Front. Comput. Sci. China 2007, 1(1): 37–49
DOI 10.1007/s11704-007-0005-4

REVIEW ARTICLE

# A quick overview of membrane computing with some details about spiking neural P systems

**Gheorghe Păun** (✉)[1,2]

1 Institute of Mathematics of the Romanian Academy, P O Box 1-764, 014700 Bucureşti, Romania
2 Department of Computer Science and Artificial Intelligence University of Sevilla, Avda. Reina Mercedes s/n, 41012 Sevilla, Spain



**Abstract** We briefly present the basic elements of membrane computing, a branch of natural computing inspired by the structure and functioning of living cells, then we give some details about spiking neural P systems, a class of membrane systems recently introduced, with motivations related to the way neurons communicate by means of spikes. In both cases, of general P systems and of spiking neural P systems, we introduce the fundamental concepts, give a few examples, then recall the types of results and of applications. A series of bibliographical references are provided.

**Keywords** natural computing, membrane computing, P system, spiking neural P system, Turing computability, chomsky hierarchy, computational biology

## 1 Introduction

Membrane computing is a branch of natural computing initiated in Ref. [1], which abstracts computing models from the structure and the functioning of living cells, as well as from the organization of cells in tissues or other higher order structures. Thus, the initial goal of membrane computing was to learn from the biology of the cell something possibly useful to computer science, and the area quickly developed in this direction. Several classes of computing models (called P systems) were defined in this context, inspired from biological facts or motivated from mathematical or computer science points of view. A series of applications were reported in recent years, in biology/medicine, linguistics, computer graphics, economics, approximate optimization, cryptography, etc.

The literature of the domain is very large (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) and the progresses rather rapid. At the end of 2006, the bibliography of the field counted more than 820 titles–see Ref. [2].

Consequently, this is a very short introduction to membrane computing, only pointing a few basic notions and only presenting some (types of) results and of applications. The reader interested in details should consult Refs. [3, 4], and the comprehensive bibliography from Ref. [2].

Besides applications, a recent tendency in membrane computing is the study of models inspired from the brain organization, that is why we give here some details about spiking neural P systems.

Received November 15, 2006; accepted December 30, 2006

E-mail: george.paun@imar.ro, gpaun@us.e

## 2 Basic elements of membrane computing

The field started by looking to the cell in order to learn something possibly useful to computer science, but then the research also considered cell organization in tissues (in general, populations of cells, such as colonies of bacteria), and, recently, also neuron organization in the brain. Thus, there are in this moment 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), delimiting compartments where multisets of symbol objects are placed; rules for evolving these multisets as well as the membranes 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.

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), but 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*.

Finally, there are two types of neural-like P systems. One of them is similar to tissue-like P systems in the fact 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* that controls the evolution. A promising device 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 a considerable interest, while the neural-like systems, mainly under the form of spiking neural P systems, are only recently investigated. Applications were reported so far only for the first two classes of P systems, the cell-like and the tissue-like ones.

In what follows, in order to let the reader have a flavor of membrane computing, we will discuss in some detail cell-like P systems and spiking neural P systems, and we refer to the area literature for other classes.

## 2.1 Cell-like P systems; an informal presentation

Because in this section we only consider cell-like P systems, they will be simply called P systems.

In brief, 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 from 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, although it started recently to be investigated), 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, that is why 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.

The objects evolve according to *rules* that are also associated with the regions. The rules say both how the objects are changed and how they can be moved (*communicated*) across membranes. There also are rules that 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 from 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 the chemical reactions are performed – in parallel, with the mathematical additional restriction to have a *maximal parallelism*: one applies a bunch of rules that is maximal, no further object can evolve at the same time by any rule. Besides this mode, several others were also 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). In all cases, a common feature is that the objects 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 that *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 the halting configuration in a specified membrane.

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 automata 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. 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 a decision problem and the output is one of two special objects representing the answers yes and no to the problem.

Thus, we can address several types of problems in this framework. The main two concerns the computing power of P systems (working in the generative or accepting modes), and their usefulness in solving decision problems (making use of the inherent parallelism, it is expected that fast solutions to problems that are hard to solve by sequential algorithms can be found). From both these two points of view, the results are quite attractive: P systems with simple ingredients (number of membranes, forms and sizes of rules, controls of using the rules) are Turing complete, while classes of P systems with enhanced parallelism (e.g., having rules for membrane division) can provide polynomial solutions to **NP**-complete (even **PSPACE**-complete) problems.

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 as-

sociated with the evolution rules); we do not enter into details here, rather we refer the reader to the bibliography from [2]. Similarly, in applications to biology, specific strategies of evolution are considered – see also Section 5.

## 2.2 Basic ingredients of P systems

Let us now go into some more specific details – still remaining at an informal level.

As said 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. The membranes both define protected "reactors", where specific biochemical reactions take place (starting with the cell membrane, which delimits and protects the cell from the environment), and contain proteins that catalyze reactions and, through protein channels, ensure the passage of chemicals from a compartment of the cell to another compartment, as well as the communication with the environment. Thus, the main ingredient of a P system is the *membrane structure*, a hierarchical arrangement of membranes, which delimits compartments; in these compartments there are objects that evolve by means of rules that are also localized, assigned to compartments. Figure 1 illustrates this notion and the related terminology.

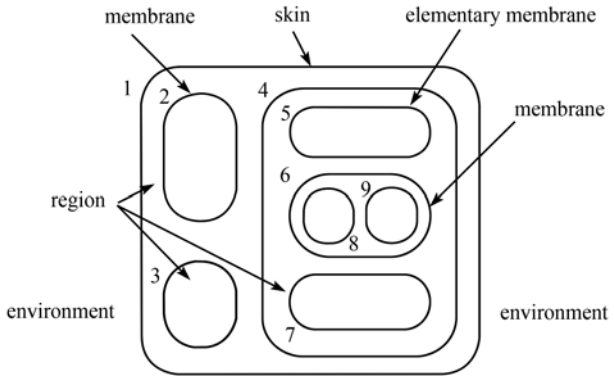


**Fig. 1** A membrane structure

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.

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 rule as above, with several objects in its left hand member, is said to be *cooperative*; a particular case is that of *catalytic* rules, of the form $ca \rightarrow cx$, where $a$ is an object and $c$ is a catalyst, appearing only in such rules, never changing. A rule of the form $a \rightarrow x$, where $a$ is an object, is called *non-cooperative*.

