Membrane Computing: A General View

Oscar H. Ibarra

Department of Computer Science, University of California
Santa Barbara, CA 93106, USA

Gheorghe Păun

Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania

Abstract

We give an informal presentation of the basic ideas, results, and applications of membrane computing, a branch of natural computing inspired by the structure and the functioning of biological cells, cell tissues, or colonies of cells. Membrane computing has given rise to computing models (called P systems) that are distributed and parallel, which process multisets of objects in compartments defined by membranes.

After introducing the main classes of P systems, illustrated with some simple examples, we recall some results, especially those that concern their computing power and computing efficiency: the equivalence with Turing machines for many classes of P systems, and the possibility of designing devices which are capable of solving computationally intractable problems in feasible time. We then briefly discuss some applications (to biology, bio-medicine, economics, etc.), giving a typical example to illustrate this research direction. Finally, we report on some software simulators and hardware implementations of P systems that have been developed.

Keywords

Natural computing; biologically-motivated models; membrane computing; P system; symport/antiport; spiking neural P system

1. Introduction

Membrane computing is a branch of natural computing, the broad area of research concerned with computation taking place in nature and with human-designed computing inspired by nature. Membrane computing abstracts computing models from the architecture and the functioning of living cells, as well as from the organization of cells in tissues, organs (brain included) or other higher order structures such as colonies of cells (e.g., bacteria). The initial goal was to learn from cell biology something possibly useful to computer science, and the area quickly developed in this direction. Several classes of computing models were defined in this context, inspired from biological facts or motivated from mathematical or computer science points of view. A number of applications were reported in the last few years in several areas – biology, bio-medicine, linguistics, computer graphics, economics, approximate optimization, cryptography, etc.

The models investigated in membrane computing area are called *P systems*. The main ingredients of a P system are (i) the membrane structure, (ii) the multisets of objects placed in the compartments of the membrane structure, and (iii) the rules for processing the objects and the membranes. Thus, membrane computing can be defined as a framework for devising cell-like or tissue-like computing models which process multisets in compartments defined

Ann. - Eur. Acad. Sci. (2006-2007) Liège: EAS Publishing House, 83-101.

by means of membranes. These models are (in general) distributed and parallel. When a P system is considered as a computing device, hence it is investigated in terms of (theoretical) computer science, the main issues studied concern the *computing power* (in comparison with standard models from computability theory, especially Turing machines and their restrictions) and the *computing efficiency* (the possibility of using parallelism for solving computationally hard problems in a feasible time). Computationally and mathematically oriented ways of using the rules and of defining the result of a computation are considered in this case (e.g., maximal or minimal parallelism, halting, counting objects). When a P system is constructed as a model of a bio-chemical process, it is examined in terms of dynamical systems, with the evolution in time being the issue of interest, not a specific output.

From a theoretical point of view, P systems are both powerful (most classes are Turing complete, even when using ingredients of a reduced complexity – a small number of membranes, rules of simple forms, ways of controlling the use of rules directly inspired from biology) and efficient (many classes of P systems, especially those with enhanced parallelism, can solve computationally hard problems – typically **NP**-complete problems, but also harder problems, e.g., **PSPACE**-complete problems – in feasible time – typically polynomial). Then, as a modeling framework, membrane computing is rather adequate for handling discrete (biological) processes, having many attractive features: easy understandability, scalability and programmability, inherent compartmentalization and non-linearity, etc. Ideas from cell biology as captured by membrane computing proved to be rather useful in handling various computer science topics – one typical example is that of membrane evolutionary algorithms, used for solving optimization problems.

Membrane computing was initiated in 1998 (with the seminal paper published in 2000, [34]) and the literature of this area has grown very fast (already in 2003, Thompson Institute for Scientific Information, ISI, has qualified the initial paper as "fast breaking" and the domain as "emergent research front in computer science" – see `http://esi-topics.com`). For a comprehensive information about membrane computing (full bibliography, many downloadable papers, including all pre-proceedings volumes of the annual meetings on membrane computing, PhD theses, research reports) the reader is advised to visit the web site from [46] maintained in Milano, Italy, under the auspices of European Molecular Computing Consortium (EMCC). Details can be also found in the monograph [36], with several applications presented in [9], where a friendly introduction to membrane computing is given in the first chapter.

As mentioned earlier, membrane computing started by looking to the cell in order to learn something that could be useful to computer science, but the research also considered cell organization in tissues (in general, populations of cells, such as colonies of bacteria), and, recently, also organization of neurons in the brain. Thus, at this moment, there are three main types of P systems: (i) cell-like P systems, (ii) tissue-like P systems, and (iii) neural-like P systems.

The first type imitates the (eukaryotic) cell, and its basic ingredient is the *membrane structure*, a hierarchical arrangement of membranes (understood as three dimensional vesicles), i.e., delimiting compartments where multisets of objects are placed; the objects are in general described by symbols from a given alphabet, but also string-objects can be considered; rules for evolving these objects are provided, also localized, acting in specified compartments or on specified membranes. The objects not only evolve, but they also pass through membranes (we say that they are "communicated" among compartments). The rules can have several forms, and their use can be controlled in various ways: promoters, inhibitors, priorities, etc. Also the hierarchy of membranes can evolve, e.g., by creating and destroying membranes, by division, by bio-like operations of exocytosis, endocytosis, phagocytosis, and so on.

In tissue-like P systems, several one-membrane cells are considered as evolving in a common environment. They contain multisets of objects, while also the environment contains objects.

Certain cells can communicate directly (channels are provided between them) and all cells can communicate through the environment. The channels can be given in advance or they can be dynamically established – this latter case appears in so-called *population P systems*. In the case when the cells are simple, of a limited capacity (as the number of objects they contain or of rules they can use), we obtain the notion of *P colony*.

Finally, there are two types of neural-like P systems. One is similar to tissue-like P system in that the cells (neurons) are placed in the nodes of an arbitrary graph and they contain multisets of objects, but they also have a *state* which controls the evolution. Another promising variant was recently introduced, under the name of *spiking neural P systems*, where one uses only one type of objects, the *spike*, and the main information one works with is the distance between consecutive spikes.

The cell-like P systems were introduced first and their theory is now very well developed; tissue-like P systems have also attracted considerable interest, while the neural-like systems, mainly under the form of spiking neural P systems, were only recently investigated.

In what follows, we will mainly discuss cell-like P systems and spiking neural P systems, and we refer the reader to the bibliography from [46] for the other classes.

2. Cell-Like P Systems

In this section we only consider cell-like P systems. For convenience, we will simply refer to them as P systems.

Briefly, such a system consists of a hierarchical arrangement of *membranes*, which delimit *compartments*, where *multisets* (sets with multiplicities associated with their elements) of abstract *objects* are placed. These objects correspond to the chemicals in the compartments of a cell; the chemicals swim in water (many of them are bound on membranes, but we do not consider this case here), and their multiplicity matters – that is why the data structure most adequate to this situation is the multiset (a multiset can be seen as a string modulo permutation; hence, in membrane computing one usually represents the multisets by strings). In what follows the objects are supposed to be unstructured, hence we represent them by symbols from a given alphabet. There also are classes of P systems dealing with string objects.

