulated, and `evolve_stat` includes in the output that is displayed not only the configuration but also the

number of membranes and information related to the used rules. Besides, the simulator informs if any objects have been sent out to the environment.

```
?- evolve_stat(p2,10).

p2 :: e ec-1 at [5] with [p-12, a-24, q0-1] at_time 10
p2 :: e ec-1 at [6] with [p-14, a-22, q0-1] at_time 10
p2 :: e ec-1 at [3] with [p-20, a-14, q0-1] at_time 10
p2 :: e ec-1 at [2] with [p-26, a-6, q0-1] at_time 10
p2 :: e ec 0 at [4] with [a-16, p-19, q1-1] at_time 10
p2 :: e ec 0 at [7] with [a-29, p-8, q1-1] at_time 10
p2 :: e ec 0 at [8] with [a-31, p-6, q1-1] at_time 10
p2 :: e ec 0 at [10] with [a-39, q0-1] at_time 10
p2 :: r ec-1 at [1] with [b-1, h0-1, d0-1] at_time 10
p2 :: s ec 1 at [] with [# -370, g0-8] at_time 10

The P-system has sent out d1 at step 10

- The P system has 10 membranes at time 10
- Used rules in the step 10:
  * The rule 25 has been used 3 times
  * The rule 26 has been used 4 times
  * The rule 27 has been used 3 times
  * The rule 28 has been used 5 times
  * The rule 60 has been used only once
  * The rule 61 has been used only once
```

4 Comparing the Simulators

In this section, we shall *compare* the different performances obtained when simulating cellular solutions to hard problems using either the SCPS simulator that we are introducing in this paper, the previous Prolog simulator presented in [13], or the CLIPS simulator presented in [24]. Note that the statistics that will be shown only intend to illustrate some differences between the three simulators concerning their *speed*, but there are many other parameters that should be taken into account for a better comparison.

In order to get a battery of examples we have used a family of recognizer P systems solving the Partition problem (see [15, 26] for details about the design of the cellular solution that will be simulated).

The Partition problem is known to be **NP**–complete, but it is possible to find cellular solutions (i.e., solutions by means of families of P systems) that run in polynomial time (i.e., that perform a polynomial number of cellular steps).

The underlying idea is to use membrane division to generate in a polynomial time an exponential number of membranes that are able to work in parallel. Of

course, when we try to simulate such solutions on a sequential computer we lose the massive parallelism and therefore the time (number of sequential steps) is not polynomial anymore. However, we have chosen to simulate such families of P systems because there are huge amounts of information to be handled during the simulation (exponentially many objects evolving in an exponential number of membranes), and in this way we can really take the simulators to the limit of their capability.

Before going on, let us recall that the Partition problem is the following one: *Consider a set S with N elements. The weight of each element s in S is a non-negative integer w(s), where the weight function, w, is additive. Determine if S can be split into two subsets S1 and S2 such that the sum of the weights of their elements is the same.*

Please note that the important parameters for each instance are the constant N, that determines the *size* of the instance, and the list of weights, that will indicate the number of objects of the initial multiset (numbers are encoded in a 1-ary way). Actually, all the instances of the same size are processed by the same P system (fixed set of rules and membrane structure), although the input multisets that encode the specific list of weights are different for each instance.

The procedure that we have followed is to take an instance P system and run the corresponding three simulations, asking the simulators to go through all the computation and to show only the final output. Then, we record the time spent by each simulation[5] and we pass to another P system.

It is worth mentioning that, in order to interpret the following statistics, one has to take into account that we are comparing the results of running *one* instruction (macro) that tells the simulator to go through all the computation, giving as result of its application the *output* of the corresponding P system. Moreover, we would like to remark that there are many things that influence the resources (time, space) needed by the simulator to run that instruction, mainly:

- Representation of the knowledge.
- Implementation of the application of *one* evolution rule.
- Management of stored information of previous computation steps (keep everything vs. keep only the last configuration).

Indeed, we believe that one of the most important differences between the SCPS simulator and the other ones that we are considering lies in the internal representation of the knowledge. More precisely, the fact that in the first case we do not represent the multisets as lists of symbols, but we use instead a list of pairs (object, multiplicity) will show to be of high relevance in what concerns the size of the instances that can be handled by the simulator. Figure 1 shows the statistics.

The number of membranes that will be generated during the computation only depends on N, but the number (multiplicity) of objects that will be present in the system, evolving during the computation, depends on $w(S)$, as well as the number of cellular steps that have to be simulated (for a given instance, its associated P

[5] We use always the same machine, a Pentium(R) 4 at 2.8GHz, 504 MB of RAM.

system will halt in at most $2N + w(S) + 11$ steps, sending to the environment either *yes* or *no*).

