Chapter V

Cellular Solutions to Some Numerical NP-Complete Problems: A Prolog Implementation

Andrés Cordón-Franco, University of Sevilla, Spain

Miguel Angel Gutiérrez-Naranjo, University of Sevilla, Spain

Mario J. Pérez-Jiménez, University of Sevilla, Spain

Agustín Riscos-Núñez, University of Sevilla, Spain

ABSTRACT

This chapter is devoted to the study of numerical ***NP****-complete problems in the framework of cellular systems with membranes, also called* P systems *(Păun, 1998). The chapter presents* efficient *solutions to the subset sum and the knapsack problems. These solutions are obtained via families of P systems with the capability of generating an exponential working space in polynomial time. A simulation tool for P systems, written in Prolog, is also described. As an illustration of the use of this tool, the chapter includes a session in the Prolog simulator implementing an algorithm to solve one of the above problems.*

INTRODUCTION

The race to miniaturize silicon microchips to get more and more powerful (smaller and faster) processors is expected to hit its own physical limits very soon. This is why it is necessary to look for new *unconventional models* of computation. One of the main research lines in this direction is focusing on obtaining new computational paradigms inspired from various well-established natural phenomena in physics, chemistry, and biology. This approach is generically known as natural computing.

This chapter is part of the framework of one of these nature-inspired models, namely, cellular computing with membranes. This model abstracts from the structure and the functioning of a living cell. At the moment it is just at the theoretical level, and it is not likely that it would be implemented in vivo in the near future. However, some simulations *in silico* (i.e., software implementations) have been recently presented, written in various programming languages (Java, C, Scheme, etc.). Although they are not able to actually implement the massive parallelism inherent to the original model, these approaches may be regarded as a proof of concept for this new computational paradigm in dealing with hard problems and as a tool that is able to support both research and pedagogical purposes.

The simulator presented here is written in Prolog, and it was created with the aim of assisting in theoretical research in cellular computing. That is, it is not intended to get an efficient implementation, but to be an intuitive tool that provides faithful and detailed information about the computations taking place within cellular systems. More interestingly, during the development of this tool simulator, we realised that we needed new information that helped the formal verification process of cellular computing.

From Nature to Membrane Computing

In recent years the research field generically named natural computing has been under enormous scrutiny and development. This discipline has started off the investigation of both mathematical models and technological requirements for the implementation of bio-inspired computing paradigms. The research within this field studies the way nature *computes*, conceiving and abstracting new paradigms and computing models.

There are several areas within natural computing that are now well established. *Genetic algorithms* (or, more generally, *evolutionary computing*), introduced by J. Holland (1975), uses some operations inspired by natural evolution and selection in order to improve the process of finding a good solution in a huge set of feasible candidate solutions. *Neural networks,* introduced by W.S. McCulloch and W. Pitts (1943), were inspired by the interconnections and the functioning of neurons in the brain. *Molecular computing* is a research area concerned with the use of molecules as biological hardware to perform computations. *DNA-based* molecular computing was born when L. Adleman (1994), published a solution to an instance of the Hamiltonian path problem by manipulating DNA strands in a lab.

We should mention here *splicing systems,* a notion introduced by T. Head (1987), which constituted the theoretical precursor of this type of computation. This model is not oriented toward performing computations; it is just a formalization of the DNA strand recombinations via restriction enzymes. DNA computing is a subarea of molecular computing that uses DNA strands to take advantage of the huge parallelism provided by the biochemical reactions occurring in a DNA solution. *Membrane computing* was introduced by Păun (1998), and it is inspired from the structure and the functioning of molecules and cells as living organisms able to process and generate information. Indeed, the cells contain different vesicles, each of them delimited by membranes leading to a hierarchical structure. Inside of these vesicles some chemical reactions involving biochemical substances take place, modifying the substances contained in them, but also generating a flow of biochemical elements among different compartments that compose the cell. These processes at the cellular level can be interpreted as computing processes.

When designing a formal system that abstracts the structure and functioning of a cell, there are two ways to follow one can describe, in as much detail as possible, the processes that take place, with the aim of getting a deeper understanding about cells; or one can extract the main characteristics that define a cell, with the intention of obtaining a new computing model — simple but powerful — that allows solving problems that are especially hard in other more classical models. This second approach was the one followed by Păun through transition P systems (1998). Since then, a number of variants of P systems have been considered in the literature (see Păun, 2002, for a comprehensive exposition).

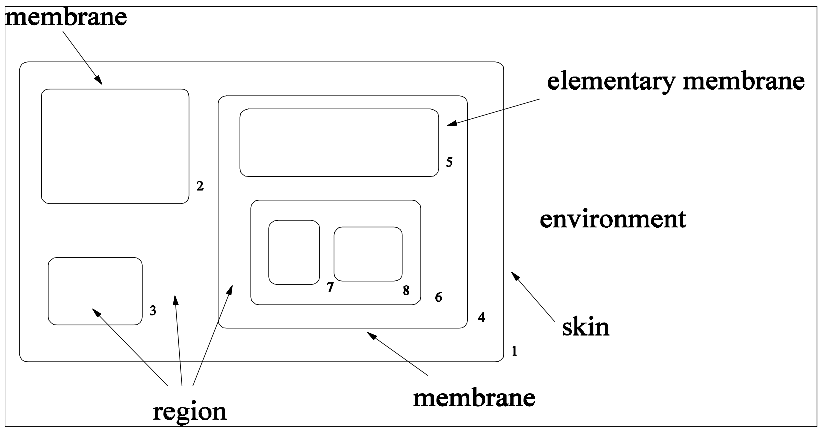
The notion of P systems is directly derived from one of the fundamental components of the cell, the biological membrane. It is well-known that all of the internal compartments that form a cell (even the cell itself) are delimited by membranes. Nevertheless, these membranes do not generate tight compartments, but they allow biological substances to pass through them, most of the time in a selective manner. It is inside these compartments that chemical reactions take place.

Basically, a P system consists of a set of membranes, usually organized in a hierarchical structure (Figure 1). There exists a *skin membrane*, which embraces all the others, separating the system from the external environment. The membranes that do not contain other membranes inside are called *elementary membranes*. The *regions* delimited by the membranes (that is, the space bounded by a membrane and the immediately lower membranes, if there are any) can contain multiple copies of certain objects. By means of fixed evolution rules associated with the membranes (or regions), these objects can evolve producing new objects, and can even travel from one region to an adjacent one, crossing the membranes that separate the system's compartments. Transition P systems offer two levels of parallelism: on the one hand, the rules within a membrane are applied simultaneously; on the other hand, these operations are performed in parallel in all of the regions of the system.

Each region can be seen as a computing unit (a processor), having its own data (biological substances) and its local program (given by biochemical reactions). So, the cell can be seen as an *unconventional computing device*.

Observe that a cellular computing system with membranes is not described through a sequence of basic operations capable of being sequenced over an input data in order to obtain a final result. Instead, a P system is a device whose execution, as for Turing machines, modifies the content of the distinct components that form it until reaching, if so, a halting state, when the system halts. In that sense, this execution could be called *user-independent* since, once the system is constructed, it is not necessary, in principle, to guide it.

Figure 1. Structure of a P system



Prolog

The evolution of a P system has a lot of similarities with the execution of a production system based on rules and, because of that, it seems natural to consider a declarative language to simulate cellular computations. In this work we use Prolog (PROgrammation en LOGique) to design a tool for representing and experimenting with P systems in an effective way.

Prolog is a programming language based on clausal logic together with a mechanism of theorem proving (clause resolution). A Prolog program can be regarded as a set of first-order sentences expressing *facts* and *rules*. Providing a comprehensive description of the Prolog language is, of course, beyond the scope of this work (a good starting point can be Bratko, 2001, or *www.afm.sbu.ac.uk/logic-prog/*).

Organization of the Chapter

The chapter is organized as follows: the following section is devoted to briefly describing the class of P systems used in this chapter. In the next section, two families of P systems with active membranes that solve *in linear time* the subset sum and knapsack problems, respectively, are presented and a detailed overview of the computation is given.

The second part of the chapter is devoted to the simulation of the P systems. One section is dedicated to our simulator for P systems with active membranes. After a short presentation about the way of representing P systems in Prolog, the algorithm of the inference engine of the simulator is briefly presented. Finally, we include a subsection showing a Prolog session of the simulator performing the evolution of an instance of the knapsack problem. Interested readers can find more details about the way the simulator works in Cordón Franco et al. (2004).

At the end of the chapter, some conclusions and final remarks are discussed.

RECOGNIZER P SYSTEMS WITH ACTIVE MEMBRANES

Membrane division is inspired from cell division, a well-known process in biology. The replication is one of the most important functions of a cell. In ideal circumstances, by division it is possible to obtain 2^n cells in n steps. That is, membrane division is able to produce an exponential working space in a polynomial time. This is actually the key feature of P systems with active membranes (Păun, 2002). This characteristic of the P systems allows a significant speed-up in the computation process. Indeed, Zandron (2001) has shown that if **P≠NP**, then a deterministic P system without membrane division is not able to solve an **NP**-complete problem in polynomial time (moreover, this result was generalized in Pérez Jiménez et al. 2004). This speed-up can be especially relevant when dealing with real world problems, such as optimization algorithms in a factory or algorithms to decrypt an encoded message.