The objects evolve according to *rules* which are also associated with the regions. The rules dictate how the objects are changed and how they can be moved (*communicated*) across membranes. There also are rules which only move objects across membranes, as well as rules for evolving the membranes themselves (e.g., by destroying, creating, dividing, merging membranes). By using these rules, we can change the *configuration* of a system (the multisets in its compartments as well as the membrane structure); we say that we get a *transition* among system configurations.

The rules can be applied in many ways. The basic mode imitates the biological way chemical reactions are performed – in parallel, with the (mathematical) additional restriction to have *maximal parallelism*: one applies a bunch of rules which is maximal, no further object can evolve at the same time by any rule. Besides this mode, several others have been considered: sequential (one rule is used in each step), bounded parallelism (the number of membranes to evolve and/or the number of rules to be used in any step is bounded in advance), minimal parallelism (in each compartment where a rule *can* be used, at least one rule *must be* used [8]). In all cases, a common feature is that the objects to evolve and the rules by which they evolve are chosen in a *non-deterministic* manner. A sequence of transitions forms a *computation* and with computations which *halt* (reach a configuration where no rule is applicable) we associate a *result*, for instance, in the form of the multiset of objects present in a specified membrane when the system halts.

This way of using a P system, starting from an initial configuration and computing a number,

is a grammar-like (generative) one. We can also work in an automaton-style: an input is introduced in the system, for instance, in the form of a number represented by the multiplicity of an object placed in a specified membrane, and we start computing; the input number is accepted if and only if the computation halts. The respective devices were called P automata (see [10] for a survey of results). A combination of the two modes leads to a functional behavior: an input is introduced in the system (at the beginning, or symbol by symbol during the computation) and also an output is produced. In particular, we can have a decidability case, where the input encodes an instance of a decision problem and the output is one of two special objects representing the answers `yes` and `no` to the problem.

The generality of this approach is obvious. We start from the cell, but the abstract model deals with very general notions: membranes interpreted as separators of regions with filtering capabilities, objects and rules assigned to regions; the basic data structure is the multiset. Thus, membrane computing can be interpreted as a *bio-inspired framework for distributed parallel processing of multisets.*

As briefly introduced above, the P systems are synchronous systems, and this feature is useful for theoretical investigations (e.g., for obtaining universality results or results related to the computational complexity of P systems). Also non-synchronized systems were considered, asynchronous in the standard sense or even time-free, or clock-free (e.g., generating the same output, irrespective of the duration associated with the evolution rules). Similarly, in applications to biology, specific strategies of evolution are considered. We do not enter here into details, rather we refer the reader to the bibliography given below.

Let us now go into some more specific details – still remaining at an informal level. As mentioned above, we look to the cell structure and functioning, trying to get suggestions for an abstract computing model. The fundamental feature of a cell is its compartmentalization through membranes. Accordingly, the main ingredient of a P system is the *membrane structure*, a hierarchical arrangement of membranes, which delimit compartments. Fig. 1 illustrates this notion and the related terminology.

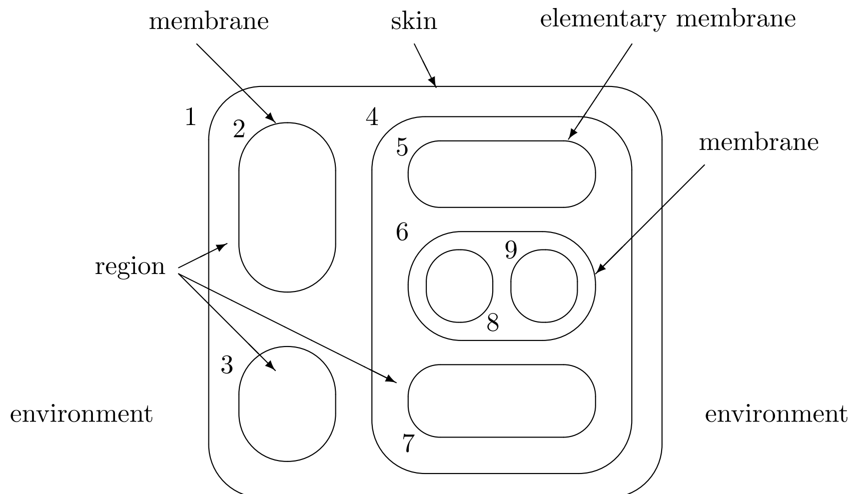

Fig. 1. A membrane structure

We distinguish the external membrane (corresponding to the plasma membrane and usually called the *skin* membrane) and several internal membranes; a membrane without any other membrane inside it is said to be *elementary*. Each membrane determines a compartment, also called *region*, the space delimited from above by it and from below by the membranes placed

directly inside, if any exists. The correspondence membrane–region is one-to-one, so that we identify by the same label a membrane and its associated region.

In the basic class of P systems, each region contains a multiset of symbol-objects, described by symbols from a given alphabet, but also objects described by strings, or even by more complex structures were considered.

The objects evolve by means of evolution rules, which are also localized, associated with the regions of the membrane structure. The typical form of such a rule is $cd \rightarrow (a, here)(b, out)(b, in)$, with the following meaning: one copy of object c and one copy of object d react and the reaction produces one copy of a and two copies of b; the newly produced copy of a remains in the same region (indication *here*), one of the copies of b exits the compartment, going to the surrounding region (indication *out*) and the other enters one of the directly inner membranes (indication *in*). We say that the objects a, b, b are *communicated* as indicated by the commands associated with them in the right hand member of the rule. When an object exits the skin membrane, it is "lost" in the environment, it never comes back into the system. If no inner membrane exists (that is, the rule is associated with an elementary membrane), then the indication *in* cannot be followed, and the rule cannot be applied.

A membrane structure and the multisets of objects from its compartments identify a *configuration* of a P system. By a non-deterministic maximally parallel use of rules as suggested above we pass to another configuration; such a step is called a *transition.* A sequence of transitions constitutes a *computation.* A computation is successful if it halts – it reaches a configuration where no rule can be applied to the existing objects. With a halting computation we can associate a *result* in various ways. The simplest possibility is to count the objects present in the halting configuration in a specified elementary membrane; this is called *internal output.* We can also count the objects which leave the system during the computation, and this is called *external output.* In both cases the result is a number. If we distinguish among different objects, then we can have as the result a vector of natural numbers. The objects which leave the system can also be arranged in a sequence according to the moments when they exit the skin membrane, and in this case the result is a string.

Because of the non-determinism of the application of rules, starting from an initial configuration, we can get several successful computations, hence several results. Thus, a P system *computes* (or *generates*) a set of numbers, or a set of vectors of numbers, or a language.

Many classes of P systems can be obtained by considering various possibilities for the various ingredients. We enumerate here several of these possibilities, without exhausting the list:

- Objects: symbols, strings of symbols, spikes, arrays, trees, numerical variables, other data structures, combinations of the above.
- Data structure: multisets, sets (languages in the case of strings), fuzzy sets or fuzzy multisets.
- Place of objects: in compartments, on membranes (corresponding to proteins embedded in membranes, as considered, e.g., in [5]), combined.
- Forms of rules: multiset rewriting, symport/antiport (as in biology, see [31]), communication rules, boundary rules (involving multisets of objects from the two sides of a membrane, see [3]), with active membranes (involving membranes in the rules, hence leading to a dynamical membrane structure, see [35]), combined, string rewriting, array/trees processing, spike processing.
- Controls on rules: catalysts, priority, promoters, inhibitors, activators, sequencing, energy, probabilities.