It can be observed how the time (and the number of inferences on the Prolog simulators) increases as we consider larger weights for instances with the same size.

5 Final Comments

In spite of their limitations, the success of the first generation of simulators of P systems is beyond any doubt. They are a useful tool 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 line. On the other hand, it has proved to be a useful assistant tool for the design and formal verification of complex P systems which solve problems, saving the researchers heavy hand-made calculations.

In this paper we go beyond the usual targets in the design of P system simulators. We are not thinking in the particular framework of a specific model of P systems, but having in mind a broad range of such models, even in a non-explored combination of existing ones. For that, we use a language (Prolog) which is able to simulate natural language and, therefore, a user without knowledge of Prolog is able to follow the simulation.

As pointed in [16], one of the common features of the first generation of simulators was the lack of efficiency in favor of the expressivity. In this simulator we keep a high grade of expressivity, but we have tried to make more efficient algorithms and implementations. As the statistics show, the result has been a simulator much faster than the comparable simulators.

As a drawback of the simulator, we can remark that the time of execution (and the number of inferences on the Prolog simulators) increases as we consider larger weights for instances with the same size, but the main constraint is intrinsic to the target of the simulator.

This software is thought to deal with many different kinds of rules and in each step the simulator looks for rules of any possible type. If the number of types of rules increase, the time required for performing each cellular step also increases.

A natural way to avoid this drawback is to ask the user at the beginning of the simulation which are the type of rules of the model and ignore the rest. This question should be revisited in forthcoming releases of the simulator.

Acknowledgement

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

Instances	CLIPS	previous Prolog	SCPS
$N = 4, \quad w = [10, 7, 9, 14]$	3069 rules fired / 1.22 sec	2,316,424 inferences / 1.89+2.01 = 3.90 sec	30,799 inferences / 0.08 sec
$N = 4, \quad w = [18, 27, 50, 35]$	8523 rules fired / 10.81 sec	Out of global stack (step 7)	75,418 inferences / 0.14 sec
$N = 4$, $w = [135, 215, 500, 150]$	cannot generate the initial configuration	Out of global stack (step 2)	523,167 inferences / 1.05 sec
$N = 4$, $w = [2345, 7500, 1916, 6006]$	id.	Out of global stack (first step)	8,842,019 infs. / 17.45 sec
$N = 5, \quad w = [9, 7, 4, 2, 8]$	4329 rules fired / 2.27 sec	5,610,673 inferences / 1.58+3.75+1.72=7.05 sec	48,329 inferences / 0.22 sec
$N = 5, \quad w = [6, 10, 11, 16, 25]$	8943 rules fired / 8.89 sec	Out of global stack (step 9)	83,357 inferences / 0.38 sec
$N = 6, \quad w = [9, 2, 5, 7, 13, 6]$	10704 rules fired / 13.98 sec	Out of global stack (step 10)	112,592 inferences / 0.23 sec
$N = 10$, $w = [3, 5, 10, 7, 1, 2, 8, 1, 4, 2]$	7181.80 sec ($\approx$2 h)	Out of global stack (step 13)	3,246,323 inferences / 10.08 sec
$N = 10$, $w = [340, 500, 107, 777, 221, 192, 850, 199, 444, 250]$		Out of global stack (first step)	109,229,897 infs. / 624.45 sec