Next, following Păun, we introduce the definition of P systems with active membranes and electrical charges. We consider only 2-division for elementary membranes, and we do not use cooperation or priority among rules (Păun, 2002).

Definition

A *P system with active membranes and electrical charges* is a tuple $\Pi = (\Gamma, H, \mu, w_1, \dots, w_q, R)$ where:

- Γ is a finite alphabet (the working alphabet) whose elements are called objects.
- H is a finite set of labels for membranes.
- μ is a membrane structure of degree q, with labels from H. Membranes have electrical charges (0, $+$ or $-$).
- $w_1, \dots, w_q$ are multisets over Γ describing the multisets of objects initially placed in membranes from μ.
- R is a finite set of developmental rules of the following types:

a) $[_l\, a \rightarrow v]_l^{\beta}$, where $a \in \Gamma$, $v \in \Gamma^*$, $\beta \in \{0, +, -\}$ (object evolution rules). Internal rules associated with membranes and depending on the label and the charge of the membranes. An object a can evolve to a string v without modifying the polarity of the membrane.

b) $[_l\, a\,]_l^{\beta} \rightarrow b[_l\,]_l^{\gamma}$, where $a, b \in \Gamma$, $\beta, \gamma \in \{0,+,-\}$ (communication rules). An object a can get out of a membrane labelled by l and with electrical charge β, possibly transformed in a new one b and, simultaneously, the polarization of the membrane can be changed to γ; its label remains the same.

c) $a\, [_l\,]_l^{\beta} \rightarrow [_l\, b\,]_l^{\gamma}$, where $a, b \in \Gamma$, $\beta, \gamma \in \{0, +, -\}$ (communication rules). An object a can get into a membrane labelled by l and with electrical charge β, possibly transformed in a new one b and, simultaneously, the polarization of the membrane can be changed to γ; its label remains unchanged.

d) $[_l\, a\,]_l^{\beta} \rightarrow b$, where $a, b \in \Gamma$, $\beta \in \{0, +, -\}$, $l \neq skin$ (dissolution rules). An object *a* in a membrane labelled by *l* and with electrical charge β, is transformed in a new one *b* and, simultaneously, in the presence of the object *a,* the membrane is dissolved.

e) $[_l\, a\,]_l^{\beta} \rightarrow [_l\, b\,]_l^{\gamma} [_l\, c\,]_l^{\delta}$, where $a, b, c \in \Gamma$, $\beta, \gamma, \delta \in \{0, +, -\}$, $l \neq$ *skin* (*2*-division rules for elementary membranes). In the presence of an object *a*, the membrane labelled by *l* and with electrical charge β, is divided into two membranes, eventually allowing independent transformation for the element *a* on each one of the resulting membranes (i.e., objects *b* and *c*, respectively), and possibly the two membranes produced have different polarizations (i.e.,γ and δ, respectively).

Let us observe that the rules of the system are associated with labels (for example, the rule $[_l\, a \rightarrow v]_l^{\beta}$ is associated with the label $l \in H$). According to rules of type (e), it follows that there may exist membranes in a system with the same label.

The rules are applied according to the following principles:

1. Their use, according to the general framework of membrane computing, is in a maximal parallel way. In one step, each object in a membrane can only be used by one rule (nondeterministically chosen when there are several possibilities), but any object that can evolve by a rule of any form must do it (with restrictions).
2. If a membrane is dissolved, its content (multiset and interior membranes) becomes part of the immediately external one. The skin membrane is never dissolved.
3. All of the elements that are not involved in any of the rules available remain unchanged.
4. If a division rule is applied to a membrane and, at the same time, some objects inside that membrane evolve through a rule of type (a), then in the two new copies of the membrane we introduce the result of the evolution. That is, we suppose that first the evolution rules of type (a) are used, changing the objects, and then the division is produced, so that in the two new membranes we introduce copies of the changed objects.
5. The rules associated with the label *l* are used for all membranes with this label, whether or not the membrane is an initial one or it was obtained through a division process. At one step, different rules can be applied to

different membranes with the same label, but one membrane can be the subject of *only one* rule of types (b), (c), (d), or (e).
6. The skin membrane can be electrically charged, but can never divide.

Hence, the rules of type (a) are applied in *parallel*; that is, all objects that can evolve by such rules must do it, while the rules of type (b), (c), (d), and (e) are used *sequentially*, in the sense that *one* membrane can be used by, at most, one rule of these types in each step (time unit).

In this work we deal with decision problems, and therefore for each instance of the problem, we are only interested in a binary answer (*yes* or *no*). In this line, there is an underlying similarity between solving a decision problem and solving a recognition problem for a certain language; deciding if an instance has an affirmative or negative solution is equivalent to deciding if a word belongs to the language or not.

We present in this section an adaptation of P systems as devices that accept languages and decide upon some properties. These are called *recognizer P systems*. This adaptation will allow us to efficiently attack some **NP**-complete problems.

Definition

A *P system with input* is a tuple (Π, Σ, i_Π), where:

- Π is a P system, with working alphabet Γ, with q membranes labelled by $1, \ldots, q$, and initial multisets $w_1, \ldots, w_q$ associated with them.
- Σ is an (input) alphabet strictly contained in Γ.
- The initial multisets are over $\Gamma - \Sigma$.
- i_Π is the label of a distinguished (input) membrane.

Let m be a multiset over Σ. The initial configuration of (Π, Σ, i_Π) with input m is $(\mu, w_1, \ldots, w_{i\Pi} \cup m, \ldots, w_q)$. We have thus defined a class of P systems that receive an input before starting the computations. As we intend to consider P systems as black boxes, where we introduce an input and wait for the answer without knowing about the inner processes, we can agree that the output of the computation will be collected outside the P system (through the application of rules of type (b) in the skin membrane). This leads to P systems with input and with external output, which is the variant in which our recognizer systems are included.

Definition

A *recognizer P system with active membranes* is a P system with active membranes, with input and with external output, (Π, Σ, i_Π), such that:

- The working alphabet contains two distinguished elements: *yes*, *no*.
- All of its computations halt.
- If $\boldsymbol{C}$ is a computation of Π, then either an object *yes* or an object *no* (but not both) has to be sent out to the external environment, and only in the last step of the computation.

Let Π be a recognizer P system, and let $\boldsymbol{C}$ be a computation of Π. Let w be the multiset of objects that have been sent out during the computation $\boldsymbol{C}$. We define then the function *Output* as $Output(\boldsymbol{C}) = yes$ if $yes \in w$ and $Output(\boldsymbol{C}) = no$ otherwise.

We say that $\boldsymbol{C}$ is an *accepting* computation (respectively, *rejecting* computation) if the object *yes* (respectively, *no*) appears in the external environment associated with the corresponding halting configuration of $\boldsymbol{C}$—that is, if $yes = Output(\boldsymbol{C})$ (respectively, $no = Output(\boldsymbol{C})$).

Let us note that a recognizer P system is a *confluent* system in the following sense: every computation with the same initial configuration has the same output.

LINEAR SOLUTIONS TO NP-COMPLETE PROBLEMS

Most of the P systems from the current literature address questions from a formal language theory angle, mainly concerned with universality results and solutions to classical **NP**-complete problems such as SAT or validity (Păun, 2002). We are interested not only in solving new problems, but also in doing so in an *efficient* way. This is why we have chosen P systems with active membranes because, as mentioned in the previous section, it is a model that allows fast solutions to hard problems. We shall focus on the following numerical NP-complete decision problems:

- *Subset sum problem*. Given a finite set, A, a weight function, $w: A \rightarrow \mathbf{N}$, and a constant $k \in \mathbf{N}$, determine whether or not there exists a subset $B \subseteq A$ such that $w(B)=k$.
  It is clear that we can assume that $k \leq w(A)$; otherwise, the solution is always negative.

- *Decision knapsack problem (0/1 bounded version)*. Given a knapsack of capacity $k \in \mathbf{N}$, a finite set, A, a weight function, $w: A \to \mathbf{N}$, a value function, $v: A \to \mathbf{N}$, and a constant, $c \in$ N, decide whether or not there exists a subset of A such that its weight does not exceed k and its value is greater or equal than c.

We shall use tuples $(n, (w_1, \dots , w_n), k)$ and $(n, (w_1, \dots, w_n), (v_1, \dots, v_n), k, c)$ to represent the instances of the subset sum problem and the knapsack problem, respectively, where n stands for the size of $A = \{a_1, \dots, a_n\}$, $w_i = w(a_i)$, $v_i = v(a_i)$ and k and c are the constants mentioned above.