- Form of membrane structure: cell-like (tree), tissue-like (arbitrary graph).
- Type of membrane structure: static, dynamical, pre-computed (arbitrarily large).
- Timing: synchronized, non-synchronized, local synchronization, time-free.
- Ways of using the rules: maximal parallelism, minimal parallelism, bounded parallelism, sequential.
- Successful computations: global halting, local halting (at least one compartment stops working), with specified events signaling the end of a computation, non-halting.
- Modes of using a system: generative, accepting, computing an input-output function, deciding.
- Ways to define the output: internal, external, traces (the paths of a distinguished object across membranes is taken as the result of the computation), tree of membrane structure (in the case of P systems with a dynamical membrane structure, the membrane structure itself – hence the tree describing it – can be considered as the processed data), spike train.
- Types of results: set of numbers, set of vectors of numbers, languages, set of arrays, sets of trees, `yes/no`.

We refer to the literature for details and we only add here the fact that when using P systems as models of biological systems/processes, we have to apply the rules in ways suggested by biochemistry, according to reaction rates or probabilities; in many cases, these rates are computed dynamically, depending on the current population of objects in the system.

In general, a (cell-like) P system is formalized as a construct $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$, where O is the alphabet of objects, μ is the membrane structure (with m membranes), $w_1, \ldots, w_m$ are multisets of objects present in the m regions of μ at the beginning of a computation, $R_1, \ldots, R_m$ are finite sets of evolution rules, associated with the regions of μ, and i_o is the label of a membrane, used as the output membrane.

We end this section with a simple example, illustrating the architecture and the functioning of a (cell-like) P system. Fig. 2 indicates the initial configuration (the rules included) of a system which computes a function, namely $n \longrightarrow n^2$, for any natural number $n \geq 1$. Besides catalytic and non-cooperating rules, the system also contains a rule with promoters, $b_2 \to b_2(e, in)|_{b_1}$: the object b_2 evolves to $b_2 e$ only if at least one copy of object b_1 is present in the same region.

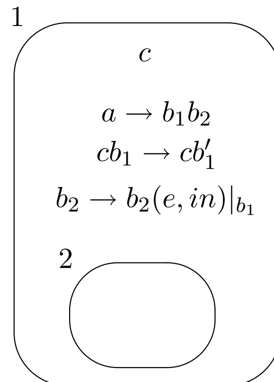

Fig. 2. A P system with catalysts and promoters

In symbols, the system is given as follows:

$$\begin{aligned}
\Pi &= (O, C, \mu, w_1, w_2, R_1, R_2, i_o), \text{ where:}\\
O &= \{a, b_1, b_1', b_2, c, e\} \text{ (the set of objects)}\\
C &= \{c\} \text{ (the set of catalysts)}\\
\mu &= [_1\ [_2\ ]_2\ ]_1 \text{ (membrane structure)}\\
w_1 &= c \text{ (initial objects in region 1)}\\
w_2 &= \lambda \text{ (initial objects in region 2)}\\
R_1 &= \{a \to b_1 b_2,\ cb_1 \to cb_1',\ b_2 \to b_2 e_{in}|_{b_1}\} \text{ (rules in region 1)}\\
R_2 &= \emptyset \text{ (rules in region 2)}\\
i_o &= 2 \text{ (the output region).}
\end{aligned}$$

We start with only one object in the system, the catalyst c. If we want to compute the square of a number n, then we have to input n copies of the object a in the skin region of the system. In that moment, the system starts working, by using the rule $a \to b_1 b_2$, which has to be applied in parallel to all copies of a; hence, in one step, all objects a are replaced by n copies of b_1 and n copies of b_2. From now on, the other two rules from region 1 can be used. The catalytic rule $cb_1 \to cb_1'$ can be used only once in each step, because the catalyst is present in only one copy. This means that in each step one copy of b_1 gets primed. Simultaneously (because of the maximal parallelism), the rule $b_2 \to b_2(e, in)|_{b_1}$ should be applied as many times as possible and this means n times, because we have n copies of b_2. Note the important difference between the promoter b_1, which allows using the rule $b_2 \to b_2(e, in)|_{b_1}$, and the catalyst c: the catalyst is involved in the rule, it is counted when applying the rule, while the promoter makes possible the use of the rule, but it is not counted; the same (copy of an) object can promote any number of rules. Moreover, the promoter can evolve at the same time by means of another rule (the catalyst is never changed).

In this way, in each step we change one b_1 to b_1' and we produce n copies of e (one for each copy of b_2); the copies of e are sent to membrane 2 (the indication in from the rule $b_2 \to b_2(e, in)|_{b_1}$). The computation should continue as long as there are applicable rules. This means exactly n steps: in n steps, the rule $cb_1 \to cb_1'$ will exhaust the objects b_1 and in this way neither this rule can be applied, nor $b_2 \to b_2(e, in)|_{b_1}$, because its promoter does no longer exist. Consequently, in membrane 2, considered as the output membrane, we get n^2 copies of object e.

Note that the computation is deterministic, always the next configuration of the system is unique, and that, changing the rule $b_2 \to b_2(e, in)|_{b_1}$ with $b_2 \to b_2(e, out)|_{b_1}$, the n^2 copies of e will be sent to the environment, hence we can read the result of the computation outside the system, and in this case membrane 2 is useless.

3. Cell-Like P Systems with Symport and Antiport Rules

P systems as above are sometimes called transition P systems, and their rules correspond to reactions taking place in the cell, *inside* the compartments. However, an important part of the cell activity is related to the passage of substances through membranes, and one of the most interesting ways to handle this trans-membrane communication is by coupling molecules as suggested by cell biology. The process by which two molecules pass together across a membrane (through a specific protein channel) is called *symport*; when the two molecules pass simultaneously through a protein channel, but in opposite directions, the process is called *antiport*.

We can formalize these operations, in a general form, by considering symport rules of the form (x, in) and (x, out), and antiport rules of the form $(z, out; w, in)$, where x, z, and w are

multisets of arbitrary size; one says that the length of x, denoted $|x|$, is the *weight* of the symport rule, and $\max(|z|, |w|)$ is the *weight* of the antiport rule. Thus, we obtain the following important class of P systems.

A *P system with symport/antiport rules* is a construct of the form

$$\Pi = (O, \mu, w_1, \dots, w_m, E, R_1, \dots, R_m, i_o),$$

where:

1. O is the alphabet of objects,
2. μ is the membrane structure (of degree $m \geq 1$, with the membranes labeled $1, 2, \dots, m$ in a one-to-one manner),
3. $w_1, \dots, w_m$ are strings over O representing the multisets of objects present in the m compartments of μ in the initial configuration of the system,
4. $E \subseteq O$ is the set of objects supposed to appear in the environment in arbitrarily many copies,
5. $R_1, \dots, R_m$ are the (finite) sets of rules associated with the m membranes of μ,
6. $i_o \in H$ is the label of a membrane of μ, which indicates the *output* region of the system.