The rules associated with a compartment are applied to the objects from that compartment. The most investigated way to use the rules is the *maximally parallel one*: all objects that can evolve by means of local rules should do it (we assign objects to rules, until no further assignment is possible). The used objects are "consumed", the newly produced objects are placed in the compartments of the membrane structure according to the communication commands assigned to them. The rules to be used and the objects to evolve are chosen in a non-deterministic manner. In turn, all compartments of the system evolve at the same time, synchronously (a common clock is assumed for all membranes). Thus, we have two layers of parallelism, one at the level of compartments and one at the level of the whole "cell".

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 that 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 that 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* (one also uses to say *generates*) a set of numbers, or a set of vectors of numbers, or a language.

As mentioned in the previous subsection, a P system can also be used in the accepting mode, with a particular case being that of solving decision problems, which will be discussed further in Section 4.

### 2.3 A large number of variants

Let us start by considering the possibility offered by the form of rules. In the systems described above, the symbol objects were processed by multiset rewriting-like rules (some objects are transformed into other objects, which have associated communication targets; the resemblance with chemical reactions is obvious). Coming closer to the trans-membrane transfer of molecules, we can consider rules that model the active passage of chemicals through membranes, by so-called *uniport, symport*, and *antiport* (see Ref. [5] for details). Symport refers to the transport where two (or more) molecules pass together through a membrane in the same direction, antiport refers to the transport where two (or more) molecules pass through a membrane simultaneously, but in opposite directions, while the case when a molecule does not need a "partner" for a passage is referred to as uniport.

In mathematical terms, we can consider object processing rules of the following forms: a symport rule (associated with a membrane $i$) is of the form ($ab$, *in*) or ($ab$, *out*), stating that the objects $a$ and $b$ enter/exit together membrane $i$, while an antiport rule is of the form ($a$, *out*; $b$, *in*), stating that, simultaneously, $a$ exits and $b$ enters membrane $i$; uniport corresponds to a particular case of symport rules, of the form ($a$, *in*), ($a$, *out*). An obvious generalization is to consider symport rules ($x$, *in*), ($x$, *out*) and antiport rules ($x$, *out*; $y$, *in*) with $x$, $y$ arbitrary multisets of objects.

Symport/antiport rules can be used alone, thus leading to symport/antiport P systems, or in combination with multiset rewriting rules. In the first case, because by communication we do not create new objects, we need a supply of objects in the environment, otherwise we are only able to handle a finite population of objects, those provided in the initial multisets. Thus, the environment takes an active part in the computation, which is an attractive feature of this class of P systems, together with the conservation of objects, the mathematical elegance, the computational power, the direct biological inspiration. Also the case of evolving objects by means of multiset rewriting rules and communicating by symport/antiport rules leads to a rather interesting class of P systems, the so-called evolution-communication P systems.

Recently, efforts have been made to also take into consideration the fact that the cell biochemistry is controlled to a large extent by the proteins embedded in the membranes. For instance, rules of the form $a[_i p|b \rightarrow a'[_i p'|b'$ are proposed, where $a$, $a'$, $b$, $b'$ are objects, $p$, $p'$ are proteins, and $[_i p|$ is a notation of the fact that $p$ is placed on membrane $i$; several restrictions can be considered, for instance, with $p = p'$, and/or $a = b'$, $b = a'$, etc.

We can then pass to rules that handle not only objects, but also membranes. There is a large list of suggestions coming from biology: membranes can be broken (and the contents remain free in the surrounding region), divided (with the replication of the contents), their contents can be merged or separated according to given criteria; then, there are operations like exocytosis and endocytosis/phagocytosis, budding, matting, gemmating (sending vesicles at specified destinations), and so on and so forth. These last rules ensure that not only the multisets of objects evolve, but also the membrane structure of a P system.

Many possibilities arise in what concerns the way the rules are used. More precisely, the non-determinism of choosing the rules to apply can be decreased in various ways: using a priority among rules (a partial order relation), using promoters (objects that should be present in order to apply a rule – see the first example below) or inhibitors (objects that should not be present), controlling the permeability of membranes (some rules can increase the permeability of the membrane where they are used, other rules can decrease the permeability; a rule that asks for sending an object across a membrane that is not permeable cannot be applied, and in this way the rules can influence the way the next rules are chosen).

Then, we can use the rules in the maximally parallel manner, but also in other ways: sequentially (one rule in the whole system, or in each region), with a bounded parallelism (at least $k$ or exactly $k$ rules in the whole system or in each region), with a minimal parallelism (at least one rule is used in each region where a rule can be used), synchronously or asynchronously, etc. These strategies of applying the rules are biologically inspired but mathematically oriented; when using P systems as models of biological systems/processes, we have to use more realistic features, in general, of a numerical nature (e.g., reaction rates, probabilities), computed dynamically, depending on the current population of objects in the system.

In general, a P system is formalized as a construct $\Pi = (O, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, i_0)$, where $O$ is the alphabet of objects, $\mu$ is the membrane structure, with $m$ membranes, $w_1, \cdots, w_m$ are multisets of objects present in the $m$ regions of $\mu$ at the beginning of a computation, $R_1, \cdots, R_m$ are finite sets of evolution rules, associated with the regions of $\mu$, and $i_0$ is the label of a membrane, used as the output membrane.

We do not give a more formal definition of a P system here. The reader interested in mathematical and bibliographical details can consult the mentioned Ref. [2], as well as the relevant papers from Ref. [3] Rather, in the next section we present two simple examples, just to illustrate the architecture and the functioning of a (cell-like) P system.

### 2.4 Two examples

Both examples present systems that compute a function, namely $n \rightarrow n^2$, for any natural number $n \geqslant 1$. The initialconfiguration of the first system is given in Figure 2. Besides catalytic and non-cooperating rules, it also contains a rule with promoters, $b_2 \rightarrow b_2 e_{in} | b_1$: 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.

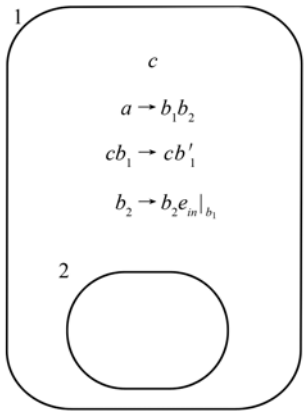


**Fig. 2** A P system with catalysts and promoters

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 no longer exists. 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.

The second system is presented in Figure 3. It computes the same function, but in a slightly different way: this time the functioning of the system is not deterministic; if the computation will halt, then the result is obtained and it is the correct one, if not, then we get no result (a non-halting computation provides no output).

