International Journal of Foundations of Computer Science
Vol. 18, No. 3 (2007) 435–455

World Scientific
www.worldscientific.com

SPIKING NEURAL P SYSTEMS: AN EARLY SURVEY

GHEORGHE PĂUN
Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania, and
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: george.paun@imar.ro, gpaun@us.es

MARIO J. PÉREZ-JIMÉNEZ
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: marper@us.es

ARTO SALOMAA
Turku Center for Computer Science – TUCS
Lemminkäisenkatu 14, 20520 Turku, Finland
asalomaa@utu.fi

Received 17 June 2006
Accepted 20 September 2006
Communicated by K. Nakano

Spiking neural P systems were introduced in the end of the year 2005, in the aim of incorporating in membrane computing the idea of working with unique objects ("spikes"), encoding the information in the time elapsed between consecutive spikes sent from a cell/neuron to another cell/neuron. More than one dozen of papers where written in the meantime, clarifying many of the basic properties of these devices, especially related to their computing power.

The present paper quickly surveys the basic ideas and the basic results, presenting a complete to-date bibliography, and also giving a completing result related to the normal forms possible for spiking neural P systems: we prove that the indegree of such systems (the maximal number of incoming synapses of neurons) can be bounded by 2 without losing the computational completeness.

A series of research topics and open problems are formulated.

Keywords: spiking neuron; membrane computing; P system; register machine; semilinear set; normal form; chomsky hierarchy.

2000 Mathematics Subject Classification: 68Q10, 68Q42, 68Q45

1. Introduction; A Quick Overview of the Literature

Spiking neural P systems (SN P systems, for short) were introduced in [13] in the aim of defining computing models based on ideas specific to spiking neurons, currently much investigated in neural computing (see, e.g., [7], [15], [16]). The resulting models are a variant of tissue-like and neural-like P systems from membrane computing (see [19] and the up-to-date information at the web site [24]), with very specific ingredients and way of functioning.

Very shortly, an SN P system consists of a set of *neurons* (cells, consisting of only one membrane) placed in the nodes of a directed graph and sending signals (*spikes*, denoted in what follows by the symbol a) along *synapses* (arcs of the graph). Thus, the architecture is that of a tissue-like P system, with only one kind of objects present in the cells. The objects evolve by means of *spiking rules*, which are of the form $E/a^c \to a; d$, where E is a regular expression over $\{a\}$ and c, d are natural numbers, $c \geq 1, d \geq 0$. The meaning is that a neuron containing k spikes such that $a^k \in L(E), k \geq c$, can consume c spikes and produce one spike, after a delay of d steps. This spike is sent to all neurons to which a synapse exists outgoing from the neuron where the rule was applied. We will give details in Section 2. There also are *forgetting rules*, of the form $a^s \to \lambda$, with the meaning that $s \geq 1$ spikes are forgotten, provided that the neuron contains exactly s spikes. We say that the rules "cover" the neuron, all spikes are taken into consideration when using a rule. (This is another major difference with respect to usual P systems, where a sub-multiset of the multiset of objects is "rewritten" by each applied rule.) The system works in a synchronized manner, i.e., in each time unit, each neuron which can use a rule should do it, but the work of the system is sequential in each neuron: only (at most) one rule is used in each neuron. One of the neurons is considered to be the *output neuron*, and its spikes are also sent to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1, the other moments are marked with 0. This binary sequence is called the *spike train* of the system – it might be infinite if the computation does not stop.

In the spirit of spiking neurons, the result of a computation is encoded in the distance between consecutive spikes sent into the environment by the (output neuron of the) system. (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 [1].) In [13] only the distance between the first two spikes of a spike train was considered, then in [21] 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 [22]. Languages – even on arbitrary alphabets – can be obtained also in other ways: following the path of a designated spike across neurons, as proposed in [4] (this essentially resembles the trace languages investigated for usual P systems, see [19] and [24]), or generalizing the form of rules. Specifically, one uses rules of the form $E/a^c \to a^p; d$, with the meaning that, provided that the neuron is covered by E, c spikes are consumed and p spikes are produced, and sent to all connected neurons after d steps (such rules are called *extended*). Then, with a step when the system sends out i spikes, we associate a symbol b_i, and thus we get a language over an alphabet as many symbols as the number of spikes simultaneously produced. This case was investigated in [6].