We shall solve these problems via brute-force algorithms, using recognizer P systems with active membranes with 2-division but without dissolution rules. The solution to these problems may be broken down into several stages:

- *Generation stage*: a single specific membrane for every subset of A is generated via membrane division.
- *Calculation stage*: in each membrane the functions w for the subset sum problem or w and v for the knapsack one are evaluated over the associated subset. This stage will take place in parallel with the previous one.
- *Checking stage*: in each membrane, it is checked whether or not the conditions of the problem for each function are satisfied ($w(B) = k$ for the subset sum problem or $w(B) \leq k$ and $v(B) \geq c$ for the knapsack one). This stage cannot start in a membrane before the previous ones are over in *that* membrane.
- *Output stage*: when the previous stages have been completed in *all* membranes, the system sends out the answer (*yes* or *no*) to the environment.

We shall introduce two families of recognizer P systems with active membranes using 2-division solving in *linear* time the subset sum problem and the knapsack problem.

The Subset Sum Problem

The P systems presented here have a recursive description with respect to the two parameters n and k. In order to express the family using only one parameter (i.e., $\mathbf{\Pi} = \{\Pi(t) : t \in \mathbf{N}\}$), we shall use the polynomial-time computable bijection $\langle n, k \rangle = ((n+k)(n+k+1)/2)+n$.

For each $(n, k) \in \mathbf{N}^2$ we consider the P system $(\Pi(\langle n, k\rangle), \Sigma(n, k), i(n, k))$, where the input alphabet is $\Sigma(n, k) = \{x_1, ..., x_n\}$, and $\Pi(\langle n, k\rangle) = (\Gamma(n, k), \{\mathrm{e}, \mathrm{s}\}, \mu, m_{\mathrm{s}}, m_{\mathrm{e}}, R)$ is defined as follows:

- Working alphabet: $\Gamma(n,k) = \{e_0,..., e_n, z_0,..., z_{2n+2k+2}, q, q_0,..., q_{2k+1}, x_0,..., x_n, a_0, a, \bar{a}_0, \bar{a}, yes, no, \#, d_+, d\_\}$.
- Membrane structure: $\mu = [_{\mathrm{s}} [_{\mathrm{e}}]_{\mathrm{e}}]_{\mathrm{s}}$
- Initial multisets: $m_{\mathrm{s}} = z_0$; $m_{\mathrm{e}} = e_0 \bar{a}^k$
- Input membrane: $i(n, k) = \mathrm{e}$
- The evolution rules from the set R are listed below, classified into several groups that are briefly commented:

(R1) $[_{\mathrm{e}}\, e_i\,]_{\mathrm{e}}^0 \to [_{\mathrm{e}}\, q\,]_{\mathrm{e}}^- [_{\mathrm{e}}\, e_i\,]_{\mathrm{e}}^+$, for $i = 0, ..., n$,
$[_{\mathrm{e}}\, e_i\,]_{\mathrm{e}}^+ \to [_{\mathrm{e}}\, e_{i+1}\,]_{\mathrm{e}}^0 [_{\mathrm{e}}\, e_{i+1}\,]_{\mathrm{e}}^+$, for $i = 0, ..., n-1$

The goal of these rules is to generate one membrane for each subset of A. When an object e_i $(i < n)$ is present in a neutrally charged membrane, we pick the element a_i for its associated subset and divide the membrane. In the new membrane where q appears, no further elements will be added to the subset, but the other new membrane must generate membranes for other possible subsets that are obtained by adding elements of index $i+1$ or greater.

(R2) $[_{\mathrm{e}}\, x_0 \to \bar{a}_0\,]_{\mathrm{e}}^0$; $[_{\mathrm{e}}\, x_0 \to \lambda\,]_{\mathrm{e}}^+$; $[_{\mathrm{e}}\, x_i \to x_{i-1}\,]_{\mathrm{e}}^+$, for $i = 1, ..., n$.

At the beginning, in the input multiset that is introduced before starting the computation, objects x_j (with $1 \le j \le n$) encode the weights of the corresponding elements of A: for each a_j we have w_j copies of x_j. Together with elements added to the subset associated with the membrane, these rules calculate the weight of such a subset.

(R3) $[_{\mathrm{e}}\, q \to q_0\,]_{\mathrm{e}}^-$; $[_{\mathrm{e}}\, \bar{a}_0 \to a_0\,]_{\mathrm{e}}^-$; $[_{\mathrm{e}}\, \bar{a} \to a\,]_{\mathrm{e}}^-$.

The occurrence of the objects q_0, a_0, and a marks the beginning of the checking stage. The multiplicity of object a_0 encodes the weight of the associated subset, and the constant k is represented by the number of objects a.

(R4) $[_{\mathrm{e}}\, a_0\,]_{\mathrm{e}}^- \to [_{\mathrm{e}}\,]_{\mathrm{e}}^0\, \#$; $[_{\mathrm{e}}\, a\,]_{\mathrm{e}}^0 \to [_{\mathrm{e}}\,]_{\mathrm{e}}^-\, \#$.

We compare the number of occurrences of objects a and a_0, sending them out of the membrane alternatively and changing the polarity of the membrane each time. The two rules of this group describe this *checking loop*.

(R5) $[_e\, q_{2j} \to q_{2j+1}\,]_e^-$, for $j = 0, ..., k$; $[_e\, q_{2j+1} \to q_{2j+2}\,]_e^0$, for $j = 0, ..., k–1$.

Objects q_i, with $0 \le i \le 2k+1$, act as a counter for the checking stage, controlling the number of checking loops that take place.

(R6) $[_e\, q_{2k+1}\,]_e^- \to [_e\,]_e^0\, yes$; $[_e\, q_{2k+1}\,]_e^0 \to [_e\,]_e^0\, \#$; $[_e\, q_{2j+1}\,]_e^- \to [_e\,]_e^-\, \#$, for $j = 0, ..., k–1$.

Finally, these rules use the information given by the counter to deal with the different checking results — the same number of objects a_0 and a, objects a_0 in excess, or more a objects than a_0.

(R7) $[_s\, z_i \to z_{i+1}\,]_s^0$, for $i = 0, ..., 2n+2k+1$; $[_s\, z_{2n+2k+2} \to d_+\, d_-\,]_s^0$.

There is another counter in the skin membrane that waits for all membranes to finish their checking stage and then releases objects d_+ and $d\_$ in the skin.

(R8) $[_s\, d_+\,]_s^0 \to [_s\,]_s^+ d_+$; $[_s yes\,]_s^+ \to [_s\,]_s^0\, yes$; $[_s\, d_- \to no\,]_s^+$; $[_s\, no\,]_s^+ \to [_s\,]_s^0\, no$.

The answering process is now activated: first the object d_+ acts as a query, changing the polarity of the skin membrane, and then any possible object *yes* that may be present in the membrane is sent out (notice that there is no conflict because in this moment there are no objects *no* present in the skin, since the rule $d\_ \to no$ needs a positive charge to be applied).

Let us recall that the evolution rules of $\Pi(\langle n, k \rangle)$ are defined in a recursive manner from the instance u. Furthermore, the necessary resources to build $\Pi(\langle n, k \rangle)$ from a given instance $u = (n, (w_1 ,..., w_n), k)$ are the following:

- Size of the alphabet: $4n+4k+17 \in \Theta\ (n+k)$
- Number of membranes: $2 \in \Theta\ (1)$
- $|m_s|+|m_e| = k+2 \in \Theta\ (k)$
- Sum of the lengths of the rules: $35n+27k+110 \in \Theta\ (n+k)$

At this point, we would like to remark that the values of the main parameters that determine the size of an instance (n and k) are encoded in the system in a unary fashion. Indeed, n is the number of different objects $e_i (i \neq 0)$ that belong to the alphabet and k is the number of copies of $\bar{a}$ that are present in the inner membrane at the beginning of the computation.

Please note that the weights of the elements of A are also introduced in an unary fashion, through the input multiset. However, these weights do not influence the amount of resources needed to build the system, nor the upper bound for the number of steps of any computation. Indeed, let us (informally) calculate this bound.

There is no synchronization among membranes for the generation stage. This stage starts in the first step of the computation for the unique inner membrane of the system, and then the new membranes created by division will evolve independently. However, it can be proved that after $2n+1$ steps no more divisions will take place in the system. The last membrane to leave the generation stage is the one whose associated subset is $B = A$. It is clear that this membrane is also the last one to finish the calculation stage (in the same step). After an additional step is performed, for renaming purposes, the membrane associated with A appears.

Next, in order to complete the third stage, and regardless of its associated subset, each membrane has to perform at most $2k$+2 steps (see rules (R4), (R5), and (R6)). The exact number of steps is less if $w(B) < k$, but we are looking for an upper bound. Therefore, after $2n$+$2k$+4 steps all of the inner membranes have completed the first three stages.