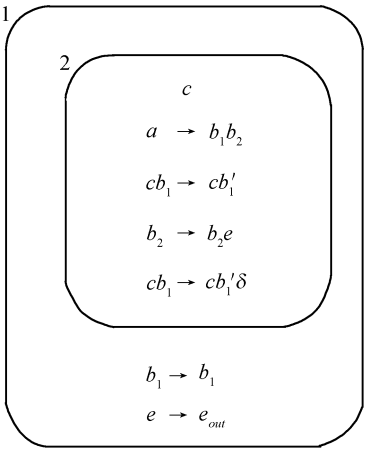


**Fig. 3** A P system with membrane dissolution

We start again with a copy of the catalyst in the system, and we introduce *n* copies of *a* in membrane 2 in order to compute the square of *n*. As above, we change all objects *a* into $b_1$, $b_2$, then one $b_1$ becomes primed and, in parallel, each copy of $b_2$ introduces a copy of *e*. The use of the rule $b_2 \to b_2 e$ can continue forever, and the only way to prevent this is to dissolve membrane 2 by means of using the rule $cb_1 \to cb'_1 \delta$ (the symbol $\delta$ indicates that if this rule is applied, the membrane is dissolved; dissolving a membrane means to let all its objects free in the surrounding membrane, but the rules of the dissolved membrane are removed -- the intuition is that these rules are specific to the "reactor" enclosed by the membrane, hence, if the reactor is lost, its rules are also lost).

The rule $cb_1 \to cb'_1 \delta$ can be used at any step instead of the rule $cb_1 \to cb'_1$. However, if the dissolution of membrane 2 happens too early − while there are copies of $b_1$ that are not primed − then the computation never stops: the rule $b_1 \to b_1$ from region 1 will be used forever. Therefore, in order to stop the computation, the rule $cb_1 \to cb'_1 \delta$ must be used precisely for the last object $b_1$ present in membrane 2; in this case, only primed versions of $b_1$ arrive in the skin region and no rule can be applied here to this object. After dissolving membrane 2, all copies of *e* (and, again, there are $n^2$ such copies) arrive in membrane 1 and from here they exit immediately, all in one step, by means of the rule $e \to e_{out}$. Consequently, this time the result is obtained in the environment.

We stress the fact that these systems were *computing* systems: an input is introduced and an output is obtained after (successful) computations. If this output is of the yes/no type, then the system will be an accepting/recognizing one. If we simply start from an initial configuration and we collect all possible outputs (they can be several, because of the non-determinism), then we will have a generative system. We do not give further examples here, the reader is refereed to the literature for details.

## 3 Computing power: easy universality

As we have mentioned before, many classes of P systems, combining various ingredients (as described above or similar) are able of simulating Turing machines, hence they are *computationally complete*. Always, the proofs of results of this type are constructive, and this have an important consequence from the computability point of view: there are *universal* (hence *programmable*) P systems. In brief, 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 sufficient in orderto characterize the power of Turing machines by means of P systems is always rather small.

We only mention here three of the most interesting (types of) universality results:

1. P systems with symbol-objects with catalytic rules, using only two catalysts and two membranes, are computationally universal [6].

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 [7].

3. P systems with symport/antiport rules (of arbitrary size), using only four membranes and only three objects, are universal [8].

There are several results of the form of those mentioned in the points 1 and 2 above, improvements or extensions of these results; details can be found in Refs. [2, 9, 10].

We can conclude that the compartmental computation in a cell-like membrane structure (using various ways of communicating among compartments) is rather powerful. The "computing cell" is a powerful "computer".

Universality results were obtained also in the case of P systems working in the accepting mode. An interesting problem appears in this case, because we can consider 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 Ref. [11], for the accepting catalytic P systems.

The hierarchy on the number of membranes collapses in many cases also for non-universal classes of P systems [3], but there also are cases when "the number of membrane matters", to cite the title of Ref. [12], where two classes of P systems were defined for which the hierarchies on the number of membranes are infinite.

## 4 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 such main 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*.

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). Details can be found in Refs. [3, 13], as well as at the web page [2].

Roughly speaking, the framework for dealing with complexity matters is that of *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 such systems; working in a deterministic mode (or a *confluent* mode: some non-determinism is allowed, provided that the branching converges after a while to a unique configuration, or, in the weak confluent case, all computations halt and all of them provide the same result), in a given time one of the answers yes/no is obtained, in the form of specific objects sent to the environment. The family of systems should be constructed in a uniform mode by a Turing machine, working a polynomial time.

This direction of research is very active at the present moment. More and more problems are considered, the membrane computing complexity classes are refined, characterizations of the $\mathbf{P} \neq \mathbf{NP}$ conjecture were obtained in this framework, several characterizations of the class **P**, even problems that are **PSPACE**-complete were proven to be solvable in polynomial time by means of membrane systems provided with membrane division or membrane creation.

We do not enter into details here, but we refer, e.g., to the chapter from Ref. [4] devoted to this topic, to Ref. [13], and to Ref. [14].

## 5 A glimpse of applications

As mentioned above, membrane computing was initiated having as primary goals computability in general and natural computing in particular, without aiming to faithfully model biological facts in such a way to provide a modeling framework for the use of biologists. However, after significantly developing at the theoretical level, the domain started to be useful for biological and medical applications, in general in the following scenario: a membrane computing model is written for a given process taking place in the cell, a program is written (or one of the many computer programs available is used) for simulating the model, and then computer experiments are carried out, tuning certain parameters

and playing with the inputs, thus generating (numerical or graphical) data of interest for the study of the process considered.

The approach has a series of features that answer several of the drawbacks of models based on differential equations: modularity (intrinsic to a membrane system), scalability/extensibility (further membranes and/or further evolution rules can be added without essentially changing the way of working with the system), understandability (evolution rules directly correspond to chemical reactions), programmability (a rewriting-based model can be easily transformed into a program, with certain programming languages, such as CLIPS, perfectly adequate to such a job), while preserving other good features of differential equations models, such as non-linearity of evolution.

The range of applications reported so far is very large: respiration in bacteria, photosynthesis, population dynamics in eco-systems, oscillations in chemical reactions, circadian rhythms, quorum sensing in bacteria, healing of knee injuries, robustness of certain parameters evolution in EGFR-related processes, apoptosis, and so on. We refer to Ref. [4] for bibliographical information, and to Refs. [2, 15] for recent information, software for applications, and case studies.

Besides applications in biology, membrane computing was also considered in other areas, such as computer graphics (models based on compartmentalized Lindenmayer systems proved to be more powerful and more efficient than those using classic L systems), in linguistics, economics, etc. Again, we refer to Refs. [2, 4] for details.