The rules from R can be of two types, symport rules and antiport rules, of the forms specified above.

The rules are used in the non-deterministic maximally parallel manner. We define transitions, computations, and halting computations in the usual way. The number of objects present in region i_o in the halting configuration is said to be computed by the system by means of that computation; the set of all numbers computed in this way by Π is denoted by $N(\Pi)$.

Note that the objects from set E are supposed to be present in the environment in an arbitrary number of copies, and that that this time the rules are associated with membranes, and not with regions.

We illustrate the definition of P systems with symport/antiport with an example of a more general interest: we give a general construction for simulating a *register machine*. In this way, we also introduce one of the widely used proof techniques for the universality results in this area.

Informally speaking, a register machine consists of a specified number of counters (also called registers) which can hold any natural number, and which are handled according to a program consisting of labeled instructions; the counters can be increased or decreased by 1 – the decreasing possible only if a counter holds a number greater than or equal to 1 (we say that it is non-empty) – and checked whether they are non-empty.

Formally, a (non-deterministic) *register machine* is a device $M = (m, B, l_0, l_h, R)$, where $m \geq 1$ is the number of counters, B is the (finite) set of instruction labels, l_0 is the initial label, l_h is the halting label, and R is the finite set of instructions labeled (hence uniquely identified) by elements from B. The labeled instructions are of the following forms:

– $l_1 : (\texttt{add}(r), l_2, l_3)$, $1 \leq r \leq m$ (add 1 to counter r and go non-deterministically to one of the instructions with labels l_2, l_3),

– $l_1 : (\texttt{sub}(r), l_2, l_3)$, $1 \leq r \leq m$ (if counter r is not empty, then subtract 1 from it and go to the instruction with label l_2, otherwise go to the instruction with label l_3).

A counter machine generates a set of natural numbers in the following manner: we start computing with all m counters empty, with the instruction labeled l_0; if the label l_h is reached, then the computation *halts* and the value of counter 1 is the number generated by the computation. The set of all natural numbers generated in this way by M is denoted by $N(M)$. It is known (see [29]) that non-deterministic counter machines (with three counters) can compute any set of Turing computable sets of natural numbers.

Now, a register machine can be easily simulated by a P system with symport/antiport rules. The idea is illustrated in Fig. 3, where we have represented the initial configuration of the system, the rules associated with the unique membrane, and the set E of objects present in the environment.

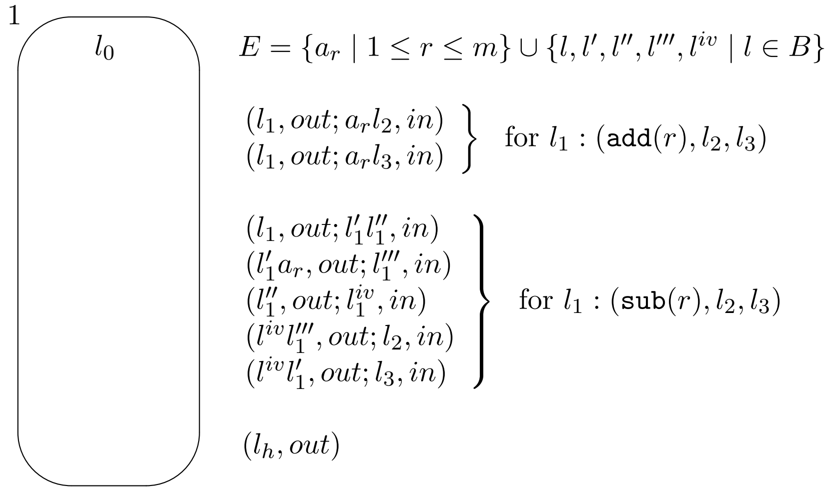

Fig. 3. An example of symport/antiport P system

The value of each register r is represented by the multiplicity of object $a_r, 1 \leq r \leq m$, in the unique membrane of the system. The labels from B, as well as their primed versions, are also objects of our system. We start with the unique object l_0 present in the system. In the presence of a label object l_1 we can simulate the corresponding instruction $l_1 : (\mathtt{add}(r), l_2, l_3)$ or $l_1 : (\mathtt{sub}(r), l_2, l_3)$.

The simulation of an `add` instruction is clear, so we discuss only a `sub` instruction. The object l_1 exits the system in exchange of the two objects $l_1' l_1''$ (rule $(l_1, out; l_1' l_1'', in)$). In the next step, if any copy of a_r is present in the system, then l_1' has to exit (rule $(l_1' a_r, out; l_1''', in)$), thus diminishing the number of copies of a_r by one, and bringing inside the object l_1'''; if no copy of a_r is present, which corresponds to the case when the register r is empty, then the object l_1' remains inside. Simultaneously, rule $(l_1'', out; l_1^{iv}, in)$ is used, bringing inside the "checker" l_1^{iv}. Depending on what this object finds in the system, either l_1''' or l_1', it introduces the label l_2 or l_3, respectively, which corresponds to the correct continuation of the computation of the register machine.

When the object l_h is introduced, it is expelled into the environment and the computation stops.

Clearly, the (halting) computations in Π directly correspond to (halting) computations in M; hence $N(M) = N(\Pi)$.

4. Spiking Neural P Systems

Spiking neural P systems (SN P systems) were introduced in [21] with the aim of defining P systems based on ideas specific to spiking neurons, recently much investigated in neural computing.

Very shortly, an SN P system consists of a set of *neurons* (cells, consisting of only one membrane) placed in the nodes of a directed graph and sending signals (*spikes*, denoted in what follows by the symbol a) along *synapses* (arcs of the graph). Thus, the architecture is that of a tissue-like P system, with only one kind of objects present in the cells. The objects evolve by means of *spiking rules*, which are of the form $E/a^c \to a; d$, where E is a regular expression over $\{a\}$ and c, d are natural numbers, $c \geq 1, d \geq 0$. The meaning is that a neuron containing k spikes such that $a^k \in L(E), k \geq c$, can consume c spikes and produce one spike, after a delay of d steps. This spike is sent to all neurons to which a synapse exists outgoing from the neuron where the rule was applied. There also are *forgetting rules*, of the form $a^s \to \lambda$, with the meaning that $s \geq 1$ spikes are forgotten, provided that the neuron contains exactly s spikes. We say that the rules "cover" the neuron, all spikes are taken into consideration when using a rule. The system works in a synchronized manner, i.e., in each time unit, each neuron which can use a rule should do it, but the work of the system is sequential in each neuron: only (at most) one rule is used in each neuron. One of the neurons is considered to be the *output neuron*, and its spikes are also sent to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1, the other moments are marked with 0. This binary sequence is called the *spike train* of the system – it might be infinite if the computation does not stop.

In the spirit of spiking neurons, the result of a computation is encoded in the distance between consecutive spikes sent into the environment by the (output neuron of the) system. For example, we can consider only the distance between the first two spikes of a spike train, or the distances between the first k spikes, the distances between all consecutive spikes, taking into account all intervals or only intervals that alternate, all computations or only halting computations, etc.