Other extensions were proposed in [11] and [10], 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 [14], 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 [18] 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 [9], 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, while the outdegree of the synapse graph can be bounded by 2, and the regular expressions from firing rules can be of very restricted forms.

The dual problem, of the indegree bounding, was formulated as an open problem in [9]. We solve here this problem, proving, like in the case of the outdegree, that again a normal form holds true: systems with indegree two are computationally complete.

In the next section we will introduce the spiking neural P systems, then

(Section 3) we will give some examples, also introducing in this way other ways of using them (generating strings, considering traces). Section 4 presents some results, without proofs, illustrating the computing power of these devices. Section 5 gives the indegree normal form mentioned above. Further remarks and, mainly, further research topics are mentioned in Section 6.

2. Spiking Neural P Systems

We assume the reader to have some familiarity with (basic elements of) language and automata theory, e.g., from [23], and introduce directly the computing devices we discuss here.

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

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

where:

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

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

where:

a) $n_i \geq 0$ 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) $E/a^c \rightarrow a; d$, where E is a regular expression with a the only symbol used, $c \geq 1$, and $d \geq 0$;
   (2) $a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction[a] that $a^s \in L(E)$ for no rule $E/a^c \rightarrow a; d$ of type (1) from R_i;

(3) $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses*);
(4) $out \in \{1, 2, \ldots, m\}$ 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 $k \geq c$, then the rule $E/a^c \rightarrow a; d$ can be applied, and this means that c spikes are consumed, only $k - c$ remain 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 $d \geq 1$, if the rule is used in step t, then in steps $t, t + 1, t + 2, \ldots, t + d - 1$ 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 $t + d$,

[a]This restriction is introduced in order to decrease the non-determinism of the system, but from a mathematical point of view it could be of interest to investigate the case when it is omitted.

the neuron spikes and becomes again open, hence can receive spikes (which can be used in step $t + d + 1$). A spike emitted by a neuron σ_i replicates and goes to all neurons σ_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 which 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 $L(E_1) \cap L(E_2) \neq \emptyset$, 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 $n_1, n_2, \ldots, n_m$ 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 again open, i.e., the configuration is written in the form $\langle p_1/q_1, \ldots, p_m/q_m \rangle$; the neuron σ_i contains $p_i \geq 0$ spikes and will be open after $q_i \geq 0$ 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 $C_1 \Longrightarrow C_2$. 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 $t_1, t_2, \ldots$ of natural numbers $1 \leq t_1 < t_2 < \ldots$, 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 [13], 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 Π. The set of all numbers computed in this way by Π 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 [21] to several other sets of numbers which can be associated with a spike train: taking into account the intervals between the first k spikes, $k \geq 2$ (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 also an input neuron is provided, then a transducer is obtained, translating input binary strings into binary strings. Some of these possibilities will be illustrated below.

3. Four Examples

Before presenting results concerning the computing power of SN P systems, we consider several examples, some of them also indicating the modifications mentioned above.

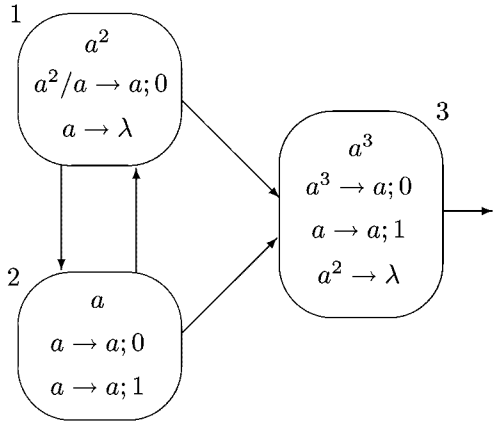

Fig. 1. An SN P system generating all natural numbers greater than 1.