## 6 Nishida's membrane algorithms

A very promising direction of research, namely, applying membrane computing in devising approximate algorithms for hard optimization problems, was initiated by Nishida [16] (see also Ref. [17]), who proposed *membrane algorithms*. These algorithms can be considered as high-level (distributed and dynamically evolving their structure during the computation) evolutionary algorithms. The basic variant is the following one: a (small) number of candidate solutions to an optimization problem are placed in the regions of a membrane structure of a linear shape (with the membranes embedded in one another), together with local sub-algorithms that can improve the local solutions; after a (small) number of steps of local work, when the solutions from each membrane are evolved, the best of them is sent to the immediately lower membrane and the worst is sent to the immediately upper membrane (with exceptions to this rule in the innermost and the outermost membrane); in this way, the better solutions are moved down and the worst ones are moved up in the membrane hierarchy; this process is iterated until either a specified number of steps is reached, or no improvement of the best solution is obtained for a specified number of steps. When halting, the central membrane provides the answer, the solution to the problem. There are several variants, in terms of the number of membranes, with the initial solutions generated by a first generation of membrane algorithms (thus working in a two-stage manner, which proves to be very efficient), with the possibility to create or to destroy certain membranes during the computation, etc. (This approach is similar to distributed evolutionary computing, but it has a series of features inspired from membrane computing.)

Nishida has checked this strategy for the traveling salesman problem, and the results were more than encouraging. Known benchmark problems were addressed and always the conclusions were the same: at the beginning, the convergence is very fast, so that, after a small number of steps we get a solution that is almost the best that can be provided by the algorithm, irrespective of how much we continue to work; the number of membranes is rather influential on the quality of the solution, but, depending also on the dimension of the problem (the number of nodes in the graph), 50 to 100 membranes are sufficient for obtaining a solution that is better than that provided by simulated annealing; 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 that have followed the same approach in addressing other hard optimization problems. We mention here only a few recent papers (their titles are self-explanatory in what concerns the problems considered): Refs. [18–21]. Always, benchmark problems were considered and the results were compared with those provided by other methods, existing in literature.

## 7 Spiking neural P systems

Spiking neural P systems (SN P systems, for short) were introduced in Ref. [22] with the aim of defining membrane systems based on ideas specific to spiking neurons, currently much investigated in neural computing (see, e.g., Refs. [23, 24]). The resulting models are a variant of tissue-like and neural-like P systems.

### 7.1 A quick overview of the domain

Briefly, 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. We will give a formal definition in Subsection 7.2. There

also are *forgetting rules*, of the form $a^s \to \lambda$, with the meaning that $s \geqslant 1$ 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 that 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. (This idea, of taking the distance between two events as the result of a computation, was already considered for symport/antiport and for catalytic P systems in Ref. [25].) In Ref. [22] only the distance between the first two spikes of a spike train was considered, then in Ref. [26] several extensions were examined: the distance between the first $k$ spikes of a spike train, or the distances between all consecutive spikes, taking into account all intervals or only intervals that alternate, all computations or only halting computations, etc.

Systems working in the accepting mode were also considered: 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.

Two main types of results were obtained: computational completeness in the case when no bound was imposed on the number of spikes present in the system, and a characterization of semilinear sets of numbers in the case when a bound was imposed.

Another attractive possibility is to consider the spike trains themselves as the result of a computation, and then we obtain a (binary) language generating device. We can also consider input neurons and then an SN P system can work as a transducer. Such possibilities were investigated in Ref. [27]. Languages−even on arbitrary alphabets−can be obtained also in other ways: following the path of a designated spike across neurons (this essentially resembles the trace languages investigated for usual P systems, see Refs. [2, 3]), or generalizing the form of rules. Specifically, one uses rules of the form $E/a \to a^p; d$, 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. This case was investigated in Ref. [28].

Other extensions were proposed in Refs. [29, 30], where several output neurons were considered, thus producing vectors of numbers, not only numbers. A detailed typology of systems (and generated sets of vectors) is investigated in the two papers mentioned above, with classes of vectors found in between the semilinear and the recursively enumerable ones.

The proofs of all computational completeness results known up to now in this area are based on simulating register machines. Starting the proofs from small universal register machines, as those produced in Ref. [31], one can find small universal SN P systems (working in the generating mode, as sketched above, or in the computing mode, i.e., having both an input and an output neuron and producing a number related to the input number). This idea was explored in Ref. [32] and the results are as follows: 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.

In the initial definition of SN P systems several ingredients are used (delay, forgetting rules), some of them of a general form (general synapse graph, general regular expressions). As shown in Ref. [33], rather restrictive normal forms can be found, in the sense that some ingredients can be removed or simplified without losing the computational completeness. For instance, the forgetting rules or the delay can be removed, both the indegree and the outdegree of the synapse graph can be bounded by 2, while the regular expressions from firing rules can be of very restricted forms.

### 7.2 A formal definition

A *spiking neural P system* (in short, an SN P system), of degree $m \geqslant 1$, is a construct of the form

$\Pi = (O, \sigma_1, \cdots, \sigma_m, syn, out)$,

where:

1. $O = \{a\}$ is the singleton alphabet ($a$ is called *spike*);
2. $\sigma_1, \cdots, \sigma_m$ are *neurons*, of the form

$\sigma_i = (n_i, R_i)$, $1 \leqslant i \leqslant m$,

where:

a) $n_i \geqslant 0$ is the *initial number of spikes* contained by the neuron;

b) $R_i$ is a finite set of *rules* of the following two forms:

(1) $E/a^c \to a; d$, where $E$ is a regular expression with $a$ the only symbol used, $c \geqslant 1$, and $d \geqslant 0$;

(2) $a^s \to \lambda$, for some $s \geqslant 1$, with the restriction that $a^s \in L(E)$ for no rule $E/a^c \to a; d$ of type (1) from $R_i$;

3. $syn \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\}$ with $(i, i) \notin syn$ for $1 \leqslant i \leqslant m$ (*synapses*);

4. $out \in \{1, 2, \cdots, m\}$ indicates the *output neuron*.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows: if the neuron contains $k$ spikes, $a^k \in L(E)$ and $k \geqslant c$, then the rule $E/a^c \to a; d$ can be applied, and this means that $c$ spikes are consumed, only $k–c$ remains in the neuron, the neuron is fired, and it produces a spike after $d$ time units (a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized). If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, and so on. In the case $d \geqslant 1$, if the rule is used in step $t$,

then in steps $t$, $t + 1$, $t + 2, \cdots$, $t + d-1$ the neuron is *closed*, and it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then the spike is lost). In step $t + d$, the neuron spikes and becomes open again, and hence can receive spikes (which can be used in step $t + d + 1$). A spike emitted by a neuron $\sigma_i$ replicates and goes to all neurons $\sigma_j$ such that $(i, j) \in syn$. If in a rule $E/a^c \to a$; $d$ we have $L(E) = \{a^c\}$, then we write it in the simpler form $a^c \to a$; $d$.

