Solving Numerical NP-Complete Problems with Spiking Neural P Systems

Alberto Leporati, Claudio Zandron
Claudio Ferretti, Giancarlo Mauri

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy

{leporati,zandron,ferretti,mauri}@disco.unimib.it

Summary. Starting from an extended nondeterministic spiking neural P system that solves the Subset Sum problem in a constant number of steps, recently proposed in a previous paper, we investigate how different properties of spiking neural P systems affect the capability to solve numerical **NP**–complete problems. In particular, we show that by using maximal parallelism we can convert any given integer number from the usual binary notation to the unary form, and thus we can initialize the above P system with the required (exponential) number of spikes in polynomial time. On the other hand, we show that this conversion cannot be performed in polynomial time if the use of maximal parallelism is forbidden. Finally, we show that by *selectively* using nondeterminism and maximal parallelism (that is, for each neuron in the system we can specify whether it works in deterministic or nondeterministic way, as well as in sequential or maximally parallel way) there exists a *uniform* spiking neural P system that solves all the instances of Subset Sum of a given size.

1 Introduction

Membrane systems (also called *P systems*) were introduced in [16] as a new class of distributed and parallel computing devices, inspired by the structure and functioning of living cells. The basic model consists of a hierarchical structure composed by several membranes, embedded into a main membrane called the *skin*. Membranes divide the Euclidean space into *regions*, that contain some *objects* (represented by symbols of an alphabet) and *evolution rules*. Using these rules, the objects may evolve and/or move from a region to a neighboring one. Usually, the rules are applied in a nondeterministic and maximally parallel way; moreover, all the objects that may evolve are forced to evolve. A *computation* starts from an initial configuration of the system and terminates when no evolution rule can be applied. The result of a computation is the multiset of objects contained into an *output*

membrane, or emitted to the environment from the skin of the system. For a systematic introduction to P systems we refer the reader to [18], whereas the latest information can be found in [23].

In an attempt to pass from cell-like to tissue-like architectures, in [14] *tissue P systems* were defined, in which cells are placed in the nodes of a (directed) graph. Since then, this model has been further elaborated, for example, in [4] and [21], with recent results about both theoretical properties [1] and applications [15]. This evolution has led to explore also *neural-like* architectures, yielding to the introduction of *spiking neural P systems* (SN P systems, for short) [8], based on the neurophysiological behavior of neurons sending electrical impulses (*spikes*) along axons to other neurons. We recall that this biological background has already led to several models in the area of neural computation, e.g., see [12, 13, 6].

Similarly to tissue P systems, in SN P systems the cells (neurons) are placed in the nodes of a directed graph, called the *synapse graph*. The contents of each neuron consist of a number of copies of a single object type, called the *spike*. The *firing rules* assigned to a cell allow a neuron to send information to other neurons in the form of electrical impulses (also called spikes) which are accumulated at the target cell. The application of the rules depends on the contents of the neuron; in the general case, applicability is determined by checking the contents of the neuron against a regular set associated with the rule. As inspired from biology, when a cell sends out spikes it becomes "closed" (inactive) for a specified period of time, that reflects the refractory period of biological neurons. During this period, the neuron does not accept new inputs and cannot "fire" (that is, emit spikes). Another important feature of biological neurons is that the length of the axon may cause a time delay before a spike arrives at the target. In SN P systems this delay is modeled by associating a delay parameter to each rule which occurs in the system. If no firing rule can be applied in a neuron, there may be the possibility to apply a *forgetting rule*, that removes from the neuron a predefined number of spikes.

In the original model of SN P systems defined in [8], computations occur as follows. A *configuration* specifies, for each neuron of the system, the number of spikes it contains and the number of computation steps after which the neuron will become "open" (that is, not closed). Starting from an initial configuration, a positive integer number is given in input to a specified *input neuron*. The number is encoded as the interval of time steps elapsed between the insertion of two spikes into the neuron (note that this is a unary encoding). To pass from a configuration to another one, for each neuron a rule is chosen among the set of applicable rules, and is executed. The computation proceeds in a sequential way into each neuron, and in parallel among different neurons. Generally, a computation may not halt. However, in any case the output of the system is considered to be the time elapsed between the arrival of two spikes in a designated *output cell*. Defined in this way, SN P systems compute functions of the kind $f : \mathbb{N} \to \mathbb{N}$ (they can also indirectly compute functions of the kind $f : \mathbb{N}^k \to \mathbb{N}$ by using a bijection from $\mathbb{N}^k$ to $\mathbb{N}$). By neglecting the output neuron we can define *accepting* SN P systems, in which the

natural number given in input is accepted if the computation halts. On the other hand, by ignoring the input neuron (and thus starting from a predefined input configuration) we can define *generative* SN P systems.

In [8] it was shown that generative SN P systems are universal, that is, can generate any recursively enumerable set of natural numbers. Moreover, a characterization of semilinear sets was obtained by spiking neural P systems with a bounded number of spikes in the neurons. These results can also be obtained with even more restricted forms of spiking P systems; for example, [7] shows that at least one of these features can be avoided while keeping universality: time delay (refractory period) greater than 0, forgetting rules, outdegree of the synapse graph greater than 2, and regular expressions of complex form. Finally, in [19] the behavior of spiking neural P systems on infinite strings and the generation of infinite sequences of 0 and 1 was investigated, whereas in [2] spiking neural P systems were studied as language generators (over the binary alphabet $\{0, 1\}$).

In [10] we have shown that by slightly extending the original definition of SN P system given in [8] and [9], it is possible to solve any given instance of Subset Sum by using a nondeterministic (extended) SN P system. The solution is given in the so called *semi–uniform* setting, that is, for every fixed instance of Subset Sum a specific SN P system that solves it is built. In particular, the rules of the system and the number of spikes which occur in the initial configuration depend upon the instance to be solved. A drawback of this solution is that in general the number of spikes needed to initialize the system is exponential with respect to the usually agreed instance size of Subset Sum. However, in this paper we show that this preparation can be performed in polynomial time by traditional SN P systems that, endowed with the power of maximal parallelism, read from the environment the k-bit integer numbers $v_1, v_2, \ldots, v_n$ encoded in binary and produce $v_1, v_2, \ldots, v_n$ spikes, respectively, in n specified neurons. We also prove that this operation cannot be performed in polynomial time if the use of maximal parallelism is forbidden. Then we design an SN P system that performs the opposite conversion: it takes a given (k-bit) number N of spikes occurring in a certain neuron, and produces the coefficients of the binary encoding of N in k predefined neurons. Thanks to these two modules, that allow us to move from binary to unary encoding and back, we finally design a *uniform* family $\{\Pi(\langle n, k\rangle\}_{n,k\in\mathbb{N}}$ of SN P systems, where $\Pi(\langle n, k\rangle)$ solves all possible instances $(\{v_1, v_2, \ldots, v_n\}, S)$ of Subset Sum such that all v_i and S are k-bit positive integer numbers. As we will see, the construction of $\Pi(\langle n, k\rangle)$ relies upon the assumption that different subsystems can work under different regimes: deterministic vs. nondeterministic, and sequential vs. maximally parallel.