Now let us focus on the skin membrane for the last stage. The counter z_j is working from the very beginning of the computation, and in the ($2n$+$2k$+3)-th step it evolves into the objects d_- and d_+. The latter object then leaves the system, preparing the skin to send out the final answer (by changing the charge of the skin to positive). In this moment no rules are applicable in any inner membrane, and the only ones that can be applied are the ones from (R8). Thus, the total number of steps will be $2n$+$2k$+5, if the answer is affirmative, or $2n$+$2k$+6, otherwise. In this way we can say that the family $\mathbf{\Pi} = (\Pi(t))_{t \in \mathbf{N}}$ solves the subset sum problem in *linear* time.

Notice that the answer will be sound because all of the checking stages for all of the membranes have been completed (and, thus, all of the subsets have been tested) before checking the presence of an object *yes* in the skin membrane by rules (R8).

The Knapsack Problem (0/1 Bounded Version)

As we did in the previous subsection for the subset sum problem, let us present now a family of P systems that solves the knapsack problem. We shall not again discuss the groups of rules; the reader is encouraged to get through the list of rules and to find out how the computation develops in this case.

Now the relevant parameters for the design of the P systems are n, k, and c. We shall consider a computable polynomial bijection between $\mathbf{N}^3$ and $\mathbf{N}$ (e.g., the one induced by the pairing function $\langle$ x, y, z$\rangle = \langle$ x, $\langle$ y, z$\rangle\,\rangle$).

The family presented here is $\mathbf{\Pi} = \{(\Pi(\langle n, k, c\rangle), \Sigma(n, k, c), i(n, k, c)) : (n, k, c) \in \mathbf{N}^3\}$, where the input alphabet is $\Sigma(\langle nkc\rangle) = \{x_1, \ldots, x_n, y_1, \ldots, y_n\}$, and the P system $\Pi(\langle n, k, c\rangle) = (\Gamma(n, k, c), \{\mathrm{e}, \mathrm{s}\}, \mu, m_\mathrm{s}, m_\mathrm{e}, R)$ is defined as follows:

- Working alphabet: $\Gamma(n, k, c) = \{a_0, a, \bar{a}_0, \bar{a}, b_0, b, \underline{b}_0, \underline{b}, \underline{\underline{b}}_0, \underline{\underline{b}}, d_+, d_-, e_0, \ldots, e_n, q, q_0, \ldots, q_{2k+1}, r, r_0, \ldots, r_{2c+1}, x_0, \ldots, x_n, y_0, \ldots, y_n, yes, no, z_0, \ldots, z_{2n+2k+2c+6}, \#\}$.
- Membrane structure: $\mu = [_\mathrm{s}\, [_\mathrm{e}\,]_\mathrm{e}\,]_\mathrm{s}$.
- Initial multisets: $m_\mathrm{s} = z_0$; $m_\mathrm{e} = e_0\, \bar{a}^k\, \underline{b}^c$.
- Input membrane: $i(n, k, c) = \mathrm{e}$.
- Evolution rules, R, with the following rules:

$[_\mathrm{e}\, e_i\,]_\mathrm{e}^0 \to [_\mathrm{e}\, q\,]_\mathrm{e}^- \, [_\mathrm{e}\, e_i\,]_\mathrm{e}^+$, for $i = 0, \ldots, n$.
$[_\mathrm{e}\, e_i\,]_\mathrm{e}^+ \to [_\mathrm{e}\, e_{i+1}\,]_\mathrm{e}^0 \, [_\mathrm{e}\, e_{i+1}\,]_\mathrm{e}^+$, for $i = 0, \ldots, n-1$.

$[_\mathrm{e}\, x_0 \to \bar{a}_0\,]_\mathrm{e}^0$; $[_\mathrm{e}\, x_0 \to \lambda]_\mathrm{e}^+$; $[_\mathrm{e}\, x_i \to x_{i-1}\,]_\mathrm{e}^+$, for $i = 1, \ldots, n$.
$[_\mathrm{e}\, y_0 \to \underline{b}_0\,]_\mathrm{e}^0$; $[_\mathrm{e}\, y_0 \to \lambda]_\mathrm{e}^+$; $[_\mathrm{e}\, y_i \to y_{i-1}\,]_\mathrm{e}^+$, for $i = 1, \ldots, n$.

$[_\mathrm{e}\, q \to r\, q_0\,]_\mathrm{e}^-$; $[_\mathrm{e}\, \bar{a}_0 \to a_0\,]_\mathrm{e}^-$; $[_\mathrm{e}\, \bar{a} \to a\,]_\mathrm{e}^-$; $[_\mathrm{e}\, \underline{b}_0 \to \underline{\underline{b}}_0\,]_\mathrm{e}^-$; $[_\mathrm{e}\, \underline{b} \to \underline{\underline{b}}\,]_\mathrm{e}^-$.

$[_\mathrm{e}\, a_0\,]_\mathrm{e}^- \to [_\mathrm{e}\,]_\mathrm{e}^0\, \#$; $[_\mathrm{e}\, a\,]_\mathrm{e}^0 \to [_\mathrm{e}\,]_\mathrm{e}^-\, \#$.

$[_\mathrm{e}\, q_{2j} \to q_{2j+1}\,]_\mathrm{e}^-$, for $j = 0, \ldots, k$; $[_\mathrm{e}\, q_{2j+1} \to q_{2j+2}\,]_\mathrm{e}^0$, for $j = 0, \ldots, k-1$.

$[_\mathrm{e}\, q_{2j+1}\,]_\mathrm{e}^- \to [_\mathrm{e}\,]_\mathrm{e}^+\, \#$, for $j = 0, \ldots, k$.

$[_\mathrm{e}\, r \to r_0\,]_\mathrm{e}^+$; $[_\mathrm{e}\, \underline{\underline{b}}_0 \to b_0\,]_\mathrm{e}^+$; $[_\mathrm{e}\, \underline{\underline{b}} \to b\,]_\mathrm{e}^+$; $[_\mathrm{e}\, a \to \lambda\,]_\mathrm{e}^+$.

$[_\mathrm{e}\, b_0\,]_\mathrm{e}^+ \to [_\mathrm{e}\,]_\mathrm{e}^0\, \#$; $[_\mathrm{e}\, b\,]_\mathrm{e}^0 \to [_\mathrm{e}\,]_\mathrm{e}^+\, \#$.

$[_{e}\, r_{2j} \rightarrow r_{2j+1}\,]_{e}^{+}$, for $j = 0,..., c$; $[_{e}\, r_{2j+1} \rightarrow r_{2j+2}\,]_{e}^{0}$, for $j = 0,..., c-1$.

$[_{e}\, r_{2c+1}\,]_{e}^{+} \rightarrow [_{e}\,]_{e}^{0}\, yes$; $[_{e}\, r_{2c+1}\,]_{e}^{0} \rightarrow [_{e}\,]_{e}^{0}\, yes$.

$[_{s}\, z_{i} \rightarrow z_{i+1}\,]_{s}^{0}$, for $i = 0,... ,2n+2k+2c+5$; $[_{s}\, z_{2n+2k+2c+6} \rightarrow d_{+}\, d_{-}\,]_{s}^{0}$.

$[_{s}\, d_{+}\,]_{s}^{0} \rightarrow [_{s}\,]_{s}^{+}\, d_{+}$; $[_{s}\, yes\,]_{s}^{+} \rightarrow [_{s}\,]_{s}^{0}\, yes$; $[_{s}\, d_{-} \rightarrow no\,]_{s}^{+}$; $[_{s}\, no\,]_{s}^{+} \rightarrow [_{s}\,]_{s}^{0}\, no$.

Before continuing, we would like to stress the fact that the evolution rules of $\Pi(\langle n, k, c\rangle)$ are described in a recursive manner from the instance u. Let us also list, as we did in the previous section, the resources needed to build $\Pi(\langle n, k, c\rangle)$:

- Size of the alphabet: $5n+4k+4c+28 \in \Theta(n+k+c)$
- Number of membranes: $2 \in \Theta(1)$
- $|m_{s}|+|m_{e}| = k+c+2 \in \Theta(k+c)$
- Sum of the lengths of the rules: $40n+27k+20c+193 \in \Theta(n+k+c)$

Keeping in mind what was discussed about the unary encoding of the parameters and the number of steps of the computations for the subset sum problem, we say that the family presented in this section solves the knapsack problem in *linear* time.

An Overview of the Computation

This section is devoted to explaining the way the P systems described earlier work to solve numerical problems. As the solutions presented for the subset sum and knapsack problems are very similar, we shall only discuss the first.

First of all, recall that to solve an instance $u = (n, (w_{1}, ..., w_{n}), k)$ of the subset sum problem we take the P system $\Pi(\langle n, k\rangle)$ with input $x_{1}^{w1} ... x_{n}^{wn}$. We shall, therefore, refer to such P systems with these inputs from now on.

The purpose of the first stage (generation) is to get a single *relevant* membrane for each subset of A (the concept of relevant membrane is given below). This means 2^{n} different relevant membranes in all.