The rules of type (2) are *forgetting* rules, and they are applied as follows: if the neuron contains exactly $s$ spikes, then the rule $a^s \to \lambda$ can be used, and this means that all $s$ spikes are removed from the neuron.

In each time unit, in each neuron that can use a rule we have to use a rule, either a firing or a forgetting one. Because two firing rules $E_1 / a^{c_1} \to a; d_1$ and $E_2 / a^{c_2} \to a; d_2$ can have $L(E_1) \bigcap L(E_2) \neq \varnothing$, it is possible that two or more rules can be applied in a neuron, and then one of them is chosen non-deterministically. Note however that we cannot interchange a firing rule with a forgetting rule, as all pairs of rules $E/a^c \to a$; $d$ and $a^s \to \lambda$ have disjoint domains, in the sense that $a^s \notin L(E)$.

The initial configuration of the system is described by the numbers $n_1$, $n_2, \cdots$, $n_m$ of spikes present in each neuron. During a computation, the system is described both by the numbers of spikes present in each neuron and by the state of each neuron, in the open-closed sense. Specifically, if a neuron is closed, we have to specify the number of steps until it will become open again, i.e., the configuration is written in the form $\langle p_1 / q_1, \cdots, p_m / q_m \rangle$; the neuron $\sigma_i$ contains $p_i \geqslant 0$ spikes and will be open after $q_i \geqslant 0$ steps ($q_i = 0$ means that the neuron is already open).

Using the rules as suggested above, we can define transitions among configurations. A transition between two configurations $C_1$, $C_2$ is denoted by $C_1 \Rightarrow C_2$. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where all neurons are open and no rule can be used. With any computation, halting or not, we associate a *spike train*, the sequence $t_1, t_2, \cdots$ of natural numbers $1 \leqslant t_1 < t_2 < \cdots$, indicating time instances when the output neuron sends a spike out of the system (we also say that the system itself spikes at that time).

In Ref. [22], with any spike train containing at least two spikes one associates a result, in the form of the number $t_2 - t_1$; we say that this number is computed by $\Pi$. The set of all numbers computed in this way by $\Pi$ is denoted by $N_2(\Pi)$ (the subscript indicates that we only consider the distance between the first two spikes of any computation; note that 0 cannot be computed, that is why we disregard this number when estimating the computing power of any device).

This idea was extended in Ref. [26] to several other sets of numbers that can be associated with a spike train: taking into account the intervals between the first $k$ spikes, $k \geqslant 2$ (direct generalization of the previous idea), or between all intervals; only halting computations can be considered or arbitrary computations; an important difference is between the case when all intervals are considered and the case when the intervals are taken into account alternately (take the first interval, ignore the next one, take the third, and so on); the halting condition can be combined with the alternating style of defining the output.

The result of a computation can be defined also as usual in membrane computing, as the number of spikes present in the output neuron in the end of a computation − we have then to work with halting computations. It is also possible to consider SN P systems working in the recognizing mode: we designate a neuron as the input one, we start the computation from an initial configuration, and we introduce in the input neuron two spikes, in steps $t_1$ and $t_2$; the number $t_2 - t_1$ is recognized by the system if the computation eventually halts.

Then, the spike train itself can be considered as the result of a computation, codified as a string of bits: we write 1 for a step when the system outputs a spike and 0 otherwise. The halting computations will thus provide finite strings over the binary alphabet, the non-halting computations will produce infinite sequences of bits. If an input neuron is also provided, then a transducer is obtained, translating input binary strings into binary strings. Some of these possibilities will be illustrated below.

### 7.3 Three Examples

The first example, recalled from Ref. [22], is

$$\Pi_1 = (\{a\}, \sigma_1, \sigma_2, \sigma_s, syn, 3), \text{ with}$$
$$\sigma_1 = (2, \{a^2 / a \to a; 0,\ a \to \lambda\}),$$
$$\sigma_2 = (1, \{a \to a; 0,\ a \to a; 1\}),$$
$$\sigma_3 = (3, \{a^3 \to a; 0,\ a \to a; 1,\ a^2 \to \lambda\}),$$
$$syn = \{(1,2), (2,1), (1,3), (2,3)\}.$$

This system is given in a graphical form in Figure 4, following the standard way to pictorially represent a configuration of an SN P system, in particular, the initial configuration. Specifically, each neuron is represented by a "membrane" (an oval), marked with a label and having inside both the current number of spikes (written explicitly, in the form $a^n$ for $n$ spikes present in a neuron) and the evolution rules; the synapses linking the neurons are represented by arrows; besides the fact that the output neuron will be identified by its label, it is also suggestive to draw a short arrow that exits from it, pointing to the environment.

This system works as follows. All neurons can fire in the first step, with neuron $\sigma_2$ choosing non-deterministically between its two rules. Note that neuron $\sigma_1$ can fire only if it contains two spikes; one spike is consumed, the other remains available for the next step.

Both neurons $\sigma_1$ and $\sigma_2$ send a spike to the output neuron, $\sigma_3$; these two spikes are forgotten in the next step. Neurons $\sigma_1$ and $\sigma_2$ also exchange their spikes; thus, as long as neuron $\sigma_2$ uses the rule $a \to a$; 0, the first neuron receives one spike, thus completing the needed two spikes for firing again.