The rest of this paper is organized as follows. In section 2 we give some mathematical preliminaries, and we define the standard version of SN P systems (as found in [9]) as well as a slightly extended version. In section 3 we recall from [10] how the **NP**–complete problem Subset Sum can be solved in constant time by exploiting nondeterminism in our extended SN P systems. In section 4 we convert positive integer numbers from binary notation to the unary form through max-

imally parallel SN P systems, and we use such a convertion as an initialization stage to solve Subset Sum. In section 5 we perform also the opposite conversion, and we design a family of SN P systems that solves Subset Sum in a uniform way (according to the above definition). Section 6 concludes the paper and gives some directions for future research.

2 Preliminaries

Let us start by recalling the standard definition of a spiking neural P system, taken from [9]. A *spiking neural membrane system* (SN P system, for short), of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are *neurons*, of the form $\sigma_i = (n_i, R_i)$, with $1 \leq i \leq m$, where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained in σ_i;
   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 over a, and $c \geq 1$, $d \geq 0$ are integer numbers; if $E = a^c$, then it is usually written in the following simplified form: $a^c \to a; d$;
      (2) $a^s \to \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \to a; d$ of type (1) from R_i, we have $a^s \notin L(E)$ (where $L(E)$ denotes the regular language defined by E);
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, is the directed graph of *synapses* between neurons;
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons of Π.

The rules of type (1) are called *firing* (also *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains $k \geq c$ spikes, and $a^k \in L(E)$, then the rule $E/a^c \to a; d \in R_i$ can be applied. The execution of this rule removes c spikes from σ_i (thus leaving $k - c$ spikes), and prepares one spike to be delivered to all the neurons σ_j such that $(i, j) \in syn$. If $d = 0$, then the spike is immediately emitted, otherwise it is emitted after d computation steps of the system. (Observe that, as usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized.) If the rule is used in step t and $d \geq 1$, then in steps $t, t+1, t+2, \ldots, t+d-1$ the neuron is *closed*, so that it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then that particular spike is lost), and cannot fire new rules. In the step $t + d$, the neuron spikes and becomes open again, so that it can receive spikes (which can be used starting with the step $t + d + 1$) and select rules to be fired.

Rules of type (2) are called *forgetting* rules, and are applied as follows: if the neuron σ_i contains *exactly* s spikes, then the rule $a^s \to \lambda$ from R_i can be used, meaning that all s spikes are removed from σ_i. Note that, by definition, if a firing rule is applicable then no forgetting rule is applicable, and vice versa.

In each time unit, if a neuron σ_i can use one of its rules, then a rule from R_i must be used. Since two firing rules, $E_1 : a^{c_1} \to a; d_1$ and $E_2 : a^{c_1} \to a; d_2$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron. In such a case, only one of them is chosen nondeterministically. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other.

The *initial configuration* of the system is described by the numbers $n_1, n_2, \ldots, n_m$ of spikes present in each neuron, with all neurons being open. During the computation, a configuration is described by both the number of spikes present in each neuron and by the state of each neuron, which can be expressed as the number of steps to count down until it becomes open (this number is zero if the neuron is already open). A *computation* in a system as above starts in the initial configuration. In order to compute a function $f : \mathbb{N}^k \to \mathbb{N}$, one possibility is to introduce k natural numbers $n_1, n_2, \ldots, n_k$ in the system by "reading" from the environment a binary sequence $z = 0^b 10^{n_1} 10^{n_2} 1 \ldots 10^{n_k} 10^g$, for some $b, g \geq 0$; this means that the input neuron of Π receives a spike in each step corresponding to a digit 1 from the string z. Note that we input exactly $k + 1$ spikes. The result of the computation is also encoded in the distance between two spikes: we impose to the system to output exactly two spikes and halt (sometimes after the second spike) hence producing a spike train of the form $0^{b'} 10^r 10^{g'}$, for some $b', g' \geq 0$ and with $r = f(n_1, n_2, \ldots, n_k)$. As discussed in [9], there are other possibilities to encode natural numbers read from and/or emitted to the environment by SN P systems; for example, we can consider the number of spikes arriving to the input neuron and leaving from the output neuron, respectively, or the number of spikes read/produced in a given interval of time.

If we do not specify an input neuron (hence no input is taken from the environment) then we use SN P systems in the *generative* mode; we start from the initial configuration, and the distance between the first two spikes of the output neuron (or the number of spikes, etc.) is the result of the computation. Note that generative SN P systems are inherently nondeterministic, otherwise they would always reproduce the same sequence of computation steps, and hence the same output. Dually, we can neglect the output neuron and use SN P systems in the *accepting* mode; for $k \geq 1$, the natural number $n_1, n_2, \ldots, n_k$ are read in input and, if the computation halts, then the numbers are accepted.