In the first step of the computation, the rule $[_{e}\, e_{0}\,]_{e}^{0} \rightarrow [_{e}\, q\,]_{e}^{-}\, [_{e}\, e_{0}\,]_{e}^{+}$ is applied. Then the generation and calculation stages continue on in parallel, following the instructions given for the rules (R1) and (R2). These two stages do not end in a membrane as long as an object e_{i} (with $0 \leq i \leq n$) belongs to it and its charge is positive or neutral.

Let us introduce the concept of subset associated with an internal membrane through the following recursive definition:

- The subset associated with the initial inner membrane is the empty one.
- When an object e_j appears in a neutrally charged membrane (with $j < n$), then the j-th element of A is selected and added up to the previous associated subset. Once the stage is over, the associated subset will not be modified anymore.
- When a division rule is applied, the two newborn membranes inherit the associated subset from the original membrane.

We shall also refer to a membrane as associated with its corresponding subset.

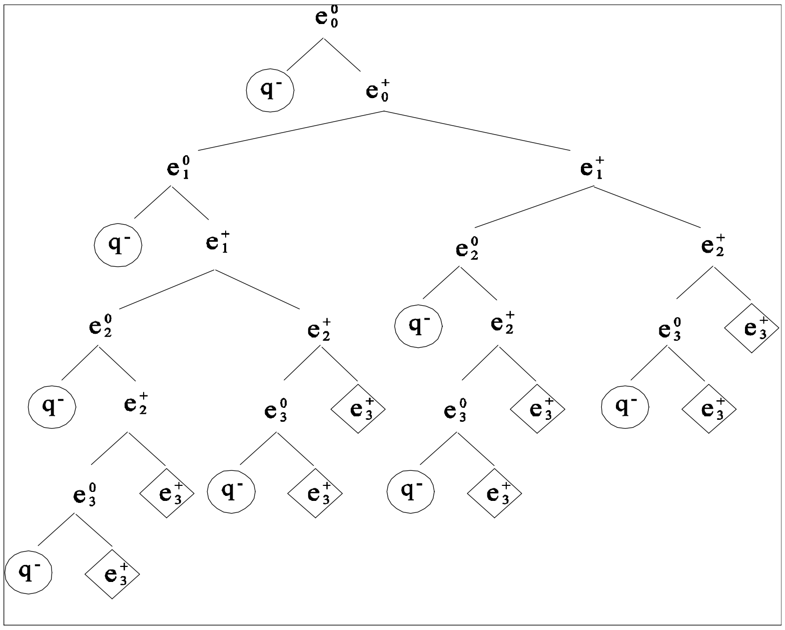
After a division rule $[_e\, e_i\,]_e^0 \rightarrow [_e\, q\,]_e^- \, [_e\, e_i\,]_e^+$ is applied, the two new membranes will behave in a quite different way. On the one hand, in the negatively charged membrane (we have marked such membranes in Figure 2 with a circle), the two first stages end, and in the next step the rules in the group (R3) are applied, renaming the objects to prepare for the third stage. This step is a significant moment, so we shall call *relevant* membranes those that have a negative charge and contain an object q_0. A relevant membrane will not divide anymore during the computation, and its associated subset will remain unchanged. On the other hand, the positively charged membrane will continue in the generation stage. This stage will give rise to membranes associated with subsets that are obtained by adding elements of index i+1 or greater to the current one. Note that if $i = n$, then the membrane cannot continue the generation stage, as there are no rules working for an object e_n in a positively charged membrane (see the membranes surrounded by a diamond in Figure 2). It makes sense that these membranes get blocked, as it is not possible to add elements of indices greater than n.

Thus, as the indices of objects e_i never decrease, we notice that the relevant membranes are generated in a sort of "lexicographic order", in the following sense: if the j-th element of A has already been added to the associated subset, then no element with index lower than j will be added later to the subset associated with that membrane nor to the subsets associated with its descendants. We can check that every subset of A will get a single relevant membrane associated with it, but these membranes are not generated altogether simultaneously. Indeed, we can also check that the membrane corresponding to the subset $\{a_{i1}, ..., a_{ir}\}$ will arise in the $(i_r$+r+2)-th step.

As noted earlier, the first two stages are carried out in parallel. Indeed, there is only a gap of one step of computation between the moment when an element is added to the associated subset and the moment when the new weight of the subset is updated. It can be proved that in a positively charged membrane $[_e\ ]_e^+$ where the object e_i occurs, the multiplicity of object x_1 is equal to the weight of element $a_{i+1} \in A$. Thus, when in the next step we apply (simultaneously) the rules $[_e x_1 \rightarrow x_0]_e^+$, $[_e x_0 \rightarrow \lambda]_e^+$ and $[_e e_i]_e \rightarrow [_e e_{i+1}]_e^0 [_e e_{i+1}]_e^+$, each child membrane will contain exactly $w(a_{i+1})$ occurrences of x_0.

In the case of the neutrally charged child, the element a_{i+1} is added to the associated subset and, simultaneously, the weight of this subset is updated because all of the objects x_0 mentioned above will be transformed into $w(a_{i+1})$ copies of $\bar{a}_0$. The situation is different for the positively charged child, where the element a_{i+1} will not be added to the associated subset or to any of the subsets associated with the descendant membranes. In this case, we are not interested in the weight of the object a_{i+1}, so all the objects x_0 present in the membrane are removed, while $w(a_{i+2})$ new copies of x_0 are created. This procedure goes on until getting into a relevant membrane, and then the number of occurrences of object a_0 will encode exactly the weight of the associated subset and the membrane will be ready to begin the next stage (checking).

Figure 2. Membrane generation for n = *3*



For example, for a_1, after two steps of computation we can see that there are three inner membranes in the configuration, and one of them is a neutrally charged membrane where the object e_1 occurs (see Figure 2). Thus, the element a_1 is added to the associated subset. It can be proven that there are, at that moment, w_1 copies of x_0 in the membrane. So, in the next step, at the same time as the membrane divides, w_1 objects $\bar{a}_0$ will be generated in the membrane by the rules in (R2). These rules will also modify the indices of all the objects x_i, with $i > 0$, so that w_2 copies of x_0 will be ready in the membrane in the next step.

The purpose of the rules in (R4) is to compare the multiplicities of objects a_0 and a — that is, to perform the checking stage for w. They will be counted one by one alternatively, changing the membrane charge each time from negative to neutral and vice versa. In the case of the subset sum problem, the checking is successful if and only if $w(B) = k$, then after $2k$ steps of the checking stage we will not have any objects a or a_0 and the charge will be negative. The counter q_i controls if the number of checking steps is actually equal to $2k$, via the rules in (R6). Let us describe in Table 1 how the third stage works in a membrane associated with a subset B of weight w_B.

Please observe that the index of q_i coincides with the total amount of copies of a and a_0 that have already been erased during the comparison stage.

If $B = \{a_{i1}, ..., a_{ir}\}$ with $i_r \neq n$, then there are in the multiset some objects x_j for $1 \leq j \leq n - i_r$, but they are irrelevant for this stage and, therefore, they are omitted.

If the number w_B of objects a_0 is equal to the number k of objects a, then the result of this stage is successful. If the number of objects a_0 is greater than k, then every time that the rule $[_e q_{2j} \rightarrow q_{2j+1}]_e^-$ is applied (that is, for $j = 0, ..., k$), the rule $[_e a_0]_e^- \rightarrow [_e\,]_e^0 \#$ will also be applied. Thus, we can never find a situation

Table 1.

Multiset	Charge	Parity of q_i
$q_0\, a_0^{w_B}\, a^k$	–	EVEN
$q_1\, a_0^{w_B - 1}\, a^k$	0	ODD
$q_2\, a_0^{w_B - 1}\, a^{k-1}$	–	EVEN
...	...	...
$q_{2j}\, a_0^{w_B - j}\, a^{k-j}$	–	EVEN
$q_{2j+1}\, a_0^{w_B-(j+1)}\, a^{k-j}$	0	ODD
...	...	...

in which the index of the counter q_i is an odd number (namely, $i = 2k+1$) and the charge is negative. This means that the rule that sends out an object *yes* to the skin can never be applied, and even more, the membrane gets *blocked* (it will not evolve anymore during the computation).

Finally, rules in (R7) and (R8) are associated with the skin membrane and complete the output stage. The counter z_i waits before releasing the objects d_+ and d_- to avoid the answer being sent out before all of the inner membranes have finished their checking stages (or have been blocked). The generation and calculation stages will last at most $2n+2$ steps; after those steps, one transition step is performed (rules in (R3)), and the checking process for w is bounded by $2k+1$. This makes $2n+2k+2$ steps in all. After all of these steps are performed, the output process is activated.

In that moment, the skin will be neutrally charged and will contain the objects d_+ and d_-. Furthermore, some objects *yes* will be present in the skin if and only if the checking stages have been successful in at least one inner membrane.

The output stage then begins. First, the object d_+ is sent out, giving positive charge to the skin. Then the object d_- evolves to *no* inside the skin and, simultaneously, if any object *yes* is present in the skin, it will be sent out of the system, giving neutral charge to the skin and making the system stop (in particular, further evolution of object *no* is avoided).