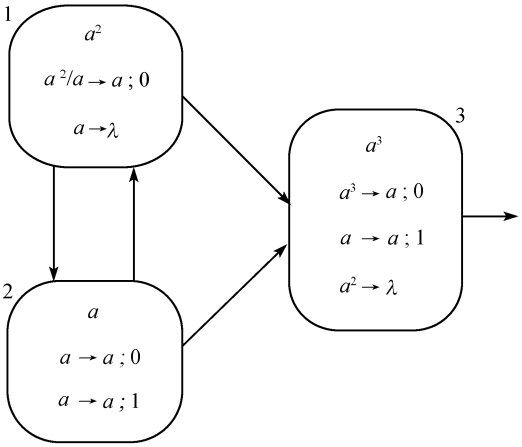


**Fig. 4** An SN P system generating all natural numbers greater than 1

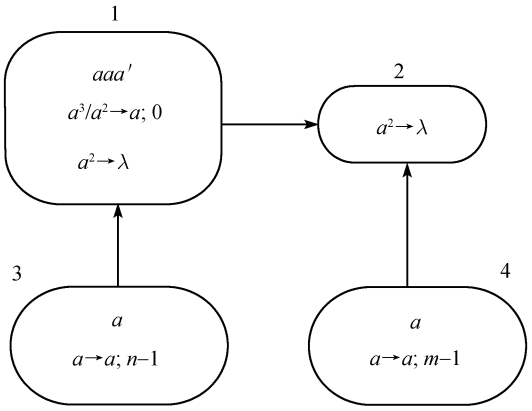


**Fig. 5** The initial configuration of system $\Pi_2$

However, at any moment, starting with the first step of the computation, neuron $\sigma_2$ can choose to use the rule $a \to a$; 1. On the one hand, this means that the spike of neuron $\sigma_1$ cannot enter neuron $\sigma_2$, it only goes to neuron $\sigma_3$; in this way, neuron $\sigma_2$ will never work again because it remains empty. On the other hand, in the next step neuron $\sigma_1$ has to use its forgetting rule $a \to \lambda$, while neuron $\sigma_3$ fires, using the rule $a \to a$; 1. Simultaneously, neuron $\sigma_2$ emits its spike, but it cannot enter neuron $\sigma_3$ (it is closed this moment); the spike enters neuron $\sigma_1$, 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 $\sigma_3$. Because of the waiting moment imposed by the rule $a \to a$; 1 from neuron $\sigma_3$, the two spikes of this neuron cannot be consecutive, but at least two steps must exist in between.

Thus, we conclude that (remember that number 0 is ignored) $N_2(\Pi_1) = \mathbf{N} - \{1\}$.

The second example is given in Figure 5, and it is meant to generate a language of traces: one spike is marked in the initial configuration (in the graphical representation, we prime one of the spikes); it is processed like any other spike, but its place in the system at the end of each computation step is recorded, and in this way we get a string. Specifically, if the marked spike is in neuron $\sigma_i$ at the end of a step, then we write the letter $b_i$. The marked spike can or cannot be consumed when applying a spiking rule that does not consume all spikes. If consumed, then the mark passes to one of the produced spikes (goes non-deterministically with one of the spikes sent to neurons linked to the neuron where the marked spike was before); if not consumed, the marked spike remains in the original neuron. If the marked spike is removed by a forgetting rule or goes to the environment, then the string is completed, even if the computation continues. However, in order to accept the string, the computation must eventually halt.

Let us examine the functioning of system $\Pi_2$ whose initial configuration is given in Fig. 5. It generates the language $T(\Pi_2) = \{b_1^n, b_2^m\}$, for given $n$, $m \geq 1$. In the first step, neuron $\sigma_1$ consumes or does not consume the marked spike, thus keeping it inside or sending it to neuron $\sigma_2$. One spike remains in neuron $\sigma_1$ and one is placed in neuron $\sigma_2$. Simultaneously, neurons $\sigma_3$ and $\sigma_4$ fire, and they spike after $n-1$ and $m-1$ steps, respectively. Thus, in steps $n$ and $m$, neurons $\sigma_1$ and $\sigma_2$, respectively, receive one more spike, which is forgotten in the next step together with the spike existing there.

Note that $n$ and $m$ can be equal or different.

The last example illustrates the following result from Ref. [27]: *Any function* $f : \{0,1\}^k \to \{0,1\}$ *can be computed by an SN P transducer with k input neurons (also using further* $2^k + 4$ *neurons, one being the output one*).

The idea of the proof of this result is suggested in Figure 6, where a system is presented, which computes the function $f : \{0,1\}^3 \to \{0,1\}$ defined by

$$f(b_1, b_2, b_3) = 1 \text{ iff } b_1 + b_2 + b_3 \neq 2.$$

The three input neurons, $\sigma_{in_1}, \sigma_{in_2}, \sigma_{in_3}$, are continuously fed with bits $b_1$, $b_2$, $b_3$, and the output neuron will provide, with a delay of 3 steps, the value of $f(b_1, b_2, b_3)$.

### 7.4 Some results

There are several parameters describing the complexity of an SN P system: number of neurons, number of rules, number of spikes consumed or forgotten by a rule, etc. Here we consider only some of them and we denote by $N_2SN\,P_m(rule_k, cons_p, forg_q)$ the family of all sets $N_2$ ($\Pi$) computed as specified in Section 7.2 by SN P systems with at most $m \geq 1$ neurons, using at most $k \geq 1$ rules in each neuron, with all spiking rules $E/a^r \to a$; $t$ having $r \leq p$, and all forgetting rules $a^s \to \lambda$ having $s \leq q$. When any parameter $m$, $k$, $p$, $q$ is not bounded, it is replaced with *. When we work only with SN P systems whose neurons contain at most $s$ spikes at any step of a computation (*finite* systems), then we add the parameter $bound_s$ after $forg_q$. (Corresponding families are defined for other definitions of the result of a computation, as well as for the accepting case, but the results are quite similar, hence we do not give details here.)

By *NFIN*, *NREG*, *NRE* we denote the families of finite, semilinear, and Turing computable sets of (positive) natural numbers (number 0 is ignored); they correspond to the

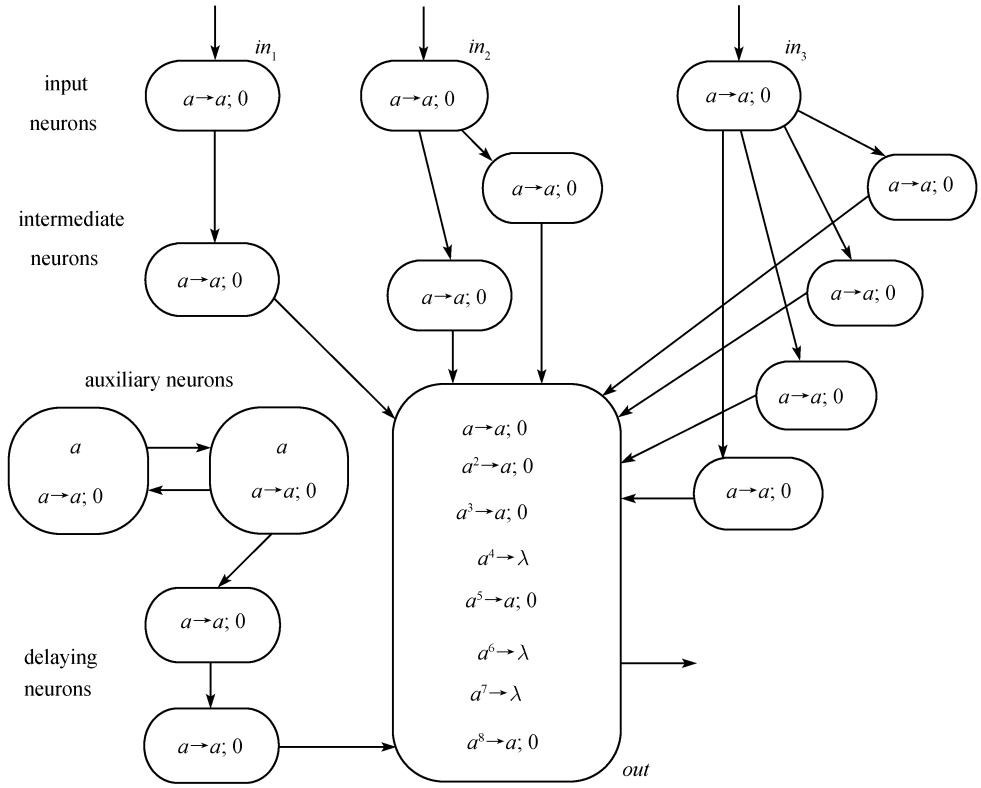


**Fig. 6** An SN P transducer computing a Boolean function of three variables

length sets of finite, regular, and recursively enumerable languages, whose families are denoted by *FIN*, *REG*, *RE*.