The first example, recalled from [13], is

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

$$\sigma_2 = (1, \{a \to a; 0, \quad a \to a; 1\}),$$
$$\sigma_3 = (3, \{a^3 \to a; 0, \quad a \to a; 1, \quad a^2 \to \lambda\}),$$
$$syn = \{(1,2),\ (2,1),\ (1,3),\ (2,3)\}.$$

This system is given in a graphical form in Figure 1, 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" (a circle or 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, *out*, it is also suggestive to draw a short arrow which exits from it, pointing to the environment.

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

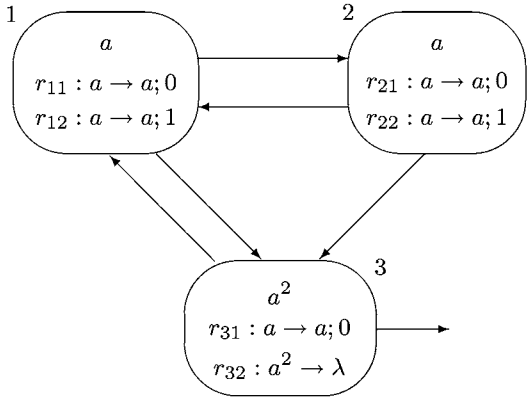

Fig. 2. The initial configuration of system Π_2.

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

However, at any moment, starting with the first step of the computation, neuron σ_2 can choose to use the rule $a \to a; 1$. On the one hand, this means that the spike of neuron σ_1 cannot enter neuron σ_2, it only goes to neuron σ_3; in this way, neuron σ_2 will never work again because it remains empty. On the other hand, in the next step neuron σ_1 has to use its forgetting rule $a \to \lambda$, while neuron σ_3 fires, using the rule $a \to a; 1$. Simultaneously, neuron σ_2 emits its spike, but it cannot enter

neuron σ_3 (it is closed this moment); the spike enters neuron σ_1, but it is forgotten in the next step. In this way, no spike remains in the system. The computation ends with the expelling of the spike from neuron σ_3. Because of the waiting moment imposed by the rule $a \to a; 1$ from neuron σ_3, the two spikes of this neuron cannot be consecutive, but at least two steps must exist in between.

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

The next example, borrowed from [2], is presented in Figure 2, and it is meant to generate binary strings.

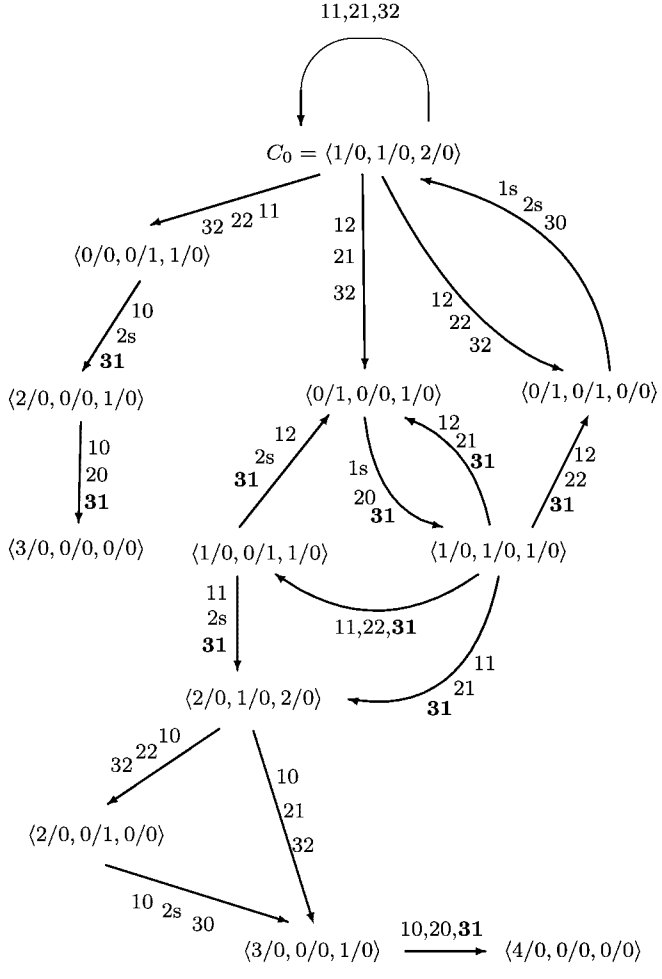

Fig. 3. The transition diagram of system Π_2.