An SN P system can also be used in the accepting mode: a neuron is designated as the *input neuron* and two spikes are introduced in it, at an interval of n steps; the number n is accepted if the computation halts.

Another possibility is to consider the spike train itself as the result of a computation, and then we obtain a (binary) language generating device. We can also consider both input and output neurons and then an SN P system can work as a transducer. Languages on arbitrary alphabets can be obtained by generalizing the form of rules: take rules of the form $E/a^c \to a^p; d$, with the meaning that, provided that the neuron is covered by E, c spikes are consumed and p spikes are produced, and sent to all connected neurons after d steps (such rules are called *extended*). Then, with a step when the system sends out i spikes, we associate a symbol b_i, and thus we get a language over an alphabet with as many symbols as the number of spikes simultaneously produced. Another natural extension is to consider several output neurons, thus producing vectors of numbers, not only single numbers.

Also for SN P systems we skip the technical details, but we consider a simple example. We give it first in a formal manner (if a rule $E/a^c \to a; d$ has $L(E) = \{a^c\}$, then we write it in the simplified form $a^c \to a; d$):

$$
\begin{aligned}
\Pi_1 &= (O, \sigma_1, \sigma_2, \sigma_s, syn, out), \text{ with} \\
O &= \{a\} \text{ (alphabet, with only one object, the spike)} \\
\sigma_1 &= (2, \{a^2/a \to a; 0, \quad a \to \lambda\}) \\
&\qquad \text{(first neuron: initial number of spikes, rules)} \\
\sigma_2 &= (1, \{a \to a; 0, \quad a \to a; 1\}) \\
&\qquad \text{(second neuron: initial number of spikes, rules)} \\
\sigma_3 &= (3, \{a^3 \to a; 0, \quad a \to a; 1, \quad a^2 \to \lambda\}) \\
&\qquad \text{(third neuron: initial number of spikes, rules)} \\
syn &= \{(1,2),\ (2,1),\ (1,3),\ (2,3)\} \text{ (synapses)} \\
out &= 3 \text{ (output neuron).}
\end{aligned}
$$

This system is represented in a graphical form in Fig. 4 and it functions as follows. All neurons can fire in the first step, with neuron σ_2 choosing non-deterministically between its two rules. Note that neuron σ_1 can fire only if it contains two spikes; one spike is consumed, the other remains available for the next step.

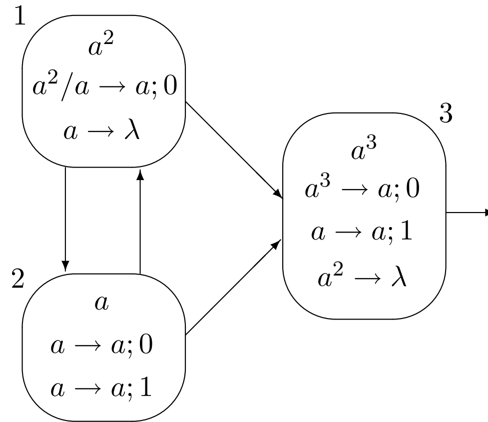

Fig. 4. An SN P system generating all natural numbers greater than 1

Both neurons σ_1 and σ_2 send a spike to the output neuron, σ_3; these two spikes are forgotten in the next step. Neurons σ_1 and σ_2 also exchange their spikes; thus, as long as neuron σ_2 uses the rule $a \to a; 0$, the first neuron receives one spike, thus completing the needed two spikes for firing again.

However, at any moment, starting with the first step of the computation, neuron σ_2 can choose to use the rule $a \to a; 1$. On the one hand, this means that the spike of neuron σ_1 cannot enter neuron σ_2, it only goes to neuron σ_3; in this way, neuron σ_2 will never work again because it remains empty. On the other hand, in the next step neuron σ_1 has to use its forgetting rule $a \to \lambda$, while neuron σ_3 fires, using the rule $a \to a; 1$. Simultaneously, neuron σ_2 emits its spike, but it cannot enter neuron σ_3 (it is closed this moment); the spike enters neuron σ_1, but it is forgotten in the next step. In this way, no spike remains in the system. The computation ends with the expelling of the spike from neuron σ_3. Because of the waiting moment imposed by the rule $a \to a; 1$ from neuron σ_3, the two spikes of this neuron cannot be consecutive, but at least two steps must exist in between.

Thus, we conclude that Π computes/generates all natural numbers greater than or equal to 2.

5. Computing Power

As we have mentioned before, many classes of P systems, which combine various ingredients (as described above or similar ones), are able of simulating Turing machines, hence they are *computationally complete*. Always, the proofs of results of this type are constructive, and this has an important consequence from the computability point of view: there are *universal* (hence *programmable*) P systems. In short, starting from a universal Turing machine (or an equivalent universal device), we get an equivalent universal P system. Among others, this implies that in the case of Turing complete classes of P systems, the hierarchy on the number of membranes always collapses (at most at the level of the universal P systems). Actually, the number of membranes in a P system sufficient to characterize the power of Turing machines is always rather small.

We only mention here three of the most interesting (types of) universality results for cell-like P systems:

1. P systems with symbol-objects with catalytic rules, using only two catalysts and two membranes, are universal.

2. P systems with symport/antiport rules of a restricted size (example: three membranes, symport rules of weight 2, and no antiport rules, or three membranes and minimal symport and antiport rules) are universal.

3. P systems with symport/antiport rules (of arbitrary size), using only three membranes and only three objects, are universal.

There are several other similar results, improvements or extensions of them. Many results are also known for tissue-like P systems. Details can be found, e.g., in the proceedings of the yearly Workshops of Membrane Computing mentioned in the bibliography of this chapter. For instance, universality results were obtained also in the case of P systems working in the accepting mode, and an interesting problem appears in this case, because we can then use deterministic systems. Most universality results were obtained in the deterministic case, but there also are situations where the deterministic systems are strictly less powerful than the non-deterministic ones. This is proven in [20], for the accepting catalytic P systems. The question of whether non-deterministic systems are more powerful than deterministic systems for non-universal P systems was studied in [17, 18], where two classes of P system acceptors with only communicating rules or symport/antiport rules were introduced. For the first class, the deterministic and non-deterministic versions are equivalent if and only if deterministic and non-deterministic linear bounded automata are equivalent. The latter problem is a long-standing open question in complexity theory. For the second class, the deterministic version is strictly weaker than the non-deterministic version. Both classes are non-universal, but can accept fairly complex languages.

The hierarchy on the number of membranes collapses in many cases also for non-universal classes of P systems, but there also are cases when "the number of membrane matters", to cite the title of [16], where two classes of P systems were defined for which the hierarchies on the number of membranes are infinite.

Also various classes of SN P systems are computationally complete, as devices which generate or accept sets of numbers. This is true when no bound is imposed on the number of spikes present in any neuron; if such a bound exists, then the sets of numbers generated (or accepted) are semilinear. In [6, 19], several classes of SN P systems were characterized in terms of their computing power and complexity. In particular, asynchronous and sequential SN P systems were analyzed and some conditions were presented under which they become (non-)universal.

The non-universal variants are characterized by monotonic counter machines and partially blind counter machines and, hence, have many decidable properties.