The following results were proved in Ref. [22] and extended in Ref. [26] to other ways of defining the output of a computation.

**Theorem 7.1** (i) $NFIN = N_2SNP_1\,(rule_*, cons_1, forg_0) = N_2SNP_2\,(rule_*, cons_*, forg_*)$.

(ii) $N_2SNP_*\,(rule_k, cons_p, forg_q) = NRE$ *for all* $k \geqslant 2$, $p \geqslant 3$, $q \geqslant 3$.

(iii) $NSLIN = N_2SNP_*\,(rule_k, cons_p, forg_q, bound_s)$, *for all* $k \geqslant 3$, $q \geqslant 3$, $p \geqslant 3$, *and* $s \geqslant 3$.

Point (ii) was proved in Ref. [22] also for the accepting case, and then the systems used can be required to be deterministic (at most one rule can be applied in each neuron in each step of the computation).

As mentioned above, the SN P systems can also be used for generating languages. In the standard definition, they can only generate strings over the binary alphabet, $B = \{0,1\}$. As proved in Ref. [34], in this setup the language generating power of SN P systems is rather eccentric; on the one hand, finite languages (like $\{0, 1\}$) cannot be generated, on the other hand, we can represent any RE language as the direct morphic image of an inverse morphic image of alanguage generated in this way. This eccentricity is due mainly to the restricted way of generating strings, with one symbol added in each computation step. This restriction does not appear in the case of extended spiking rules. In this case, a language can be generated by associating the symbol $b_i$ with a step when the output neuron sends out $i$ spikes, with an important decision to take in the case $i = 0$: we can either consider $b_0$ as a separate symbol, or we can assume that emitting 0 spikes means inserting $\lambda$ in the generated string. Thus, we both obtain strings over arbitrary alphabets, not only over the binary one, and, in the case where we ignore the steps when no spike is emitted, a considerable freedom is obtained in the way the computation proceeds. This latter variant (with $\lambda$ associated with steps when no spike exits the system) is considered below.

We denote by $LSN^e P_m\,(rule_k, cons_p, prod_q)$ the family of languages $L(\Pi)$, generated by *SN P* systems $\Pi$ using extended rules, with at most $m$ neurons, each neuron having at most $k$ rules, each rule consuming at most $p$ spikes and producing at most $q$ spikes. Again, the parameters $m$, $k$, $p$, $q$ are replaced by * if they are not bounded.

The next counterparts of the results from Ref. [34] were proved in Ref. [28].

**Theorem 7.2** (i) $FIN = LSN^e P_1\,(rule_*, cons_*, prod_*)$ *and this result is sharp in the sense that* $LSN^e P_2\,(rule_2, cons_2, prod_2)$ *contains infinite languages*.

(ii) $LSNe\ P_2(rule_*, cons_*, prod_*) \subseteq REG \subset LSN^e P_3(rule_*, cons_*, prod_*)$; *the second inclusion is proper*, *because* $LSN^e P_3(rule_3, cons_4, prod_2)$ *contains non-regular languages*; *actually*, *the family* $LSN^e P_3(rule_3, cons_6, prod_4)$ *contains non-semilinear languages*.

(iii) $RE = LSN^e P_*(rule_*, cons_*, prod_*)$.

It is an open problem to find characterizations or representations in this framework for families of languages in the Chomsky hierarchy different from *FIN*, *REG*, *RE*.

There are many other research topics in this area. We close this presentation by mentioning an important issue,

which is related to the *efficiency* of SN P systems, the possibility of solving computationally hard problems in a feasible (polynomial) time. This is usually achieved in membrane computing by means of tools that allow producing an exponential working space in a linear time; 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 Ref. [35] one proposes a way 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 spikes in certain neurons (in a polynomially bounded number of neurons a polynomially bounded number of spikes are introduced); then, by moving spikes along synapses, the system self-activates, and a specific output provides the answer to the problem. This was illustrated in Ref. [35] for SAT: an SN P system is constructed, depending on SAT, and it is initiated for a given instance $\gamma$ by introducing spikes in $2nm$ neurons, where $n$ is the number of variables and $m$ is the number of clauses of $\gamma$; in four steps, the system decides whether or not $\gamma$ is satisfiable.

This way of solving problems, by activating a pre-computed resource, is not at all usual in computability, and we know no formalization of this approach; in particular, we know no complexity classes defined in this framework. However, we believe that this is a research direction worth exploring, with a good motivation in bio-inspired computing.

## 8 Closing remark

This overview was a quick one, meant only to let the reader have a general idea about membrane computing in general and spiking neural P systems in particular. As suggested in the introduction, there are several other directions of research that were not mentioned above. The interested reader should consult the bibliography mentioned below, especially the papers available at Ref. [2].

**Acknowledgements** This work was supported by the project BioMAT 2-CEx06- 11-97/19.09.06. Thanks are due to an anonymous referee for a series of comment on a first version of this paper.