We define the *description size* of an SN P system Π as the number of bits which are necessary to describe it. Since the alphabet O is fixed, no bits are necessary to define it. In order to represent *syn* we need at most m^2 bits, whereas we can represent the values of *in* and *out* by using $\log m$ bits each. For every neuron σ_i we have to specify a natural number n_i and a set R_i of rules. For each rule we need to specify its type (firing or forgetting), which can be done with 1 bit,

and in the worst case we have to specify a regular expression and two natural numbers. If we denote by N the maximum natural number that appears in the definition of Π, R the maximum number of rules which occur in its neurons, and S the maximum size required by the regular expressions that occur in Π (more on this later), then we need a maximum of $\log N + R(1 + S + 2\log N)$ bits to describe every neuron of Π. Hence, to describe Π we need a total of $m^2 + 2\log m + m\big(\log N + R(1+S+2\log N)\big)$ bits. Note that this quantity is polynomial with respect to m, R, S and $\log N$. Since the regular languages determined by the regular expressions that occur in the system are *unary* languages, the strings of such languages can be bijectively identified with their lengths. Hence, when writing the regular expression E, instead of writing unions, concatenations and Kleene closures among strings we can do the same by using the lengths of such strings. (Note that, when concatenating two languages L_1 and L_2 represented in this way, the lengths in L_1 are summed with the lengths of L_2 by combining them in all possible ways). In this way we obtain a representation of E which is *succint*, that is, exponentially more compact than the usual representation of regular expressions. As we have seen in [10], this succint representation yields some difficulties when we try to simulate a deterministic accepting SN P system that contains general regular expressions, by a deterministic Turing machine. However, as shown in [7], it is possible to restrict our attention to particularly simple regular expressions, without loosing computational completeness. For these expressions, the membership problem (is a given string into the language generated by the regular expression?) is polynomial also when representing the instances in succint form, and thus they do not yield problems when simulating the system with a deterministic Turing machine.

In what follows it will be convenient to consider also the following slightly extended version of SN P systems. Precisely, we will allow rules of the type $E/a^c \to a^p; d$, where $c \geq 1$, $p \geq 0$ and $d \geq 0$ are integer numbers. The semantics of this kind of rules is as follows: if the contents of the neuron matches the regular expression E, then the rule can be applied. When the rule is applied, c spikes are removed from the contents of the neuron and p spikes are prepared to be delivered to all the neurons which are directly connected (through an arc of *syn*) with the current neuron. If $d = 0$, then these p spikes are immediately sent, otherwise the neuron becomes closed for the next d computation steps, after which the p spikes will be sent. As before, a closed neuron does not receive spikes from other neurons, and does not apply any rule. If $p = 0$, then we obtain a forgetting rule as a particular case of our general rules.

Also in the extended SN P systems it may happen that, given two rules $E_1/a^{c_1} \to a^{p_1}; d_1$ and $E_2/a^{c_2} \to a^{p_2}; d_2$, if $L(E_1) \cap L(E_2) \neq \emptyset$ then for some contents of the neuron both the rules can be applied. In such a case, one of them is nondeterministically chosen. Note that we do not require that forgetting rules are applied only when no firing rule can be applied. We say that the system is *deterministic* if, for every neuron that occurs in the system, any two rules $E_1/a^{c_1} \to a^{p_1}; d_1$ and $E_2/a^{c_2} \to a^{p_2}; d_2$ in the neuron are such that $L(E_1) \cap L(E_2) = \emptyset$. This means

that, for any possible contents of the neuron, at most one of the rules that occur in the neuron may be applied.

By using an *input neuron* and an *output neuron*, we have SN P systems that compute functions of the kind $f : \mathbb{N} \to \mathbb{N}$ (as well as functions of the kind $f : \mathbb{N}^k \to \mathbb{N}$, by appropriate bijections between $\mathbb{N}^k$ and $\mathbb{N}$), and hence we cover both the generative and the accepting cases. If $out = 0$, then it is understood that the output is sent to the environment (as the number of spikes produced by the system, as the distance between the first two spikes, etc.). As usual, to use an SN P system in the generative mode we do not consider the input neuron, whereas by ignoring the output neuron we obtain an accepting SN P system.

The *description size* of an extended SN P system is defined exactly as we have done for standard systems, the only difference being that now we require (at most) three natural numbers to describe a rule.

3 Solving Numerical NP–complete Problems with Extended Spiking Neural P Systems

Let us start by recalling the nondeterministic extended SN P system introduced in [10] to solve the **NP**–complete problem Subset Sum in a *constant* number of computation steps. The Subset Sum problem can be defined as follows.

Problem 1. Name: Subset Sum.

- Instance: a (multi)set $V = \{v_1, v_2, \ldots, v_n\}$ of positive integer numbers, and a positive integer number S.
- Question: is there a sub(multi)set $B \subseteq V$ such that $\sum_{b \in B} b = S$?

If we allow to nondeterministically choose among the rules which occur in the neurons, then the extended SN P system depicted in Figure 1 solves any given instance of Subset Sum in a constant number of steps. We emphasize the fact that such a solution occurs in the *semi-uniform* setting, that is, for every instance of Subset Sum we build an SN P system that specifically solves that instance.

Let $(V = \{v_1, v_2, \ldots, v_n\}, S)$ be the instance of Subset Sum to be solved. In the initial configuration of the system, the leftmost neurons contain (from top to bottom) $v_1, v_2, \ldots, v_n$ spikes, respectively, whereas the rightmost neurons contain zero spikes each. In the first step of computation, in each of the leftmost neurons of the SN P system depicted in Figure 1 it is nondeterministically chosen whether to include or not the element v_i in the (candidate) solution $B \subseteq V$; this is accomplished by nondeterministically choosing among one rule that forgets v_i spikes (in such a case, $v_i \notin B$) and one rule that propagates v_i spikes to the rightmost neurons. At the beginning of the second step of computation a certain number $N = |B|$ of spikes, that corresponds to the sum of the v_i which have been chosen, occurs in the rightmost neurons. We have three possible cases:

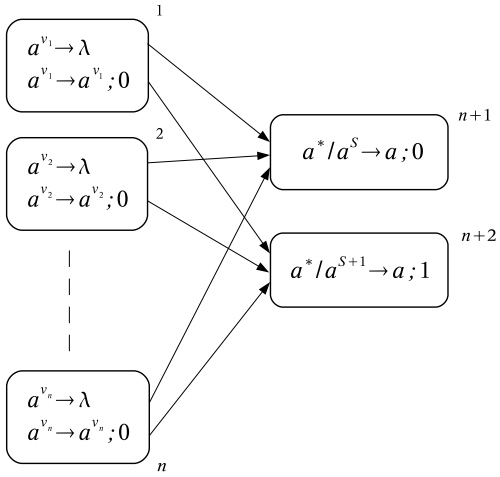

Fig. 1. A nondeterministic extended SN P system that solves the Subset Sum problem in constant time

- $N < S$: in this case neither the rule $a^*/a^S \to a; 0$ nor the rule $a^*/a^{S+1} \to a; 1$ (which occur in the neuron at the top and at the bottom of the second layer, respectively) fire, and thus no spike is emitted to the environment;
- $N = S$: only the rule $a^*/a^S \to a; 0$ fires, and emits a single spike to the environmnent. No further spikes are emitted;
- $N > S$: both the rules $a^*/a^S \to a; 0$ and $a^*/a^{S+1} \to a; 1$ fire. The first rule immediately sends one spike to the environment, whereas the second rule sends another spike at the next computation step (due to the delay associated with the rule).