It is worth noting that the proofs of computational completeness are based on simulating various types of grammars with restricted derivation (mainly matrix grammars with appearance checking) or on simulating register machines. In the case of SN P systems, this has an interesting consequence: starting the proofs from small universal register machines, one can find small universal SN P systems. For instance, as shown in [32], there are universal computing SN P systems with 84 neurons using standard rules and with only 49 neurons using extended rules. In the generative case, the best results are 79 and 50 neurons, respectively. Of course, these results are probably not optimal, hence it is a research topic to improve them.

6. Computational Efficiency

The computational power (the "competence") is only one of the important questions to be dealt with when defining a new (bio-inspired) computing model. The other fundamental question concerns the computing *efficiency*. Because P systems are parallel computing devices, it is expected that they can solve hard problems in an efficient manner – and this expectation is confirmed for systems provided with ways for producing an exponential workspace in a linear time. Three main such biologically inspired possibilities have been considered so far in the literature, and *all of them were proven to lead to polynomial solutions to* **NP***-complete problems*, by a time-space trade-off.

These three ideas are *membrane division*, *membrane creation*, and *string replication*. The standard problems addressed in this framework were decidability problems, starting with `SAT`, the Hamiltonian Path problem, the Node Covering problem, but also other types of problems were considered, such as the problem of inverting one-way functions, or the Subset-sum and the Knapsack problems (note that the last two are numerical problems, where the answer is not of the yes/no type, as in decidability problems). In general, the approach is of a brute force type: all candidate solutions are generated, making use of the massive parallelism (e.g., all truth assignments of the variables appearing in a `SAT` problem), then they are checked, again in parallel, in search of a solution.

A complete example of solving a computationally hard problem in this framework is too complex to be recalled here, but we illustrate the idea of using membrane division for producing all candidate solutions to a problem, by presenting a module of a P system which produces all truth assignments for n variables $x_1, \dots, x_n$ in a time linear with respect to n. We start from the following configuration

$$[_0[_1 a_1 a_2 \dots a_n c_0]_1]_0,$$

where objects a_i associated with Boolean variables $x_i, 1 \leq i \leq n$, are placed in the inner membrane 1. We have also considered the counter object c_0, which will change its subscript during the computation; such objects are much used for synchronization purposes, for controlling the passage to various phases of the computation, etc. The following rules are used:

$$[_1 a_i]_1 \to [_1 t_i]_1 [_1 f_i]_1, \ 1 \leq i \leq n,$$
$$[_1 c_j \to c_{j+1}]_1, \ 0 \leq j \leq n-1.$$

The first rule divides the membrane with label 1, replacing the object a_i with t_i (representing $x_i = true$) in the first copy of the membrane and with f_i (representing $x_i = false$) in the second copy of the membrane. Simultaneously, the counter increases its subscript. Because the rules are used in parallel in all membranes, in n steps we get all 2^n truth assignment of the n variables.

A simple computation, for the case of three variables, is given in Fig. 5.

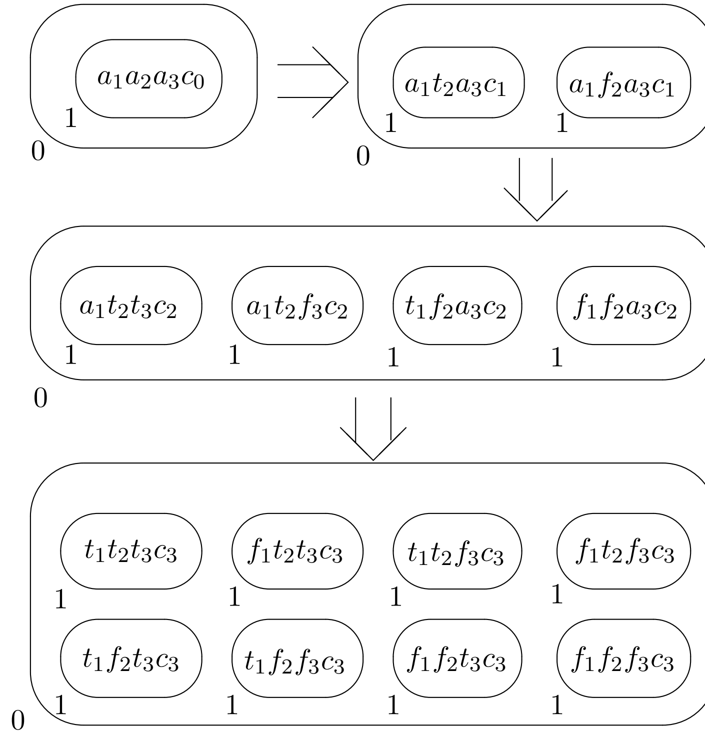

Fig. 5. Generating all truth assignments for 3 variables by membrane division

This figure also illustrates the idea of *confluence*: the variable x_i to expand in each step is not prescribed, the computation proceeds non-deterministically from this point of view, but eventually we reach the same configuration, with all truth assignments generated.

Returning now to the framework for dealing with complexity matters in terms of P systems, the used devices are the *accepting P systems with input*: a family of P systems of a given type is constructed starting from a given problem, and an instance of the problem is introduced as an input in one of such systems; working in a deterministic mode (or a *confluent* mode), the system sends to the environment the answer to the respective instance. The family of systems should be constructed in a uniform mode by a Turing machine, working a polynomial time.

This direction of research was much investigated. A large number of problems were considered, the membrane computing complexity classes were refined, characterizations of the **P**≠**NP** conjecture were obtained in this framework, several characterizations of the class **P**, even problems which are **PSPACE**-complete were proven to be solvable in polynomial time by means of membrane systems provided with membrane division or membrane creation. An important (and difficult) problem is that of finding the borderline between efficiency and non-efficiency: which ingredients should be used in order to be able to solve hard problems in a polynomial time? The reader is referred to the literature available at [46] for details – including many problems which are still open in this area.

The efficiency issue was also investigated for SN P systems, but we do not enter here into details, rather we refer to [24], [25], [7].

7. Applications

There are many features of membrane computing which make it attractive for applications in several disciplines, especially for biology and bio-medicine.

First, there are several features which are genuinely proper to membrane computing and

which are of interest for many applications: *distribution* (with the important system-part interaction, emergent behavior, non-linearly resulting from the composition of local behaviors), easy *programmability*, *scalability/extensibility*, *transparency* (multiset rewriting rules are nothing else than reaction equations as customarily used in chemistry and bio-chemistry), *parallelism*, *non-determinism*, *communication*, and so on and so forth.

Now, in what concerns the applications themselves reported up to now, they are developed at various levels. In many cases, what is actually used is the *language* of membrane computing, having in mind three dimensions of this aspect: (i) the long list of concepts either newly introduced, or related in a new manner in this area, (ii) the mathematical formalism of membrane computing, and (iii) the graphical language, the way to represent cell-like structures or tissue-like structures, together with the contents of the compartments and the associated evolution rules (the "evolution engine"). The next level is to use tools, techniques, results of membrane computing, and here there appears an important question: to which aim? Solving problems already stated, e.g., by biologists, in other terms and another framework, could be an impressive achievement, and this is the most natural way to proceed – but not necessarily the most efficient one, at least at a long term. New tools can suggest new problems, which either cannot be formulated in a previous framework or have no chance to be solved in the previous framework.