Fig. 1. Statistics of the comparisons

References

1. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. de Mingo: A recursive algorithm for describing evolution in transition P systems. In *Pre-Proceedings of Workshop on Membrane Computing* (C. Martín-Vide, Gh. Păun, eds.), Curtea de Argeş, Romania, August 2001. Technical Report GRLMC 17/01, Rovira i Virgili University, Tarragona, Spain (2001), 19–30.
2. 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 *Multiset Processing. Mathematical, Computer Science and Molecular Computing Points of View* (C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2235, Springer-Verlag, Berlin, 2001, 1–16.
3. F. Arroyo, C. Luengo, A.V. Baranda, L.F. de Mingo: A software simulation of transition P systems in Haskell. In *Membrane Computing WMC-CdeA 2002* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), LNCS 2597, Springer-Verlag, Berlin, 2003, 19–32.
4. A.V. Baranda, J. Castellanos, F. Arroyo, R. Gonzalo: Data structures for implementing P systems in silico. In *Pre-proceedings of Workshop on Multiset Processing* (C.S. Calude, M.J. Dinneen, Gh. Păun, eds.), Curtea de Argeş, Romania, CDMTCS TR 140, Univ. of Auckland, 2000, 21–34.
5. A.V. Baranda, J. Castellanos, R. Gonzalo, F. Arroyo, 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.
6. 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 *DNA Computing DNA 7* (N. Jonoska, N.C. Seeman, eds.), LNCS 2340, Springer-Verlag, Berlin, 2002, 350–359.
7. D. Balbontín-Noval, M.J. Pérez-Jiménez, F. Sancho-Caparrini: A MzScheme Implementation of Transition P Systems. In *Membrane Computing 2002* (Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds.), LNCS 2597, Springer-Verlag, Berlin, 2003, 58–73.
8. I. Bratko: *PROLOG Programming for Artificial Intelligence*. Third edition. Addison-Wesley, 2001.
9. G. Ciobanu, D. Paraschiv: P system software simulator. *Fundamenta Informaticae*, 49, 1-3 (2002), 61–66.
10. G. Ciobanu, G. Wenyuan: P systems running on a cluster of computers. In *Membrane Computing WMC 2003* (C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2933, Springer-Verlag, Berlin, 2004, 123–139.
11. A. Cordón-Franco, M.A. Gutiérrez Naranjo, M.J. Pérez Jiménez, A. Riscos Núñez, Sancho-Caparrini: Implementing in Prolog an effective cellular solution to the Knapsack problem. In *Membrane Computing WMC 2003* (C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2933, Springer-Verlag, Berlin, 2004, 140 – 152.
12. 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.
13. 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.

14. A. Cordón-Franco, M.A. Gutiérrez Naranjo, M.J. Pérez Jiménez, A. Riscos Núñez: Exploring computation trees associated with P systems. In *Membrane Computing: 5th International Workshop, WMC 2004, Milan, Italy, June 14-16, 2004, Revised Selected and Invited Papers* (G. Mauri, Gh. Păun, M.J. Pérez Jiménez, G. Rozenberg, A. Salomaa, eds.), LNCS 3365, Springer-Verlag, Berlin, 2005, 278 – 286.
15. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez: A fast P system for finding balanced 2-partition. *Soft Computing*, 9, 6 (2005).
16. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez: Available membrane computing software. In *Applications of Membrane Computing* (G. Ciobanu, Gh. Păun, M.J. Pérez Jiménez, eds.), Springer-Verlag, Berlin, in press.
17. C. Izbaşa, C. Bonchiş, C. Garboni, G. Ciobanu: WebPS: A web-based P system simulator with query facilities. In this volume.
18. P.L. Luisi.: The chemical implementation of autopoiesis. In *Self-Production of Supramolecular Structures* (G.R. Fleishaker et al., eds.), Kluwer, Dordrecht, 1994.
19. M. Maliţa: Membrane computing in Prolog. In *Pre-proceedings of the Workshop on Multiset Processing* (C.S. Calude, M.J. Dinneen, Gh. Păun, eds.), Curtea de Argeş, Romania, CDMTCS TR 140, Univ. of Auckland, 2000, 159–175.
20. I.A. Nepomuceno-Chamorro: A Java simulator for basic transition P systems. In *Proceedings of the Second Brainstorming Week on Membrane Computing* (Gh. Păun, A. Riscos, A. Romero, F. Sancho, eds.), Report RGNC 01/04, 2004, 309–315.
21. Gh. Păun: Computing with membranes. Turku Centre for Computer Science, TUCS Technical Report, N.208 (1998).
22. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
23. Gh. Păun: P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, 6, 1 (2001), 75–90.
24. M.J. Pérez-Jiménez, F.J. Romero-Campero: A CLIPS simulator for recognizer P systems with active membranes. In *Proceedings of the Second Brainstorming Week on Membrane Computing* (Gh. Păun, A. Riscos, A. Romero, F. Sancho, eds.), Report RGNC 01/04, 2004, 387–413.
25. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini: Solving VALIDITY problem by active membranes with input. In *Proceedings of the Brainstorming Week on Membrane Computing* (M. Cavaliere, C. Martín-Vide, Gh. Păun, eds.), Report GRLMC 26/03, 2003, 279–290.
26. A. Riscos-Núñez: *Cellular Programming: Efficient Resolution of Numerical NP-complete Problems.* Ph.D. Thesis, University of Seville, 2004.
27. 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.
28. A. Syropoulos, E.G. Mamatas, P.C. Allilomes, K.T. Sotiriades: A distributed simulation of transition P systems. In *Workshop on Membrane Computing 2003* (C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2933, Springer-Verlag, Berlin, 2004, 357–368.
29. `http://www.lpsi.eui.upm.es/nncg/`
30. Logic Programming: `http://www.afm.sbu.ac.uk/logic-prog/`