Hence, by counting the number of spikes emitted to the environment at the second and third computation steps we are able to read the solution of the given instance of Subset Sum: the instance is positive if and only if a single spike is emitted.

The proposed system is generative; its input (the instance of Subset Sum to be solved) is encoded in the initial configuration. We stress once again that the ability to solve Subset Sum in constant time derives from the fact that the system is nondeterministic. As it happens with Turing machines, nondeterminism can be interpreted in two ways: (1) the system "magically" chooses the correct values v_i (if they exist) that allow to produce a single spike in output, or (2) at least one of the possible computations produces a single spike in output.

The formal definition of the extended (generative) SN P system depicted in Figure 1 is as follows:

$$\Pi = (\{a\}, \sigma_1, \ldots, \sigma_{n+2}, syn, out)\,,$$

where:

- $\sigma_i = (v_i, \{a^{v_i} \to \lambda,\ a^{v_i} \to a^{v_i}; 0\})$ for all $i \in \{1, 2, \ldots, n\}$;
- $\sigma_{n+1} = (0, \{a^*/a^S \to a; 0\})$;
- $\sigma_{n+2} = (0, \{a^*/a^{S+1} \to a; 1)$;
- $syn = \bigcup_{i=1}^{n} \{(i, n+1), (i, n+2)\}$;
- $out = 0$ indicates that the output is sent to the environment.

However, here we are faced with a problem that we have already met in [11], and that we will meet again in the rest of the paper. In order to clearly expose the problem, let us consider the following algorithm that solves Subset Sum using the well known Dynamic Programming technique [3]. In particular, the algorithm returns 1 on positive instances, and 0 on negative instances.

```
SUBSET SUM({v_1, v_2, ..., v_n}, S)
for j ← 0 to S
    do M[1, j] ← 0
M[1, 0] ← M[1, v_1] ← 1
for i ← 2 to n
     do for j ← 0 to S
            do M[i, j] ← M[i − 1, j]
               if j ≥ v_i and M[i − 1, j − v_i] > M[i, j]
                  then M[i, j] ← M[i − 1, j − v_i]
return M[n, S]
```

In order to look for a subset $B \subseteq V$ such that $\sum_{b \in B} b = S$, the algorithm uses an $n \times (S+1)$ matrix M whose entries are from $\{0, 1\}$. It fills the matrix by rows, starting from the first row. Each row is filled from left to right. The entry $M[i, j]$ is filled with 1 if and only if there exists a subset of $\{v_1, v_2, \ldots, v_i\}$ whose elements sum up to j. The given instance of Subset Sum is thus a positive instance if and only if $M[n, S] = 1$ at the end of the execution.

Since each entry is considered exactly once to determine its value, the time complexity of the algorithm is proportional to $n(S+1) = \Theta(nS)$. This means that the difficulty of the problem depends on the value of S, as well as on the magnitude of the values in V. In fact, let $K = max\{v_1, v_2, \ldots, v_n, S\}$. If K is polynomially bounded with respect to n, then the above algorithm works in polynomial time. On the other hand, if K is exponential with respect to n, say $K = 2^n$, then the above algorithm may work in exponential time and space. This behavior is usually referred to in the literature by telling that Subset Sum is a *pseudo–polynomial* **NP**–complete problem.

The fact that in general the running time of the above algorithm is not polynomial can be immediately understood by comparing its time complexity with the instance size. The usual size for the instances of Subset Sum is $\Theta(n \log K)$, since for conciseness every "reasonable" encoding is assumed to represent each element of V (as well as S) using a string whose length is $O(\log K)$. Here all logarithms are taken with base 2. Stated differently, the size of the instance is usually considered to be the number of bits which must be used to represent in binary S and all the

integer numbers which occur in V. If we would represent such numbers using the unary notation, then the size of the instance would be $\Theta(nK)$. But in this case we could write a program which first converts the instance in binary form and then uses the above algorithm to solve the problem in polynomial time with respect to the new instance size. We can thus conclude that the difficulty of a numerical **NP**–complete problem depends also on the measure of the instance size we adopt.

The problem we mentioned above about the SN P system depicted in Figure 1 is that the rules $a^{v_i} \to \lambda$ and $a^{v_i} \to a^{v_i}; 0$ which occur in the leftmost neurons, as well as those that occur in the rightmost neurons, check for the existence of a number of spikes which may be exponential with respect to the usually agreed instance size of Subset Sum. Moreover, to initialize the system the user has to place a number of objects which may also be exponential. This is not fair, because it means that the SN P system that solves the **NP**–complete problem has in general an exponential size with respect to the binary string which is used to describe it; an exponential effort is thus needed to build and initialize the system, that easily solves the problem by working in unary notation (hence in polynomial time with respect to the size of the system, but not with respect to its *description size*). This problem is in some aspects similar to what has been described in [11], concerning traditional P systems that solve **NP**–complete problems.

4 Solving Subset Sum with Inputs Encoded in Binary

Similarly to what we have done in [11], in this section we show that the ability of the SN P system depicted in Figure 1 to solve Subset Sum does not derive from the fact that the system is initialized with an exponential number of spikes, at least if we allow the application of rules in the maximal parallel way.

In this paper, maximal parallelism is intended exactly as in traditional P systems. Since in SN P systems we have only one kind of objects (the spike), this means that at every computation step the (multi)set of rules to be applied in a neuron is determined as follows. Let k denote the number of spikes contained in the neuron. First, one rule is nondeterministically chosen among those which can be applied. If such a rule consumes c spikes, then the selection process is repeated to the remaining $k - c$ spikes, until no rule can be applied. Note that a rule may eventually be chosen many times, and thus at the end of the process we obtain a multiset of rules. However let us note that, for our purposes, it will suffice to define maximally parallel neurons that contain just one rule. Hence, the process with which the neuron chooses the rules to be applied is uninfluent: at every computation step the only existing rule is chosen, and is applied as many times as possible (i.e., maximizing the number of spikes which are consumed).