Applications of all these types were reported in the literature of membrane computing and, as expected, most of them were carried out in biology. These applications are usually based on experiments using programs for simulating/implementing P systems on usual computers, and there are already several such programs, more and more elaborated (e.g., with better and better interfaces, which allow for the friendly interaction with the program). An overview of membrane computing software reported in literature (some programs are available in the web page [46]) can be found in the volume [9]. Several applications are presented in detail – software included – at the web page of Sheffield membrane computing research group, `http://www.dcs.shef.ac.uk/∼marian/`, and at the page of Verona group, `www.cbmc.it`.

Of course, when using a P system for simulating a biological process we are no longer interested in its computing behavior (power, efficiency, etc.), but in its evolution in time; the P system is then interpreted as a dynamical system, and its trajectories are of interest, its "life". Moreover, the ingredients we use are different from those considered in theoretical investigations. For instance, in mathematical terms, we are interested in results obtained with a minimum of premises and with weak prerequisites, while the rules are used in ways inspired from automata and language theory (e.g., in a maximally or minimally parallel way), but when dealing with applications the systems are constructed in such a way to capture the features of reality (for instance, the rules are of a general form, they are applied according to probabilistic strategies, based on stoichiometric calculations, the systems are not necessarily synchronized, and so on).

The typical applications run as follows. One starts from a biological process described in general in graphical terms (chemicals are related by reactions represented in a graph-like manner, with special conventions for capturing the context-sensitivity of reactions, the existence of promoters or inhibitors, etc.) or already available in data bases in SBML (system biology markup language) form; these data are converted into a P system which is introduced in a simulator; the way the evolution rules (reactions) are applied is the key point in constructing this simulator (often, the classical Gilespie algorithm is used in compartments, or multi-compartmental variants of it are considered); as a result, the evolution in time of the multiplicity of certain chemicals is displayed, thus obtaining a graphical representation of the interplay in time of certain chemicals, their growth and decay, and so on. Many illustrations of this scenario can be found in the literature, many times dealing with rather complex processes.

Besides applications in biology, applications were reported in computer graphics (where the compartmentalization seems to add a significant efficiency to well-known techniques based on

Lindenmayer systems), linguistics (both as a representation language for various concepts related to language evolution, dialogue, semantics, and making use of the parallelism, in solving parsing problems in an efficient way), economics (where many bio-chemical metaphors find a natural counterpart, with the mentioning that the “reactions” which take place in economics, for instance, in market-like frameworks, are not driven only by probabilities/stoichiometric calculations, but also by psychological influences, which makes the modeling still more difficult than in biology), computer science (in devising sorting and ranking algorithms), cryptography, etc.

A very promising direction of research, namely, applying membrane computing in devising approximate algorithms for hard optimization problems, was initiated by Nishida, in [30], who proposed *membrane algorithms*, as a new class of distributed evolutionary algorithms. These algorithms can be considered as a high level (distributed and dynamically evolving their structure during the computation) evolutionary algorithms. In short, candidate solutions evolve in compartments of a (dynamical) membrane structure according to local algorithms, with better solutions migrating down in the membrane structure; after a specified halting condition is met, the current best solution is extracted as the result of the algorithm.

Nishida has checked this strategy for the traveling salesman problem, and the results were more than encouraging for a series of benchmark problems: the convergence is very fast, the number of membranes is rather influential on the quality of the solution, the method is reliable, both the average quality and the worst solutions were good enough and always better than the average and the worst solutions given by simulated annealing.

Similarly good results were obtained in a series of subsequent papers which have followed the same approach in addressing other hard optimization problems. In the bibliography below we have recalled a few recent titles, dealing mainly with the application of membrane algorithms in solving hard optimization problems. Always, benchmark problems were considered and the results were compared with those provided by other methods, existing in literature.

8. Closing Remarks

Although so much developed in the less than nine years since the investigations were initiated, membrane computing still has a large number of open problems and research topics which wait for research efforts, and new areas are continuously appearing – the most recent one is the study of spiking neural P systems.

A general class of theoretical questions concerns the borderline between universality and non-universality or between efficiency and non-efficiency, i.e., concerning the succinctness of P systems able to compute at the level of Turing machines or to solve hard problems in polynomial time, respectively. Then, because universality implies undecidability of all non-trivial questions, an important issue is that of finding classes of P systems with decidable properties.

This is also related to the use of membrane computing as a modeling framework: if no insights can be obtained in an analytical manner, algorithmically, then what remains is to simulate the system on a computer. To this aim, better programs are still needed, maybe parallel implementations, able to handle real-life questions (for instance, in the quorum sensing area, existing applications deal with hundreds of bacteria, but biologists would need simulations at the level of thousands of bacteria in order to get convincing results).

Several research topics concern the neural inspiration for membrane computing, starting with the need of introducing a more elaborated model of neural nets. An important question in this respect is to use neuro-inspired models for efficiently solving problems, maybe borrowing ideas from the traditional neural computing (learning, solving pattern matching problems, etc.). This issue seems to also open new research directions for the traditional computability theory. Here

is only one example. Efficiency is usually achieved in membrane computing by means of tools which allow producing an exponential working space in a linear time, and the standard way to do it is membrane division. However, in SN P systems we do not have such possibilities, the number of neurons remains the same and the number of spikes only increases polynomially with respect to the number of steps of a computation. How to introduce possibilities of generating an exponential workspace in a linear time remains as a research topic. Still, with inspiration from the fact that the brain consists of a huge number of neurons out of which only a small part are used in each moment, in [7] one proposes a way (illustrated for `SAT`) to address computationally hard problems in this framework, by assuming that an arbitrarily large SN P system is given "for free", pre-computed, with a structure as regular as possible, and without spikes inside; solving a problem starts by introducing a polynomial number of spikes in a polynomially bounded number of neurons; then, by moving spikes along synapses, the system self-activates, and a specific output provides the answer to the problem. This way of solving problems, by activating a pre-computed resource, is not at all usual in computability and is a research direction worth exploring.