If none of the membranes has successfully passed its checking stage, then any object *yes* will not be present in the skin membrane when the output stage begins. Thus, after generating an object *no*, the skin membrane will still have a positive charge and will be sent out. In that moment, the system halts.

A PROLOG IMPLEMENTATION FOR P SYSTEMS WITH ACTIVE MEMBRANES

Choosing a programming language for implementing a model of computation is not an easy decision. It is necessary to analyze the main difficulties to develop such an implementation and look for a programming language with the adequate features to solve them. As far as our model is concerned, formalizing a configuration of a P system involves dealing with complex data structures. We have to both represent the membrane structure of the P system and make explicit the content of every membrane. Hence, the chosen language has to be expressive enough to handle symbolic knowledge in a natural way.

Moreover, the rules of a P system are nearer to a *production* system than to a list of instructions to be executed in a sequential way. Thus, it seems natural to choose a declarative language (the programmer specifies *what* is to be computed) rather than an imperative one (the programmer specifies *how* something is to be computed).

Prolog is expressive enough to handle symbolic knowledge in a quite natural way and has the ability to evolve different configurations following a set of rules in a declarative style. Aside from the based-tree data structure and the use of infix operators defined ad hoc by the programmer, it allows us to imitate natural language, and the user can follow the evolution of the system without any knowledge of Prolog. We refer to the authors' Web page for a detailed description of the simulator (the Prolog files of the simulator, together with a user manual, are freely available by e-mail from the authors and will soon be found at *www.gcn.us.es*).

In the current version, our Prolog simulator for P systems with active membranes consists of two different parts, as shown in Figure 3. The first part is an *inference engine*. This is a Prolog program that takes as input an initial configuration and the set of rules of a P system and carries out the evolution process associated with the system. Let us emphasize that the inference engine is completely general; that is, it does not depend on the considered P system at all. The second part of the simulator (the program *generator.pl*) provides a tool to automatically build the initial configurations and the sets of rules for instances of some well-known **NP**-complete problems (e.g., SAT, validity, subset sum, and knapsack problems).

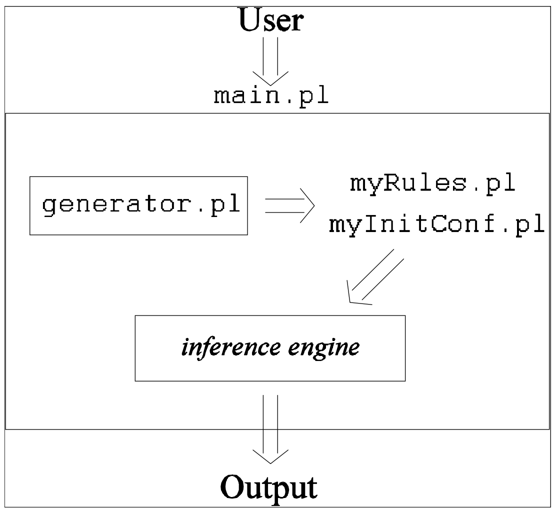
Our simulator looks in the ordered set of rules for all those that are to be applied, carries out the corresponding evolution step, and deterministically obtains a unique new configuration. Consequently, our simulator only ensures a correct simulation of evolutions of deterministic or confluent P systems. Nevertheless, most of the usual algorithms that solve interesting problems are covered.

In what follows, we describe the two parts of our P system simulator (the *inference engine* and the program *generator.pl*). For the sake of simplicity, we shall skip technical details.

Configuration Representation

The first decision related to the design of the simulator is the way to represent the knowledge domain. The problem of representing a model into a programming language is a universal question in computer science, but in this case it has a double face: on one hand, the choice has to allow an efficient

Figure 3. The Prolog simulator



treatment of the data from a programmer's point of view, and on the other hand, the design has to be intuitive enough to be understood by users, regardless of their knowledge of Prolog. These questions lead us to define several infix operators that keep the logical notation of Prolog and allow the reading of the information in English-like sentences. The usual notation for membranes and rules by using subscripts and superscripts has been replaced by usual, plain-text representation in English-like Prolog code.

In order to specify a configuration for a P system with active membranes, it is necessary to explicitly represent the membrane structure and the content of every membrane present in this structure. This involves representing the label, electrical charge, and position of each membrane as well as its content. Moreover, we shall also keep track of the current step of the computation.

In our model, the configuration of a P system in each step of the evolution is a set of one-literal clauses, each of them representing one *alive* membrane (a membrane that is still part of the current system and is not yet dissolved; consequently, it disappears for the next step of computation). Hence, each clause will show a label, an electrical charge, a position, a multiset of objects, and the current step of the computation, as well as the P system to which this membrane belongs.

To achieve this step, the operators ::, ec, at, with and at_time are used. Then, to denote that in the t-th step of its evolution the P system P has a membrane at position [pos] with label h, polarization α, and m as the multiset, we shall write:

```
P :: h ec α at [pos] with m at_time t
```

Rules Description

Next we present the general form of the different types of rules of a P system with active membranes (where P is the name of the current P system):

a) $[_l\, a \rightarrow v]_l^\alpha$ (where $v = v_1 \ldots v_n$)

```
P rule a evolves_to [v1,...,vn] in l ec α.
```

b) $[_l a\,]_l^\alpha \rightarrow b[_l\,]_l^\beta$

```
P rule a inside_of l ec α sends_out b of l ec β.
```

c) $a\, [_l]_l^\alpha \rightarrow [_l b\,]_l^\beta$

```
P rule a out_of l ec α sends_in b of l ec β.
```

d) $[_l\, a\,]_l^\alpha \rightarrow b$

```
P rule a inside_of l ec α dissolves_and_sends_out b.
```

e) $[_1\, a\,]_l^\alpha \rightarrow [_l\, b\,]_1^\beta [_l\, c\,]_1^\gamma,$

```
P rule a inside_of l ec α divides_into
        b inside_of l ec β and c inside_of l ec γ.
```

The Algorithm

The Prolog algorithm to simulate the evolution of a P system works in quite naturally. The input of the program is the initial configuration of the membranes, which is represented as a set of sentences (one-literal clauses with predicate symbol ** and all of them at_time 0) and an appropriate set of rules.

Step 1. *Initialization.* At the beginning, all membranes are set to *applicable*, and for each membrane, their objects are split into two multisets: one *usable* multiset, containing all of the objects of the membrane, and one *used* multiset, which is initially empty.

Step 2. *Transition.* If an applicable membrane enabling the associated rules to be applied exists, then these rules are applied in the following way:

(a)-stage. At this stage, only rules of type (a) are checked one by one. If an object triggers the rule, then it is removed from *usable* to prevent one object being used by two different rules at the same step. The multiset resulting from the application of the rule is added to *used*. After this step, 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)-stage. At this stage, only one rule of types (b), (c), (d), or (e) can be applied. Let us remember that this simulation works with deterministic or confluent P systems. The action depends on the type of rule to be applied:

- *Send out rule*. The element that triggers the rule is removed from the usable multiset and the new one is added to the used multiset of the father membrane. The membrane changes to *not applicable* mode. If the element is sent out of the skin, then it is marked with the *outside* label.
- *Send in rule*. This rule is the opposite of the send out rule. The element that triggers the rule is removed from the usable multiset in the father membrane and the new one is added to the used multiset. The membrane changes to not applicable mode.
- *Dissolution rule*. The element that triggers the rule is removed from the usable multiset and the new element obtained, together with the rest of the elements of the membrane, is added to the used multiset of the father membrane. The membranes inside the dissolved membranes become children of the father membrane.
- *Division rule*. The element that triggers the rule is removed from the usable multiset and the division creates two new membranes in not applicable mode. One of them keeps the original position and the second one gets a new position. The current version of the simulator can also deal with 2-division rules of nonelementary membranes, although this feature falls out of the scope of this work.

End step. 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 moment no membrane is into applicable or not applicable mode. These modes only make sense during the evolution step. Now the P system is ready for a new step of the evolution.

Step 3. End of computation. The evolution of the P system finishes when there are no rules to be applied.

A PROLOG SESSION

In this section we present a brief description of how to use the simulator, which consists of several Prolog files that are designed to run on SWI-Prolog 5.2.0, or above, available from *www.swi-prolog.org*.

In the sequel a session for one instance of the knapsack problem is shown. We consider a set $A = \{a_1, a_2, a_3, a_4\}$ ($n = 4$), with weights $w(a_1) = 3$, $w(a_2) = 2$, $w(a_3) = 3$, and $w(a_4) = 1$, and values $v(a_1) = 1$, $v(a_2) = 3$, $v(a_3) = 3$, and $v(a_4) = 2$. The question is to decide whether or not there exists $B \subseteq A$ such that $w(B) \leq 3$ ($k = 3$) and $v(B) \geq 4$ ($c = 4$). According to the presentation above, the P system that solves this instance is Π (‹4, 3, 4›) with input $x_1^3 x_2^2 x_3^3 x_4 y_1 y_2^3 y_3^3 y_4^2$.