Consider the SN P system depicted in Figure 2, in which all the neurons work in the maximal parallel way. Assume that a sequence of spikes comes from the environment, during k consecutive time steps. Such spikes can be considered as the binary encoding of a k-bit natural number N, by simply interpreting as 1 (resp.,

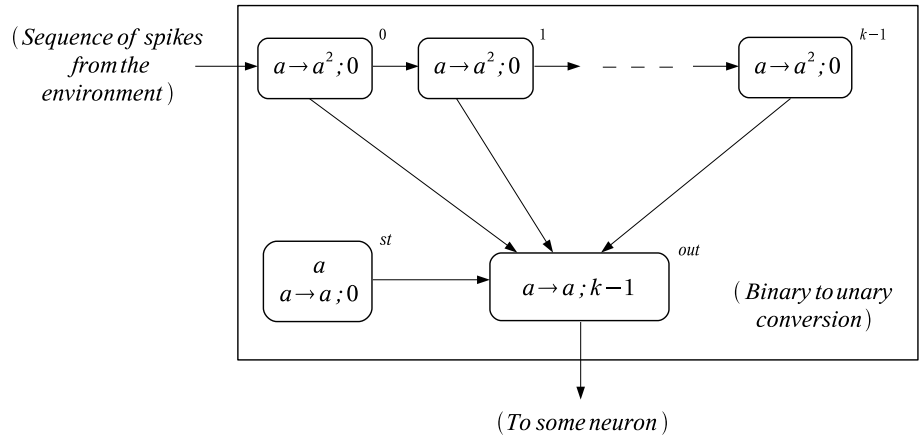

Fig. 2. A maximally parallel SN P system that converts a binary encoded positive integer number to unary form

0) the presence (resp., the absence) of a spike in each time step. The system works as follows. In the first step, the most significant bit of N enters into the neuron labelled with 0. Simultaneously, neuron *st* fires and sends a spike to neuron *out*, that will contain the resulting unary encoding of N. This is done in order to close such a neuron, so that it does not receive the intermediate results produced by neurons $0, 1, \ldots, k-1$ during the conversion. During the next $k-1$ steps, all subsequent bits of N enter into the system. Neurons $0, 1, \ldots, k-1$ act as a shift register, and they duplicate every spike before sending both copies to the neighbouring neuron. In this way, since rules are applied in the maximally parallel way, at the end of the k-th step each neuron j, with $j \in \{0, 1, \ldots, k-1\}$, will contain 2^j spikes if the j-th bit of N is 1, otherwise it will contain 0 spikes. At the $(k+1)$-th step, neuron *out* becomes open again, and receives exactly N spikes. Two little annoying details are that this neuron emits a "spurious" spike at the $(k+1)$-th computation step, and that it becomes again closed for further $k-1$ time steps. The first spike emitted from the subsystem has obviously to be ignored, whereas during the $(2k)$-th step neuron *out* emits the N spikes we are interested in. Note that this module can be used only once, since neuron *st* initially contains just one spike. By making neuron *st* work in the sequential mode (instead of the maximally parallel mode), and slightly complicating the structure of the system, we can also convert a sequence of n numbers arriving from the environment in $n \cdot k$ consecutive time steps.

By looking at Figure 3, we can see that for any instance $(\{v_1, v_2, \ldots, v_n\}, S)$ of Subset Sum it is possible to build a maximally parallel nondeterministic SN P system that solves it as follows. During the first k computation steps, the system reads n sequences of spikes, each one encoding in binary the natural number v_i. Each sequence goes to an SN subsystem which performs the conversion from binary

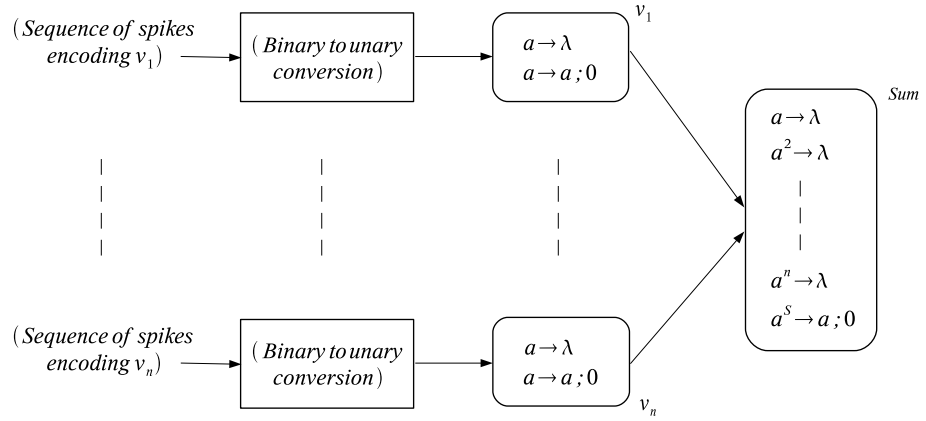

Fig. 3. A nondeterministic SN P system that solves the SUBSET SUM problem by working in the maximal parallel way (but for the neuron Sum)

to unary, as illustrated in Figure 2. Thus in the $(2k)$-th step, for all $i \in \{1, 2, \ldots, n\}$, v_i spikes reach the neuron labelled with v_i. At the next step, each of these neurons *nondeterministically* decides whether to propagate the spikes it has received, or to delete them. Hence, the rules of neurons v_i are applied not only in the maximal parallel way, but also in a nondeterministic way (in the sense that one of the two rules is nondeterministically chosen, and then is applied in the maximal parallel way). In step $2k + 2$, the neuron labelled with Sum checks whether the number of spikes it has gathered is equal to S; if so, it fires one spike to the environment, thus signalling that the given instance of SUBSET SUM is positive. Conversely, the instance is negative if and only if no spike is emitted from the system during the $(2k + 2)$-nd computation step. The forgetting rules which occur in neuron Sum are needed so that at step $k + 2$ all the spurious spikes that (eventually) reach the neuron (coming from the modules that have performed the conversions from binary to unary) are removed from the system, and are not added to the spikes that arrive at step $2k+1$. Of course, here we are assuming that $S > n$; if this is not the case, then the rules must be modified accordingly. Note that neuron Sum is deterministic, and works in the sequential way. We also observe that, if desired, we can use two neurons instead of one in the last layer of the system, as we have done in Figure 1. The first neuron would be just like Sum, the only difference being that the rule $a^S \to a; 0$ becomes $a^*/a^S \to a; 0$. The second neuron would contain the same forgetting rules as Sum, and the firing rule $a^*/a^{S+1} \to a; 1$ instead of $a^S \to a; 0$. In this way, the instance would be signalled as positive if and only if a single spike is emitted during the steps $2k + 2$ and $2k + 3$.