References

1. A. Alhazov, R. Freund and Y. Rogozhin, Computational power of symport/antiport: history, advances, and open problems, in: [12], 44–78.
2. F. Bernardini and M. Gheorghe, Population P systems, *Journal of Universal Computer Science* **10**, 5 (2004), 509–539.
3. F. Bernardini and V. Manca, P systems with boundary rules, in: [41], 107–118.
4. C.S. Calude, Gh. Păun, G. Rozenberg and A. Salomaa (eds.), *Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, LNCS **2235**, Springer, Berlin, 2001.
5. L. Cardelli and Gh. Păun, An universality result for a (mem)brane calculus based on mate/drip operations, *International Journal of Foundations of Computer Science* **17**, 1 (2006), 49–68.
6. M. Cavaliere, O. Egecioglu, O.H. Ibarra, S. Woodworth, M. Ionescu and Gh. Păun, Asynchronous spiking neural P systems: decidability and undecidability, *DNA 13*, 2005.
7. H. Chen, M. Ionescu and T.-O. Ishdorj, On the efficiency of spiking neural P systems, *Proceedings of 8th International Conference on Electronics, Information, and Communication*, Ulanbator, Mongolia, June 2006, 49–52.
8. G. Ciobanu, L. Pan, Gh. Păun and M.J. Pérez-Jiménez, P systems with minimal parallelism, *Theoretical Computer Science* **378**, 1 (2007), 117–130.
9. G. Ciobanu, Gh. Păun and M.J. Pérez-Jiménez (eds.), *Applications of Membrane Computing*, Springer, Berlin, 2006.
10. E. Csuhaj-Varjú, P automata. Models, results, and research topics, in: [28], 1–11.
11. R. Freund, L. Kari, M. Oswald and P. Sosik, Computationally universal P systems without priorities: two catalysts are sufficient, *Theoretical Computer Science* **330**, 2 (2005), 251–266.
12. R. Freund, Gh. Păun, G. Rozenberg and A. Salomaa (eds.), *Membrane Computing, 6th International Workshop, WMC6, Vienna, Austria, July 2005, Revised, Selected, and Invited Papers*, LNCS **3859**, Springer, Berlin, 2006.
13. M.A. Gutiérrez-Naranjo, Gh. Păun and M.J. Pérez-Jiménez (eds.), *Cellular Computing. Complexity Aspects.* Fenix Editora, Sevilla, 2005.

14. H.J. Hoogeboom, Gh. Păun, G. Rozenberg and A. Salomaa (eds.), *Membrane Computing, International Workshop, WMC7, Leiden, The Netherlands, 2006, Revised, Selected, and Invited Papers*. LNCS **4361**, Springer, Berlin, 2006.
15. L. Huang, N. Wang, An optimization algorithms inspired by membrane computing, *Proc. ICNC 2006*, LNCS **4222** (L. Jiao et al., eds.), Springer, Berlin, 2006, 49–55.
16. O.H. Ibarra, On membrane hierarchy in P systems, *Theoretical Computer Science* **334**, 1-3 (2005), 115–129.
17. O.H. Ibarra, On determinism versus nondeterminism in P systems, *Theoretical Computer Science* **344**, 2-3 (2005), 120–133.
18. O.H. Ibarra and S. Woodworth, On bounded symport/antiport P systems, *Proc. DNA 2005*, LNCS **3892**, Springer, Berlin, 2006, 129–143.
19. O.H. Ibarra and S. Woodworth, Spiking neural P systems: some characterizations, *Proc. FCT 2007*, Budapest, LNCS **4639**, Springer, Berlin, 2007, 23–37.
20. O.H. Ibarra and H.C. Yen, Deterministic catalytic systems are not universal, *Theoretical Computer Science* **363**, 2 (2006), 149–161.
21. M. Ionescu and Gh. Păun, T. Yokomori, Spiking neural P systems, *Fundamenta Informaticae* **71**, 2-3 (2006), 279–308.
22. N. Jonoska and M. Margenstern, Tree operations in P systems and λ-calculus *Fundamenta Informaticae* **59**, 1 (2004), 67–90.
23. J. Kleijn and M. Koutny, Synchrony and asynchrony in membrane systems, in: [14], 66–85.
24. A. Leporati, C. Zandron, C. Ferretti and G. Mauri, Solving numerical NP-complete problems with spiking neural P systems, in: *Membrane Computing, International Workshop, WMC8, Thessaloniki, Greece, 2007, Selected and Invited Papers* (G. Eleftherakis, P. Kefalas, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS **4860**, Springer-Verlag, Berlin, 2007.
25. A. Leporati, C. Zandron, C. Ferretti and G. Mauri, On the computational power of spiking neural P systems, *Inter. J.Unconventional Computing*, 2007, in press.
26. V. Manca, MP systems approaches to biochemical dynamics: biological rhythms and oscillations, in: [14], 86–99.
27. C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg and A. Salomaa, (eds.), *Membrane Computing. International Workshop, WMC2003, Tarragona, Spain, Revised Papers*. LNCS **2933**, Springer, Berlin, 2004.
28. G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg and A. Salomaa, (eds.), *Membrane Computing, International Workshop, WMC5, Milano, Italy, 2004, Selected Papers*. LNCS **3365**, Springer, Berlin, 2005.
29. M. Minsky, *Computation – Finite and Infinite Machines*, Prentice Hall, Englewood Cliffs, NJ, 1967.
30. T.Y. Nishida, An application of P systems: A new algorithm for NP-complete optimization problems, *Proceedings of the 8th World Multi-Conference on Systems, Cybernetics and Informatics* (N. Callaos et. al., eds.), Vol. V, 2004, 109–112.
31. A. Păun and Gh. Păun, The power of communication: P systems with symport/antiport, *New Generation Computers* **20**, 3 (2002), 295–306.
32. A. Păun and Gh. Păun, Small universal spiking neural P systems *BioSystems* **90**, 1 (2007), 48–60.
33. A. Păun, M.J. Pérez-Jiménez and F.J. Romero-Campero, Modeling signal transduction using P systems, in: [14], 100–122.

34. Gh. Păun, Computing with membranes, *Journal of Computer and System Sciences* **61**, 1 (2000), 108–143 (and Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).

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

36. Gh. Păun, *Membrane Computing. An Introduction*, Springer, Berlin, 2002.

37. Gh. Păun and R. Păun, Membrane computing as a framework for modeling economic processes, *Proceedings of SYNASC Conference*, Timişoara, 2005, IEEE Press, 2005, 11–18.

38. Gh. Păun, J. Pazos, M.J. Pérez-Jiménez and A. Rodriguez-Patón, Symport/antiport P systems with three objects are universal, *Fundamenta Informaticae* **64**, 1-4 (2005), 353–367.

39. Gh. Păun, M.J. Pérez-Jiménez and A. Salomaa, Spiking neural P systems. An early survey, *International Journal of Foundations of Computer Science* **18** (2007), 435–456.

40. Gh. Păun and G. Rozenberg, A guide to membrane computing, *Theoretical Computer Science* **287**, 1 (2002), 73–100.

41. Gh. Păun, G. Rozenberg, A. Salomaa and C. Zandron (eds.), *Membrane Computing. International Workshop, WMC 2002, Curtea de Argeş, Romania, August 2002, Revised Papers*, LNCS **2597**, Springer, Berlin, 2003.

42. M.J. Pérez-Jiménez, An approach to computational complexity in membrane computing, in: [28], 85–109.

43. M.J. Pérez-Jiménez, A. Romero-Jiménez and F. Sancho-Caparrini, *Teoria de la complejidad en modelos de computacion celular con membranas*, Kronos Editorial, Sevilla, 2002.

44. P. Sosik and A. Rodriguez-Patón A, Membrane computing and complexity theory: A characterization of PSPACE, *International Journal of Foundations of Computer Science* **73**, 1 (2007), 137–152.

45. D. Zaharie and G. Ciobanu, Distributed evolutionary algorithms inspired by membranes in solving continuous optimization problems, in: [14], 536–554.

46. The P Systems Web Page, `http://psystems.disco.unimib.it` (with a mirror at `http://bmc.hust.edu.cn/psystems`).