The following two Prolog facts allow us to represent this initial configuration (we have chosen the name p1 to denote the P system that solves this instance of the problem). Note that we use ASCII symbols to represent the objects of the alphabet: a_ stands for $\bar{a}$, b0g stands for $\underline{\underline{b}}$, q_ stands for r, and so on.

```
p1 :: s ec 0 at [] with [z0] at_time 0.
p1 :: e ec 0 at[1] with [e0, a_, a_, a_, b_, b_, b_, b_,
                         x1, x1, x1, x2, x2, x3, x3, x3,
                         x4, y1, y2, y2, y2, y3, y3, y3,
                         y4, y4] at_time 0.
```

The simulator automatically generates these symbols from the data introduced by the user and stores them in a text file. It may be observed that the multiplicities of the objects x_j and y_j correspond to the weights and values of the elements $a_j \in A$, respectively. Also, there are three copies of a_ (k = 3) and four copies of b_ (c = 4).

The simulator also generates the set of rules associated with the instance of the problem. This generation is completed by instantiating several schemes of rules to the concrete values of the parameters. This produces a text file containing the rules that can easily be edited, modified, or reloaded by the user. The set of rules only depends on the parameters n, k, and c. Consequently, if we want to solve several instances with the same parameters, we only need to generate the set of rules once. In this example, we have obtained 89 rules. Some of them are listed in the appendix at the end of this chapter.

To start with the simulation of the evolution of the P system p1 from the time 0, we enter the following command: evolve(p1,0). The simulator returns the configuration at time 1 and the set of used rules indicating how many times they have been used. Moreover, if the skin sends out any object, then this will be reported to the user. The multisets are represented as lists of pairs obj-n, where obj is an object and n is the multiplicity of obj in the multiset.

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

p1 :: s ec 0 at [] with [z1-1] at_time 1
p1 :: e ec -1 at [1] with  [a_-3, b_-4, q-1, x1-3, x2-2, x3-3,
                            x4-1, y1-1, y2-3, y3-3, y4-2] at_time 1
p1 :: e ec 1 at [2] with [a_-3, b_-4, e0-1, x1-3, x2-2, x3-3,
                            x4-1, y1-1, y2-3, y3-3, y4-2] at_time 1

Used rules in the step 1:
  * The rule 1 has been used only once
  * The rule 57 has been used only once
Yes.
```

In this step only rules 1 and 57 have been applied. Rule 1 is a division rule, the membrane labeled by e at position [1] divides into two membranes that are placed at positions [1] and [2], rule 57 is an evolution rule, and z_0 evolves to z_1 in the skin membrane. To obtain the next configuration in the evolution of p1, now we type:

```
?- evolve(p1,1).

p1 :: s ec 0 at [] with [z2-1] at_time 2
p1 :: e ec -1 at [1] with [a-3, bg-4, q0-1, q_-1, x1-3,
            x2-2, x3-3, x4-1, y1-1, y2-3, y3-3, y4-2] at_time 2
p1 :: e ec 0 at [2] with [a_-3, b_-4, e1-1, x0-3, x1-2,
            x2-3, x3-1, y0-1, y1-3, y2-3, y3-2] at_time 2
```

```
p1 :: e ec 1 at [3] with [a_-3, b_-4, e1-1, x0-3, x1-2,
          x2-3, x3-1, y0-1, y1-3, y2-3, y3-2] at_time 2

Used rules in the step 2:
  * The rule 6 has been used only once
  * The rule 14 has been used 3 times
  * The rule 15 has been used 2 times
          ...
  * The rule 22 has been used only once
  * The rule 25 has been used 4 times
  * The rule 26 has been used 3 times
  * The rule 58 has been used only once
```

In this step the first relevant membrane, associated with the empty set, appears at position [1]. The membrane at position [2] will continue dividing to generate new membranes. All of the subsets associated with the membrane's descendant membranes will contain the element a_1. On the other hand, the membrane at position [3] is responsible of the membranes corresponding to all the nonempty subsets that do not contain a_1. Notice that rules from (R2) have been applied (see appendix). That is, the calculation stage has already begun.

The purpose of the first two stages of the system is to generate a single relevant membrane for each subset of A (i.e., $2^n = 2^4 = 16$ relevant membranes in all) and, in parallel, to calculate the weight and the value of such subsets. Other nonrelevant membranes are generated as well, due to technical reasons. These membranes are generated in the first $2n+2 = 10$ steps of the computation.

We can go directly to the configuration at time 10, skipping the intermediate steps, by typing the following command:

```
?- configuration(p1,10).

       ...
p1 :: e ec -1 at [12] with [a-2, a0-2, b0g-5, bg-4, q2-1, q_-1]
                                                    at_time 10
       ...
p1 :: e ec -1 at [24] with [a-3, a0-9, b0g-9, bg-4, q0-1, q_-1]
                                                    at_time 10
       ...
```

Observe that the relevant membrane associated with the total subset appears at position [24]. It is the last relevant membrane to be generated; that is, no more division will take place in the rest of the computation and no new relevant membranes will appear.

Let us focus on the membrane at position [12]. This membrane encodes the subset $\{a_2, a_4\}$, which is the only solution for the instance considered in our example. Two steps of the checking stage for *w* have already been carried out (note the counter q2). In the next steps the membranes perform their checking stages (for *w* and for *v*) that mainly consist of applying rules 27 and 28 (for *w*) and rules 44 and 45 (for *v*). Let us focus now on the output stage.

```
?- configuration(p1,26).

p1 :: s ec 0 at [] with [# -127, z26-1] at_time 26
         ...
p1 :: e ec 0 at [12] with [q_9-1] at_time 26
         ...
```

The inner membrane at position [12] is now ready to send an object yes to the skin membrane (see rule 56 in the appendix).

```
?-  evolve(p1,26).

p1 :: s ec 0 at [] with [# -127, yes-1, z27-1] at_time 27
           ...
p1 :: e ec 0 at [12] with [] at_time 27
           ...

Used rules in the step 27:
  * The rule 56 has been used only once
  * The rule 83 has been used only once
```

Due to technical reasons, the counter in the skin will wait two more steps before releasing the special objects d_+ and d .

```
?- configuration(p1,29).

p1 :: s ec 0 at [] with [# -127, d1-1, d2-1, yes-1] at_time 29
```

In this step all inner processes are over. The only object that evolves is z28 (see the set of rules (R11) in the appendix, noting that 28 = $2n+2k+2c+6$).

```
?- evolve(p1,29).

p1 :: s ec 1 at [] with [# -127, d2-1, yes-1] at_time 30
      ...

Used rules in the step 30:
    * The rule 86 has been used only once
The P-system has sent out d1 at step 30
```

In this step the object d1 leaves the system and the skin gets a positive charge.

```
?- evolve(p1,30).

p1 :: s ec 0 at [] with [# -127, no-1] at_time 31
        ...

Used rules in the step 31:
    * The rule 87 has been used only once
    * The rule 88 has been used only once
The P-system has sent out d1 at step 30
The P-system has sent out yes at step 31
```

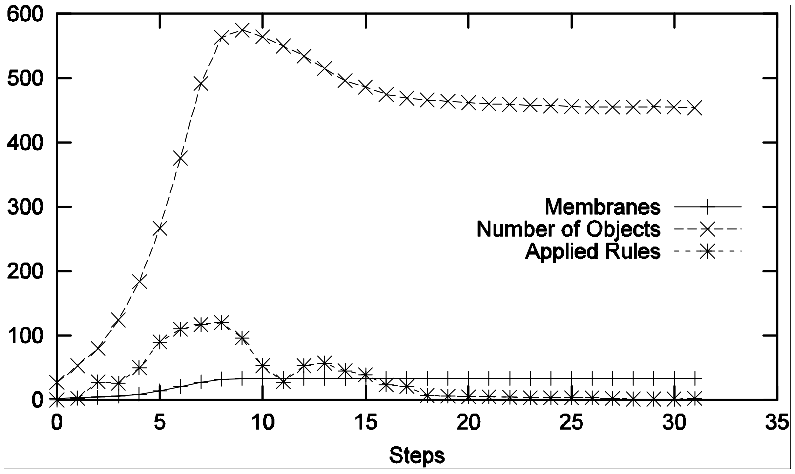
In this step, the object yes is sent out of the system. To check the system we try to evolve one more time, though this is a halting configuration.