Its evolution can be analyzed on a transition diagram as that from Figure 3,

which is a very useful tool for studying systems with a bounded number of spikes present in their neurons (we also say that such a system is *finite*): because the number of configurations reachable from the initial configuration is finite, we can place them in the nodes of a graph, and between two nodes/configurations we draw an arrow if and only if a direct transition is possible between them. In Figure 3, also the rules used in each neuron are indicated, with the following conventions: for each r_{ij} we have written only the subscript ij, with **31** being written in boldface, in order to indicate that a spike is sent out of the system at that step; when a neuron $\sigma_i, i = 1, 2, 3$, uses no rule, we have written $i0$, and when it spikes (after being closed for one step), we write is.

We do not enter into details concerning the paths in this diagram. Anyway, the transition diagram of a finite SN P system can be interpreted as the representation of a non-deterministic finite automaton, with C_0 being the initial state, the halting configurations being final states, and each arrow being marked with 0 if in that transition the output neuron does not send a spike out, and with 1 if in the respective transition the output neuron spikes; in this way, we can identify the language generated by the system. In the case of the finite SN P system Π_2, the generated language is $L(\Pi_2) = (0^*0(11 \cup 111)^*110)^*0^*(011 \cup 0(11 \cup 111)^+(0 \cup 00)1)$.

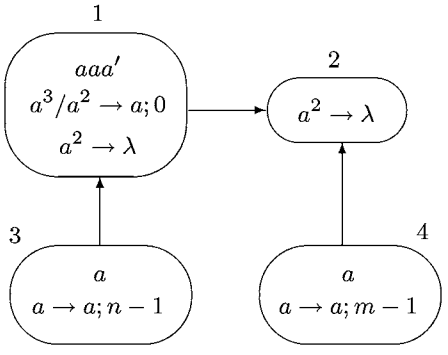

Fig. 4. The initial configuration of system Π_3.

We consider now an SN P system, the one from Figure 4, 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 σ_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 which 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 Π_3 whose initial configuration is given in Figure 4. It generates the language $T(\Pi_3) = \{b_1^n, b_2^m\}$, for $n, m \geq 1$. In the first step, neuron σ_1 consumes or does not consume the marked spike, thus keeping it inside or sending it to neuron σ_2. One spike remains in neuron σ_1 and one is placed in neuron σ_2. Simultaneously, neurons σ_3 and σ_4 fire, and they spike after $n-1$ and $m-1$ steps, respectively. Thus, in steps n and m, neurons σ_1 and σ_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.