## References

1. Păun Gh. Computing with membranes. Journal of Computer and System Sciences, 2000, 61(1): 108−143 (and Turku Center for Computer Science-TUCS Report 208, 1998 www.tucs.fi)
2. The P Systems Web Page: http://psystems.disco.unimib.it, with mirrors at http://bmc.hust.edu.cn/psystems or http://bmchust.3322.org/psystems.
3. Păun Gh. Membrane Computing. An Introduction. Berlin: Springer, 2002
4. Ciobanu G, Păun Gh, Pérez-Jiménez M J, eds. Applications of Membrane Computing. Berlin: Springer, 2006
5. Alberts B, Johnson A, Lewis J, et al. Molecular Biology of the Cell. 4th ed. New York: Garland Science, 2002
6. Freund R., Kari L., Oswald M., et al. Computationally universal P systems without priorities: two catalysts are sufficient. Theoretical Computer Sci, 2005, 330(2): 251−266
7. Alhazov A, Freund R, Rogozhin Y. Computational power of symport/antiport: history, advances, and open problems. In: Freund R, Păun Gh, Rozenberg G, et al, eds. Membrane Computing. 6th International Workshop, Austria, 2005, Revised Selected and Invited Papers. LNCS 3859, Berlin: Springer, 2006, 44−78
8. Păun Gh, Pazos J, Pérez-Jiménez M J, et al. Symport/antiport P systems with three objects are universal. Fundamenta Informaticae. 2005, 64(1−4): 353−367
9. Freund R, Păun Gh, Rozenberg G, et al, eds. Membrane Computing. 6th International Workshop, WMC6, Vienna, Austria, July 2005, Revised Selected and Invited Papers. LNCS 3859, Berlin: Springer, 2006
10. Hoogeboom H J, Păun Gh, Rozenberg G, et al, eds. Membrane Computing, International Workshop, Leiden, The Netherlands, 2006, Selected and Invited Papers. LNCS 4361, Berlin: Springer, 2007
11. Ibarra O H, Yen H C. Deterministic catalytic systems are not universal. Theoretical Computer Sci. 2006, 363: 149−161
12. Ibarra O H. The number of membranes matters. In: Martín-Vide C, Mauri G, Păun Gh, et al, eds. Membrane Computing. International Workshop, Spain, Revised Papers. LNCS 2933. Berlin: Springer, 2004, 218−231
13. Pérez-Jiménez M J, Romero-Jiménez A, Sancho-Caparrini F. Teoria de la complejidad en modelos de computacion celular con membranas. Sevilla:Kronos Editorial, 2002
14. Gutiérrez-Naranjo M A, P\u aun Gh, Pérez-Jiménez M J, eds. Cellular Computing. Complexity Aspects. Sevilla:Fenix Editora, 2005
15. The Sheffield Applications Web Page: http://www.dcs.shef.ac.uk/~marian/PSimulatorWeb/P_Systems_applications.htm
16. Nishida T Y. An application of P systems: A new algorithm for NP-complete optimization problems. In: Callaos N, et al, eds. Proceedings of the 8th World Multi-Conference on Systems, Cybernetics and Informatics, 2004, 5:109−112
17. Nishida T Y. Membrane algorithms: Approximate algorithms for NP-complete optimization problems. In: Ciobanu G, Păun Gh, Pérez-Jiménez M J, eds. Applications of Membrane Computing. Berlin: Springer, 2006, 301−312
18. Huang L, He X X, Wang N, et al. P systems based multi-objective optimization algorithm. In: Pre-proceedings of BIC-TA 2006. Volume of Membrane Computing Section, 113−123
19. Huang L, Wang N. An optimization algorithms inspired by membrane computing. In: Jiao L, et al, eds. Proceeding of ICNC 2006. LNCS 4222. Springer, 2006, 49−55
20. Leporati A, Pagani D. A membrane algorithm for the min storage problem. In: Hoogeboom H J, Păun Gh, Rozenberg G, et al, eds. Membrane Computing, International Workshop, Netherlands, 2006, Selected and Invited Papers. LNCS 4361, Berlin: Springer, 2007, 443−462

21. Zaharie D, Ciobanu G. Distributed evolutionary algorithms inspired by membranes in solving continuous optimization problems. In: Hoogeboom H J, Păun Gh, Rozenberg G, et al, eds. Membrane Computing, International Workshop, Netherlands, 2006, Selected and Invited Papers. LNCS 4361, Berlin: Springer, 2007, 536−554
22. Ionescu M, Păun Gh, Yokomori T. Spiking neural P systems. *Fundamenta Informaticae*. 2006, 71(2−3): 279−308
23. Gerstner W, Kistler W. Spiking Neuron Models. Single Neurons, Populations, Plasticity. Cambridge Univ. Press, 2002
24. Maass W. Computing with spikes. Special Issue on Foundations of Information Processing of TELEMATIK, 2002, 8(1): 32−36
25. Cavaliere M, Freund R, Leitsch A, et al. Event-related outputs of computations in P systems. In: Proceeding of Third Brainstorming Week on Membrane Computing. RGNC Report. 2005, 107−122
26. Păun Gh, Pérez-Jiménez M J, Rozenberg G. Spike trains in spiking neural P systems. Intern. J. Found. Computer Sci. 2006, 17(4): 975−1002
27. Păun Gh, Pérez-Jiménez M J, Rozenberg G. Infinite spike trains in spiking neural P systems. 2006 (Submitted)
28. Chen H, Ishdorj T O, Păun Gh, et al. Spiking neural P systems with extended rules. In: Gutiérrez-Naranjo M A, et al, eds. Proceedings of Fourth Brainstorming Week on Membrane Computing, Sevilla, 2006, Vol I, 241−265
29. Ibarra O H, Woodworth S, Yu F, et al. On spiking neural P systems and partially blind counter machines. In:Proceedings of Fifth Unconventional Computation Conference, UK, 2006
30. Ibarra O H, Woodworth S. Characterizations of some restricted spiking neural P systems. In: Hoogeboom H J, Păun Gh, Rozenberg G, et al, eds. Membrane Computing, International Workshop, Netherlands, 2006, Selected and Invited Papers. LNCS 4361, Berlin: Springer, 2007, 424−442
31. Korec I. Small universal register machines. Theoretical Computer Science. 1996, 168: 267−301
32. Păun A, Păun Gh. Small universal spiking neural P systems. In: Gutiérrez-Naranjo M A, et al, eds. Proceedings of Fourth Brainstorming Week on Membrane Computing, Sevilla, 2006, Vol II, 213−234; and BioSystems (In press)
33. Ibarra O H, Păun A, Păun Gh, et al. Normal forms for spiking neural P systems. In: Gutiérrez-Naranjo M A, et al, eds. Proceedings of Fourth Brainstorming Week on Membrane Computing, Sevilla, 2006, Vol II, 105−136; and Theoretical Computer Sci.(In press)
34. Chen H, Freund R, Ionescu M, et al. On string languages generated by spiking neural P systems. In: Gutiérrez-Naranjo M A, et al, eds. Proceedings of Fourth Brainstorming Week on Membrane Computing, Sevilla, 2006, Vol I, 169−194
35. Chen H, Ionescu M, Ishdorj T O. On the efficiency of spiking neural P systems. In: Gutiérrez-Naranjo M A, et al, eds. Proceedings of Fourth Brainstorming Week on Membrane Computing, Sevilla, 2006, Vol I, 195−206; and In: Proceedings of 8th Intern. Conf. on Electronics, Information, and Communication, Mongolia, 2006, 49−52