```
?- evolve(p1,31).
No more evolution!
The P-system p1 has already reached a halting
configuration at step 31
```

Statistics

Figure 4 shows the evolution along the computation of several indices: the number of membranes, the total number of objects in the P system, and the number of applications of rules that take place. Note that at step nine, the maximum number of membranes is reached, which corresponds to the maxi-

Figure 4. Statistics of the simulation



mum number of objects in the P system. Recall that the generation stage ends at time *2n+1 = 9*. From this point, the number of membranes remains constant because no membrane is divided or dissolved, but the number of objects in the P system begins to decrease. Note also that after objects are renamed (calculation stage), the number of applied rules decreases significantly since in the next stage (checking for w) only two rules are applied in each relevant membrane.

In the last steps of the computation, the total number of objects remains almost unchanged because the checking stages are finished in all membranes but one.

The reader is invited to run simulations for other instances of the problem or to test her or his own approach involving P systems with active membranes.

This simulator is not intended to be an efficient software, but a useful assistant that helps the formal verifications of cellular systems. In our example (*n = 4, c = 4, k = 3*), with an AMD Athlon™ processor at 1.8 GHz and 480 MB RAM, the simulation took 12.23 seconds to perform 8,045,941 Prolog inferences in order to simulate 31 cellular steps.

FINAL REMARKS

The first cellular solutions to **NP**-complete problems using P systems with no input membrane have been described (see Păun, 2001; Zandron, 2001). In such circumstances, specific P systems are built associated with *each* particular instance of the problem to solve. In this chapter, P systems with input are designed, each of them allowing the process of a set of instances of the problem (all those that have the same size, with respect to a predefined criterion).

Although P systems are in general nondeterministic devices, in this chapter we consider recognizer P systems, which provide the same output for all of the computations associated with the same input data. Thus, it is not relevant which branch of the computation is actually chosen.

In this framework, we built solutions to some well-known numerical **NP**-complete decision problems: the subset sum and the knapsack problems. An implementation of these cellular solutions, using our Prolog simulator, is also discussed.

The presented software can be used to deal with other solutions to **NP**-complete problems that use active membranes. It suffices to provide the auxiliary generator tool included in our simulator with the appropriate skeleton of the solution (or, alternatively, to introduce *ex profeso* the system to be simulated).

Although it is useful on many counts, the presented simulator is not efficient from the computational point of view. First, it is designed to run on a single sequential processor. Second, the instances of the problem are codified in 1-ary form, via multisets. It seems rather natural that one of the future improvements of the simulator will be adapting the tool to parallel Prolog.

Nevertheless, the current simulator has proven to be very useful for debugging the process of formal verification of cellular designs. And since it supplies a comprehensive description of P system evolutions, we believe that it is quite suitable for educational purposes. In this direction, we are currently working on developing a user-friendly variant with a graphical interface.

ACKNOWLEDGMENTS

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.

REFERENCES

Adleman, L. M. (1994). Molecular computations of solutions to combinatorial problems. *Science*, 226, 1021-1024.

Bratko, I. (2001). *PROLOG programming for artificial intelligence*. Boston: Addison-Wesley.

Cordón Franco, A., Gutiérrez Naranjo, M. A., Pérez Jiménez, M. J., Riscos Núñez, A., & Sancho Caparrini, F. (2004). Implementing in Prolog an effective cellular solution to the knapsack problem. In Martín-Vide, C., Mauri, G., Păun, Gh., Rozenberg, G., & Salomaa, A. (Eds.), *Membrane computing. International Workshop, WMC 2003*, Tarragona, Spain, revised papers. *Lecture Notes in Computer Science*, 2933, 140-152.

Head, T. J. (1987). Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49, 737-759.

Holland, J. H. (1975). *Adaptation in natural and artificial systems*. Ann Arbor, MI: University of Michigan Press.

McCulloch, W. S. & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 115-133.

Păun, Gh. (1998). Computing with membranes. Turku Center for Computer Science *TUCS Report*, 208. Also in (2000) *Journal of Computer and System Sciences*, *61* (1), 108-143.

Păun, Gh. (2001). P Systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics*, *6* (1), 75-90.

Păun, Gh. (2002). *Membrane computing: An introduction*. Berlin: Springer.

Pérez Jiménez, M. J., Romero Jiménez, A., & Sancho Caparrini, F. (2004). The **P** versus **NP** problem through cellular computing with membranes. *Aspects of Molecular Computing. Lecture Notes in Computer Science*, 2950, 338-352.

Zandron, C. (2001). *A model for molecular computing: Membrane systems*. Ph.D. Thesis, Università degli Studi di Milano.

APPENDIX

In what follows, we show some of the rules generated by the simulator for the instance of the problem considered. Note that the number after ** is the ordinal of the corresponding rule.

```
% Set (R1)
p1 rule e0 inside_of e ec 0 divides_into q inside_of e ec -1
   and e0 inside_of e ec 1 ** 1.
        ...
```

```
p1 rule e4 inside_of e ec 0 divides_into q inside_of e ec -1
   and e4 inside_of e ec 1 ** 5.

p1 rule e0 inside_of e ec 1 divides_into e1 inside_of e ec 0
   and e1 inside_of e ec 1 ** 6.
         ...
p1 rule e3 inside_of e ec 1 divides_into e4 inside_of e ec 0
   and e4 inside_of e ec 1 ** 9.

% Set (R2)
p1 rule x0 evolves_to [a0_] in e ec 0 ** 10.

p1 rule y0 evolves_to [b0_] in e ec 0 ** 11.

p1 rule x0 evolves_to [] in e ec 1 ** 12.

p1 rule y0 evolves_to [] in e ec 1 ** 13.

p1 rule x1 evolves_to [x0] in e ec 1 ** 14.
          ...
p1 rule x4 evolves_to [x3] in e ec 1 ** 17.

p1 rule y1 evolves_to [y0] in e ec 1 ** 18.
         ...
p1 rule y4 evolves_to [y3] in e ec 1 ** 21.

% Set (R3)
p1 rule q evolves_to [q_, q0] in e ec -1 ** 22.

p1 rule b0_ evolves_to [b0g] in e ec -1 ** 23.

p1 rule a0_ evolves_to [a0] in e ec -1 ** 24.

p1 rule b_ evolves_to [bg] in e ec -1 ** 25.

p1 rule a_ evolves_to [a] in e ec -1 ** 26.
```

```
% Set (R4)
p1 rule a0 inside_of e ec -1 sends_out # of e ec 0 ** 27.

p1 rule a inside_of e ec 0 sends_out # of e ec -1 ** 28.

% Set (R5)
p1 rule q0 evolves_to [q1] in e ec -1 ** 29.

p1 rule q2 evolves_to [q3] in e ec -1 ** 30.
          ...
p1 rule q1 evolves_to [q2] in e ec 0 ** 33.

p1 rule q3 evolves_to [q4] in e ec 0 ** 34.

p1 rule q5 evolves_to [q6] in e ec 0 ** 35.


% Set (R6)
p1 rule q1 inside_of e ec -1 sends_out # of e ec 1 ** 36.

p1 rule q3 inside_of e ec -1 sends_out # of e ec 1 ** 37.

p1 rule q5 inside_of e ec -1 sends_out # of e ec 1 ** 38.

p1 rule q7 inside_of e ec -1 sends_out # of e ec 1 ** 39.

% Set (R7)
p1 rule q_ evolves_to [q_0] in e ec 1 ** 40.

p1 rule b0g evolves_to [b0] in e ec 1 ** 41.

p1 rule bg evolves_to [b] in e ec 1 ** 42.

p1 rule a evolves_to [] in e ec 1 ** 43.

% Set (R8)
p1 rule b0 inside_of e ec 1 sends_out # of e ec 0 ** 44.
```

```
p1 rule b inside_of e ec 0 sends_out # of e ec 1 ** 45.

% Set (R9)
p1 rule q_0 evolves_to [q_1] in e ec 1 ** 46.

p1 rule q_2 evolves_to [q_3] in e ec 1 ** 47.

p1 rule q_4 evolves_to [q_5] in e ec 1 ** 48.

p1 rule q_6 evolves_to [q_7] in e ec 1 ** 49.

p1 rule q_8 evolves_to [q_9] in e ec 1 ** 50.

p1 rule q_1 evolves_to [q_2] in e ec 0 ** 51.

p1 rule q_3 evolves_to [q_4] in e ec 0 ** 52.

p1 rule q_5 evolves_to [q_6] in e ec 0 ** 53.

p1 rule q_7 evolves_to [q_8] in e ec 0 ** 54.

% Set (R10)
p1 rule q_9 inside_of e ec 1 sends_out yes of e ec 0 ** 55.

p1 rule q_9 inside_of e ec 0 sends_out yes of e ec 0 ** 56.

% Set (R11)
p1 rule z0 evolves_to [z1] in s ec 0 ** 57.

p1 rule z1 evolves_to [z2] in s ec 0 ** 58.
          ...
p1 rule z26 evolves_to [z27] in s ec 0 ** 83.

p1 rule z27 evolves_to [z28] in s ec 0 ** 84.

p1 rule z28 evolves_to [d1, d2] in s ec 0 ** 85.

% Set (R12)
p1 rule d1 inside_of s ec 0 sends_out d1 of s ec 1 ** 86.
```

```
p1 rule d2 evolves_to [no] in s ec 1 ** 87.

p1 rule yes inside_of s ec 1 sends_out yes of s ec 0 ** 88.

p1 rule no inside_of s ec 1 sends_out no of s ec 0 ** 89.
```