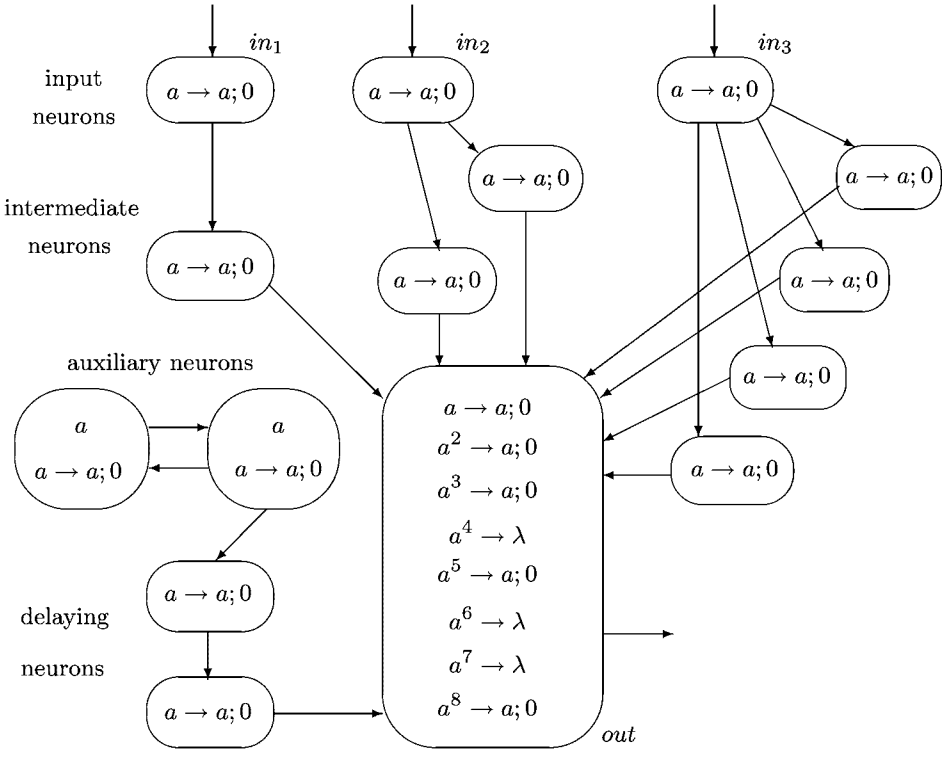

Fig. 5. An SN P transducer computing a Boolean function of three variables.

The last example illustrates the following result from [22]: *Any function* $f : \{0,1\}^k \longrightarrow \{0,1\}$ *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 is suggested in Figure 5, where a system is presented which computes the function $f : \{0,1\}^3 \longrightarrow \{0,1\}$ 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 $f(b_1, b_2, b_3)$.

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 $N_2SNP_m(rule_k, cons_p, forg_q)$ the family of all sets $N_2(\Pi)$ computed as specified in Section 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 $E/a^r \rightarrow a; t$ having $r \leq p$, and all forgetting rules $a^s \rightarrow \lambda$ having $s \leq q$. When one of the parameters 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 length sets of finite, regular, and recursively enumerable languages, whose families are denoted by FIN, REG, RE. We also invoke below the family of recursive languages, REC (those languages with a decidable membership).

The following results were proved in [13] and extended in [21] to other ways of defining the result of a computation.

Theorem 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* $k \geq 2, p \geq 3, q \geq 3$.

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

Point (ii) was proved in [13] 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).

Let us now pass to mentioning some results about languages generated by SN P systems, starting with the restricted case of binary strings. We denote by $L(\Pi)$ the set of strings over the alphabet $B = \{0,1\}$ describing the spike trains associated with halting computations in Π; then, we denote by $LSNP_m(rule_k, cons_p, forg_q)$ the family of languages $L(\Pi)$, generated by SN P systems Π with the complexity bounded by the parameters m, k, p, q as specified above.

When using only systems with at most s spikes in their neurons (finite), we write $LSNP_m(rule_k, cons_p, forg_q, bound_s)$ for the corresponding family. As usual, a parameter m, k, p, q, s is replaced with $*$ if it is not bounded.

Theorem 2. *(i) There are finite languages (for instance, $\{0^k, 10^j\}$, for any $k \geq 1$, $j \geq 0$) which cannot be generated by any SN P system, but for any $L \in FIN$, $L \subseteq B^+$, we have $L\{1\} \in LSNP_1(rule_*, cons_*, forg_0, bound_*)$, and if $L = \{x_1, x_2, \ldots, x_n\}$, then we also have $\{0^{i+3}x_i \mid 1 \leq i \leq n\} \in LSNP_*(rule_*, cons_1, forg_0, bound_*)$.*

(ii) The family of languages generated by finite SN P systems is strictly included in the family of regular languages over the binary alphabet, but for any regular language $L \subseteq V^*$ there is a finite SN P system Π and a morphism $h : V^* \longrightarrow B^*$ such that $L = h^{-1}(L(\Pi))$.

(iii) $LSNP_*(rule_*, cons_*, forg_*) \subset REC$, but for every alphabet $V = \{a_1, a_2, \ldots, a_k\}$ there are two symbols b, c not in V, a morphism $h_1 : (V \cup \{b, c\})^* \longrightarrow B^*$, and a projection $h_2 : (V \cup \{b, c\})^* \longrightarrow V^*$ such that for each language $L \subseteq V^*$, $L \in RE$, there is an SN P system Π such that $L = h_2(h_1^{-1}(L(\Pi)))$.