This solution to the SUBSET SUM problem is still semi–uniform: a single system is able to solve all the instances that have the same value of S, and in which all v_i are k-bit numbers. A way to make the system uniform would be to read

from the environment also the value of S, encoded in binary form, and send a corresponding number of spikes to a predefined neuron. The problem would thus reduce to comparing with S the number of spikes obtained by nondeterministically choosing some of the v_i. In the next section we will operate in a similar way; however, instead of comparing the contents of two neurons, expressed in unary form, we will operate as follows: we will keep S in binary form, and we will convert the sum of v_i from unary to binary. In this way, the problem to compare S with the sum of v_i is reduced to a bit-by-bit comparison.

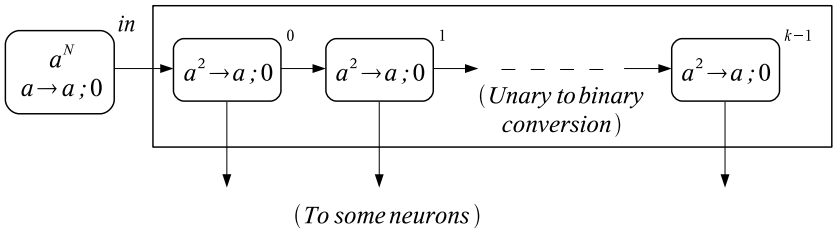

Fig. 4. A maximally parallel SN P system that converts a unary encoded positive integer number to binary form

Before doing all this, let us show that the conversion from binary to unary of a given natural number cannot be performed in polynomial time without using maximal parallelism. Let Π be a deterministic SN P system that works in the sequential way: all the neurons compute in parallel with respect to each other, but in each neuron only one rule is chosen and applied at every computation step. To be precise, even if the contents of the neuron would allow to apply the chosen rule many times (such as it happens, for example, with the rule $a \to a^2; 0$ and five spikes occurring in the neuron), only one instance of the rule is applied (in the example, one spike is consumed and two spikes are produced). Without loss of generality, we can assume that the regular expressions that occur in Π have the form a^i with $i \leq 3$ or $a(aa)^+$, which suffice to obtain computationally complete SN P systems [7]. Let m be the number of neurons in Π, and let $t(k)$ be the polynomial number of steps needed by Π to convert the k-bit natural number N given in input from the binary to the unary form. Moreover, let Q be the maximum number of spikes produced by any rule of Π. Since in the worst case every neuron is connected with every other neuron, the total number of spikes occurring in the system is incremented by at most mQ units during each computation step. If we denote by M the number of spikes occurring in the initial configuration, then after $t(k)$ computation steps the number of spikes in the system will be at most $M + mQt(k)$. This quantity is polynomial with respect to both the number of steps and the description size of Π, and thus it cannot cover the exponential gap that exists between the number of objects needed to represent N in binary and in unary form.

5 A Uniform Family of SN P Systems for Subset Sum

Let us present now a uniform family $\{\Pi(\langle n,k\rangle)\}_{n,k\in\mathbb{N}}$ of SN P systems such that for every n and k in $\mathbb{N}$, the system $\Pi(\langle n,k\rangle)$ solves all possible instances $(\{v_1,v_2,\ldots,v_n\},S)$ of Subset Sum in which $v_1,v_2,\ldots,v_n$ and S are all k-bit natural numbers.

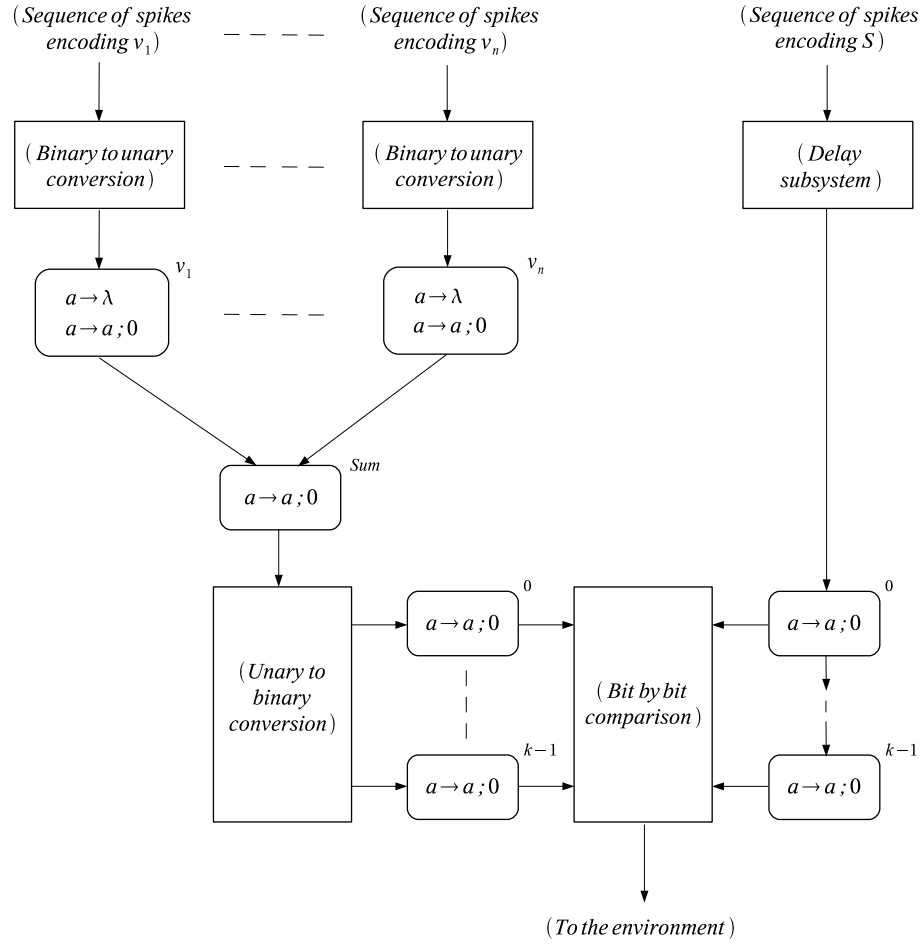

Fig. 5. The uniform SN P system $\Pi(\langle n,k\rangle)$ that solves all instances of Subset Sum composed by k-bit natural numbers

As told in the previous section, we first need a subsystem that allows to convert natural numbers from the unary to the binary form. Consider the system depicted in Figure 4. All the neurons work in the maximally parallel way. Initially, neuron

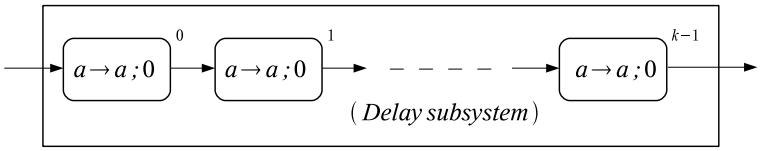

Fig. 6. An SN P system that delays of k steps the sequence of spikes given in input

in contains N spikes, where N is the k-bit integer number we want to convert. In the first computation step, all the spikes contained in neuron *in* are sent to neuron 0 (thus entering into the subsystem), thanks to the rule $a \to a$ applied in the maximally parallel way. In the second step, rule $a^2 \to a$ in neuron 0 halves the number of spikes (indeed, computing an integer division by 2) and sends the result to neuron 1. If the initial number of spikes was even, then in neuron 0 no spikes are left; instead, if the initial number of spikes was odd, then exactly one spike will remain in neuron 0. Hence, the number of spikes remaining in neuron 0 is equal to the value of the least significant bit of the binary encoding of N. The computation proceeds in a similar way during the next $k-1$ steps; in each step, the next bit (from the least significant to the most significant) of the binary encoding of N is computed. Note that the bits that have already been computed are unaffected by subsequent computation steps. After k computation steps, the neurons labelled with $0, 1, \ldots, k-1$ contain all the bits of the binary encoding of N. In order to use such bits, we can connect these neurons to other k neurons, which should be kept closed during the conversion by means of a trick similar to that used in Figure 2.

The SN P system $\Pi(\langle n, k\rangle)$ that solves all the instances $(\{v_1, v_2, \ldots, v_n\}, S)$ of Subset Sum which are composed by k-bit natural numbers is depicted (in a schematic way) in Figure 5. The sequences of spikes that encode $v_1, v_2, \ldots, v_n$ and S in binary form arrive simultaneously from the environment, and enter into the system from the top. The values $v_1, \ldots, v_n$ are first converted to unary and then some of them are summed, as before; the sequence of bits in S, instead, is just delayed (using the subsystem depicted in Figure 6) so that it arrives in the "Bit by bit comparison" subsystem simultaneously with the binary representation of the sum of the v_i. Such a binary representation is obtained through the subsystem depicted in Figure 4. The bit-by-bit comparison subsystem (depicted in Figure 7) emits a spike if and only if all the bits of the two integer numbers given in input match, that is, if and only if the two numbers are equal. If we denote by $x = \sum_{i=0}^{k-1} x_i 2^i$ and $y = \sum_{i=0}^{k-1} y_i 2^i$ the numbers to be compared, the subsystem computes the following boolean function:

$$\textsc{Compare}(x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}) = \bigwedge_{i=0}^{k-1} \left(\neg(x_i \oplus y_i)\right) = \neg\left(\bigvee_{i=0}^{k-1} (x_i \oplus y_i)\right)$$

where $\oplus$ denotes the logical XOR operation. The subsystem works as follows. Bits x_i and y_i are XORed by the neurons depicted on the top of Figure 7. The neuron

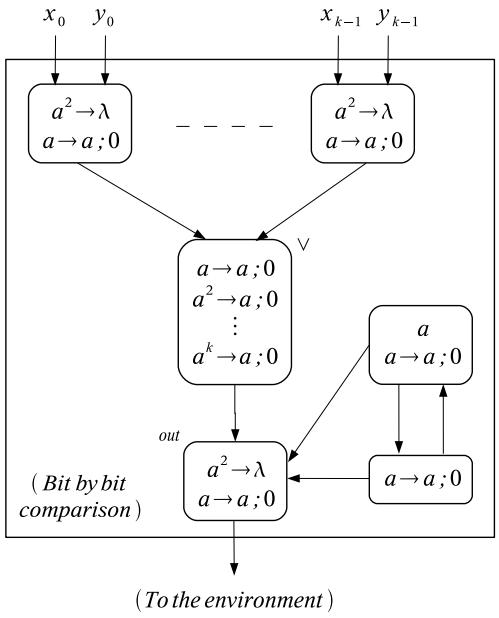

Fig. 7. A standard SN P system that compares two k-bit natural numbers

labelled with $\vee$ computes the logical OR of its inputs: precisely, it emits one spike if and only if at least one spike enters into the neuron. Neuron *out* receives the output produced by $\vee$ and computes its logical negation (NOT). In order to be able to produce one spike if no spikes come from *out*, we use two auxiliary neurons that send to *out* one spike at every computation step. The number of neurons, as well as the total number of rules, used by $\Pi(\langle n, k\rangle)$ is polynomial with respect to n and k.

We conclude by observing that the output of the SN P system depicted in Figure 5 has to be observed exactly after $3k+6$ computation steps. One spike will eventually be emitted by the system before this time, since the conversion from binary to unary of $v_1, v_2, \ldots, v_n$ produces some spurious spikes before emitting the result. These spurious spikes are added in neuron *sum*, and the result of this addition is first converted to binary and then sent to the comparison subcircuit. However, by carefully calibrating the delay subsystem this value does not interfere with the bits of S, that will arrive to the comparison subsystem only later. From a direct inspection of the system in Figure 5, it is easily seen that the correct delay to be applied is of $3k+2$ steps.

6 Conclusions and Directions for Future Research

In this paper we have continued the study concerning the computational power of SN P systems, started in [10]. In particular, by slightly extending the original definition of SN P systems given in [8] and [9] we have shown that by exploiting nondeterminism it is possible to solve numerical **NP**–complete problems such as Subset Sum and Partition (which can be considered as a particular case of Subset Sum).