These results show that 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 a language 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 λ 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 λ associated with steps when no spike exits the system) is considered below.

We denote by $LSN^eP_m(rule_k, cons_p, prod_q)$ the family of languages $L(\Pi)$, generated by SN P systems Π 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 Theorem 2 were proved in [6].

Theorem 3. *(i) $FIN = LSN^eP_1(rule_*, cons_*, prod_*)$ and this result is sharp in the sense that $LSN^eP_2(rule_2, cons_2, prod_2)$ contains infinite languages.*

(ii) $LSN^eP_2(rule_*, cons_*, prod_*) \subseteq REG \subset LSN^eP_3(rule_*, cons_*, prod_*)$; the second inclusion is proper, because $LSN^eP_3(rule_3, cons_4, prod_2)$ contains non-regular languages; actually, the family $LSN^eP_3(rule_3, cons_6, prod_4)$ contains non-semilinear languages.

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

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

5. Bounding the Indegree

In general, in the results above, one uses all features of SN P systems, but, as proved in [9], this is not necessary. Here are the results proven in [9]:

Theorem 4. *Universality of SN P systems (with standard rules) can be obtained even for systems (i) without using a delay in firing rules, (ii) without using forgetting rules, (iii) using only regular expressions of the forms* a^+ *and* $a^k, k \geq 1$, *in firing rules; (iv) each of these restrictions can be combined with the restriction of having a synapse graph with the outdegree at most two.*

Bounding also the indegree of the synapse graph was left as an open problem in [9]. We solve here this problem, affirmatively: like in the case of the outdegree, the bound two on the indegree does not prevent obtaining the computational completeness.

In the proof of this result we use the characterization of NRE by means of register machines, so that we introduce this notion here.

A (non-deterministic) register machine is a construct $M = (m, H, l_0, l_h, I)$, where m is the number of registers, H is the (finite) set of instruction labels, l_0 is the start label (labeling an ADD instruction), l_h is the halt label (assigned to instruction `HALT`), and I is the set of instructions; each label from H labels only one instruction from I, thus precisely identifying it. The instructions are of the following forms:

- l_i : (`ADD`$(r), l_j, l_k$) (add 1 to register r and then go to one of the instructions with labels l_j, l_k non-deterministically chosen),
- l_i : (`SUB`$(r), l_j, l_k$) (if register r is non-empty, then subtract 1 from it and go to the instruction with label l_j, otherwise go to the instruction with label l_k),
- l_h : `HALT` (the halt instruction).

A register machine M generates a set $N(M)$ of numbers in the following way: we start with all registers empty (i.e., storing the number zero), we apply the instruction with label l_0 and we continue to apply instructions as indicated by the labels (and made possible by the contents of registers); if we reach the halt instruction, then the number n present in register 1 at that time is said to be generated by M. (Without loss of generality we may assume that in the halting configuration all other registers are empty; also, we may assume that register 1 is never subject of SUB instructions, but only of ADD instructions.) It is known (see, e.g., [17]) that register machines generate all sets of numbers which are Turing computable.

Let us denote by $N_2SNP(ind_p, oud_q)$ the family of all sets $N_2(\Pi)$ computed by spiking neural P systems whose synapse graph has the indegree at most p and the

outdegree at most q. When one of the parameters p, q is not bounded, it is replaced with $*$.

Theorem 5. $N_2SNP(ind_2, oud_*) = NRE$.

Proof. We only have to prove the inclusion $NRE \subseteq N_2SNP(ind_2, oud_*)$, and we do this in two steps: first we modify the constructions from [13], [21] by which the similar inclusion is proved without a bound on the indegree (an SN P system is constructed, simulating a given register machine), then we also bound the indegree of the constructed SN P system.

For the first step, let us take an arbitrary register machine $M = (m, H, l_0, l_h, I)$, as specified above. We construct an SN P system Π such that $N(M) = N_2(\Pi)$, following the same idea as in [13]: modules are built for simulating the ADD and SUB instructions of M, as well as for providing the output (i.e., for sending out two spikes at the right moments of time). A neuron is associated with each register and with each label of M; if a register r contains the number n, then the corresponding neuron σ_r contains $2n$ spikes. At the beginning of the computation, there is only one spike in the system, in neuron σ_{l_0}. This means that in the first step, this neuron fires. In general, a neuron associated with a label of M is empty during the computation, except when it is activated by receiving a spike. We will describe furthermore the functioning of the system Π after presenting its modules.

An ADD instruction $l_i : (\texttt{ADD}(r), l_j, l_k)$ is simulated by a module as indicated in Figure 6, and a SUB instruction $l_i : (\texttt{SUB}(r), l_j, l_k)$ is simulated by a module as in Figure 7.

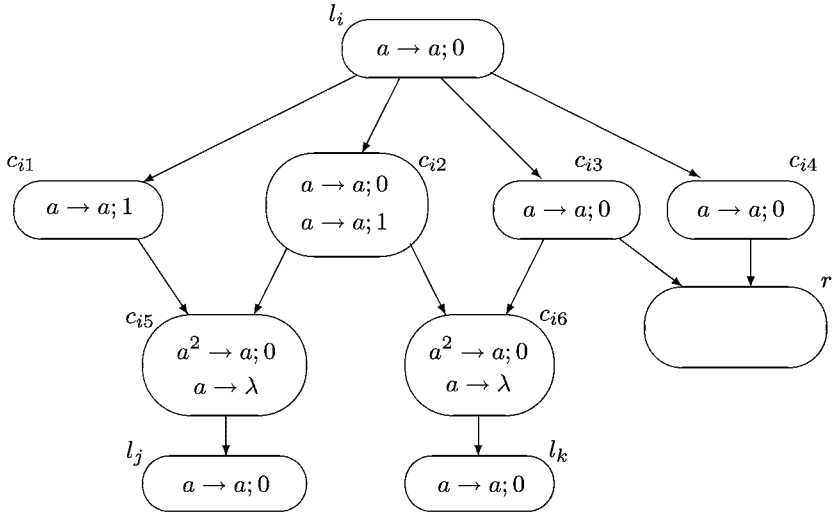

Fig. 6. Module ADD, simulating the instruction $l_i : (\texttt{ADD}(r), l_j, l_k)$.

These modules are similar to those in [13], with an additional care paid to the indegree of certain neurons. Specifically, one introduces the new neurons with labels c_{i5}, c_{i6} in module ADD and c_{i3}, c_{i4} in module SUB (note that all neurons $\sigma_{c_{ij}}$ are uniquely associated with the respective modules, because the label l_i is associated with only one instruction of M). Thus, we do not give full details concerning the functioning of these modules, and refer the reader to [13]; we only mention that the execution of a module starts by introducing a spike in neuron σ_{l_i}, and ends by introducing a spike in one of the neurons with labels l_j and l_k, thus activating the respective module. The correct choice of the "exit" neuron is ensured by the interplay between neurons with spiking rules $a \to a; 0$ and $a \to a; 1$. In the meantime, the neuron σ_r receives two spikes in the case of the ADD module, or is checked for zero and two spikes removed when this is possible, in the case of the SUB module.

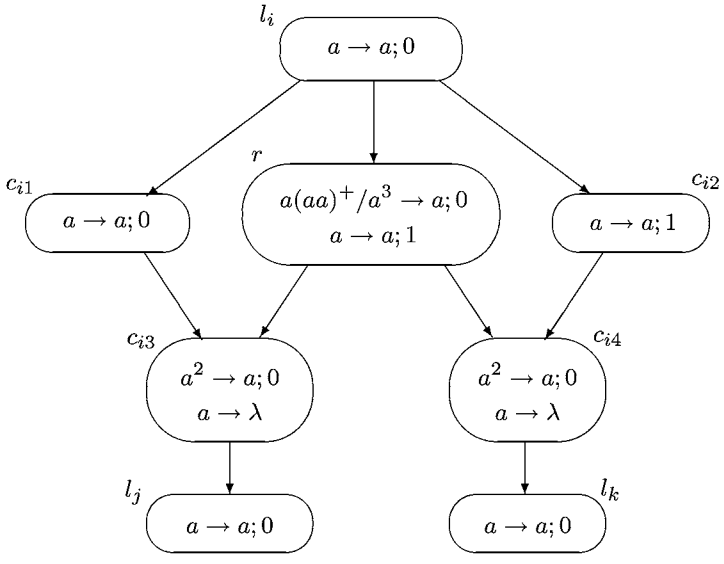

Fig. 7. Module SUB, simulating the instruction $l_i : (\texttt{SUB}(r), l_j, l_k)$.