However, a drawback of this solution is that the system may require to specify an exponential number of spikes both when defining the rules and when describing the contents of the neurons in the initial configuration. Hence, we have shown that the numbers $v_1, v_2, \ldots, v_n$ occurring in the instance of Subset Sum can be given to the system in binary form, and subsequently converted to the unary form in polynomial time. In this way we have proved that the capability of the above system to solve Subset Sum does not derive from the fact that it requires an exponential effort to be initialized.

The new SN P system thus obtained still provides a *semi–uniform* solution, since for each instance of the problem we need to build a specifically designed SN P system to solve it. Thus, we have finally proposed a family $\{\Pi(\langle n, k\rangle)\}_{n,k\in\mathbb{N}}$ of SN P systems such that for all $n, k \in \mathbb{N}$, $\Pi(\langle n, k\rangle)$ solves all the instances $(\{v_1, v_2, \ldots, v_n\}, S)$ of Subset Sum such that $v_1, v_2, \ldots, v_n$ and S are all k-bit natural numbers. This solution assumes that for each neuron (or, at least, for each subsystem) it is possible to choose whether such a neuron (resp, subsystem) works in a deterministic vs. nondeterministic way, and in the sequential vs. the maximally parallel way.

In [10] we have also studied the computational power of *deterministic* accepting SN P systems working in the sequential way. In particular, we have shown that they can be simulated by deterministic Turing machines with a polynomial slowdown. This means that they are not able to solve **NP**–complete problems in polynomial time unless $\mathbf{P} = \mathbf{NP}$, a very unlikely situation. In future work, we will address the study of the computational power of deterministic accepting SN P systems working in the maximally parallel way.

Acknowledgments

We gratefully thank Gheorghe Păun for introducing the authors to the stimulating subject of spiking neural P systems, and for asking us a "Milano theorem" (in the spirit of [22]) about their computational power, during the Fifth Brainstorming Week on Membrane Computing, held in Seville from January 29$^{\text{th}}$ to February 2$^{\text{nd}}$, 2007.

We are also truly indebted with Mario de Jesús Pérez-Jiménez for stimulating observations and suggestions made on a previous version of this paper.

References

1. A. Alhazov, R. Freund, M. Oswald. Cell/symbol complexity of tissue P systems with symport/antiport rules. *International Journal of Foundations of Computer Science*, 17(1):3–26, 2006.
2. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez. On String Languages Generated by Spiking Neural P Systems. In M.A. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero (eds.), *Fourth Brainstorming Week on Membrane Computing*, Vol. I RGCN Report 02/2006, Research Group on Natural Computing, Sevilla University, Fénix Editora, 169–194.
3. T.H. Cormen, C.H. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT Press, Boston, 1990.
4. R. Freund, Gh. Păun, M.J. Pérez-Jiménez. Tissue-like P Systems with Channel States. *Theoretical Computer Science*, 330(1):101–116, 2005.
5. M.R. Garey, D.S. Johnson. *Computers and Intractability. A Guide to the Theory on* **NP***–Completeness*. W.H. Freeman and Company, 1979.
6. W. Gerstner, W. Kistler. *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
7. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosík, S. Woodworth. Normal Forms for Spiking Neural P Systems. *Theoretical Computer Science*, 372(2-3):196–217, 2007.
8. M. Ionescu, Gh. Păun, T. Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2-3):279–308, 2006.
9. M. Ionescu, A. Păun, Gh. Păun, M.J. Pérez-Jiménez. Computing with spiking neural P systems: Traces and small universal systems. In C. Mao, T. Yokomori, B.-T. Zhang (eds.), *DNA Computing,* 12^{th} *International Meeting on DNA Computing (DNA12)*, Seoul, Korea, June 5-9, 2006, Revised Selected Papers. LNCS 4287, Springer, 2006, 1–16.
10. A. Leporati, C. Zandron, C Ferretti, G. Mauri. On the Computational Power of Spiking Neural P Systems. In M.A. Gutiérrez-Naranjo et al. (eds.), *Fifth Brainstorming Week on Membrane Computing*, Research Group on Natural Computing, Sevilla University, Fénix Editora, 2007 (in print).
11. A. Leporati, C. Zandron, M.A. Gutiérrez-Naranjo. P systems with input in binary form. *International Journal of Foundations of Computer Science*, 17(1):127–146, 2006.
12. W. Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):32–36, 2002.
13. W. Maass, C. Bishop (eds.). *Pulsed Neural Networks*, MIT Press, Cambridge (MA), 1999.
14. C. Martín-Vide, J. Pazos, Gh. Păun, A. Rodríguez-Patón. A new class of symbolic abstract neural nets: Tissue P systems. In *Proceedings of COCOON 2002*, Singapore, LNCS 2387, Springer-Verlag, Berlin, 290–299.
15. M. Oswald. Independent agents in a globalized world modelled by tissue P systems. *Conf. Artificial Life and Robotics*, 2006.
16. Gh. Păun. Computing with Membranes. *Journal of Computer and System Sciences*, 61:108–143, 2000. See also Turku Centre for Computer Science — TUCS Report No. 208, 1998. Available at: *http://www.tucs.fi/Publications/techreports/ TR208.php*
17. Gh. Păun. Computing with Membranes. An Introduction. *Bulletin of the EATCS*, 67(2):139–152, 1999.

18. Gh. Păun. *Membrane Computing. An Introduction*. Springer–Verlag, Berlin, 2002.
19. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg. Infinite spike trains in spiking neural P systems. Submitted for publication.
20. Gh. Păun, G. Rozenberg. A Guide to Membrane Computing. *Theoretical Computer Science*, 287(1):73–100, 2002.
21. Gh. Păun, Y. Sakakibara, T. Yokomori. P Systems on Graphs of Restricted Forms. *Publicationes Mathematicae Debrecen*, 60:635–660, 2002.
22. C. Zandron, C. Ferretti, G. Mauri. Solving **NP**–Complete Problems Using P Systems with Active Membranes. In I. Antoniou, C.S. Calude, M.J. Dinneen (eds.), *Unconventional Models of Computation*, Springer-Verlag, London, 2000, 289–301.
23. The P systems Web page: http://psystems.disco.unimib.it/