If the computation of M never halts, then the work of ADD and SUB modules of Π never halts. If the instruction l_h : HALT is reached, then neuron σ_{l_h} receives a spike, and then the OUTPUT module from Figure 8 is activated. Note that the ADD modules do not use rules of neurons σ_r and that neuron σ_1 is only subject of modules ADD (register 1 is never decremented). This ensures the correct functioning of module OUTPUT, which will spike exactly twice, after a number of steps equal to the contents of register 1 of M. (Specifically, if the neuron σ_1 contains $2q$ spikes when l_h is reached, then the output neuron sends two spikes out, in steps t and $t + q$, for some t depending on the length of the computation in M.)

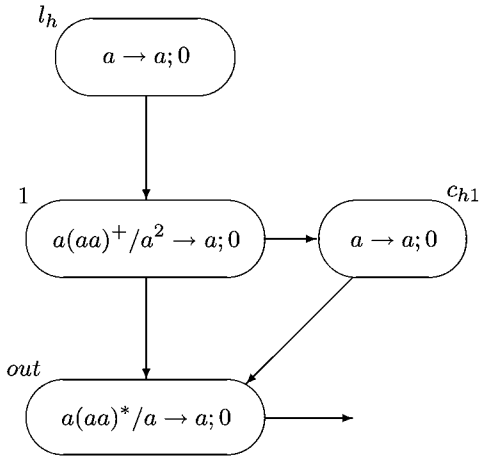

Fig. 8. Module OUTPUT.

The equality $N(M) = N_2(\Pi)$ is obtained. Let us now examine the indegree of the system Π. Neurons with labels c_{ij} have the indegree one or two, but neurons associated with labels of M and with registers of M can have an arbitrarily large indegree.

The case of neurons σ_l, where $l \in lab(M)$, is simpler: in each step of a computation, each such neuron can receive at most one spike along one of its incoming synapses. By introducing intermediate neurons as suggested in Figure 9, we can replace the synapses to neuron σ_l in such a way that its indegree becomes 2. Note that instead of one computation step, we perform now a number of steps of the order of $\log_2 k$, where k was the previous indegree of the neuron.

Slightly more complex is the situation of neurons σ_r: in a step of a computation, such a neuron receives *no spike* if it is not involved in the current operation, *one spike* if it is involved in a SUB instruction, or *two spikes* if it is involved in an ADD instruction. Assume that for a neuron σ_r we have the synapses $(e_j, r), 1 \leq j \leq s$, along which one spike can come, and the pairs of synapses $(c_j, r), (d_j, r)$, for some $1 \leq j \leq k$, such that one of the pairs of neurons $(\sigma_{c_j}, \sigma_{d_j})$ sends two spikes to σ_r (clearly, e_j corresponds to labels l_i, c_j corresponds to labels c_{i3}, and d_j to labels $c_{i,4}$). Then we can proceed as follows: we apply the procedure described in Figure 9 separately for neurons σ_{c_j}, for neurons σ_{d_j}, and for neurons σ_{e_j}, concentrating step by step the synapses until reaching an intermediate unique neuron $\sigma_C, \sigma_D, \sigma_E$, respectively. From neurons σ_D, σ_E we build synapses to a further neuron, σ_{DE}. Now, from σ_C and σ_{DE} we construct synapses to the neuron σ_r. The indegree of all neurons is now at most two (of course, "delaying" neurons, using a rule $a \to a; 0$ only for passing the spike further, are necessary if $s \neq k$, in order to synchronize the the times of spikes arriving in σ_r).

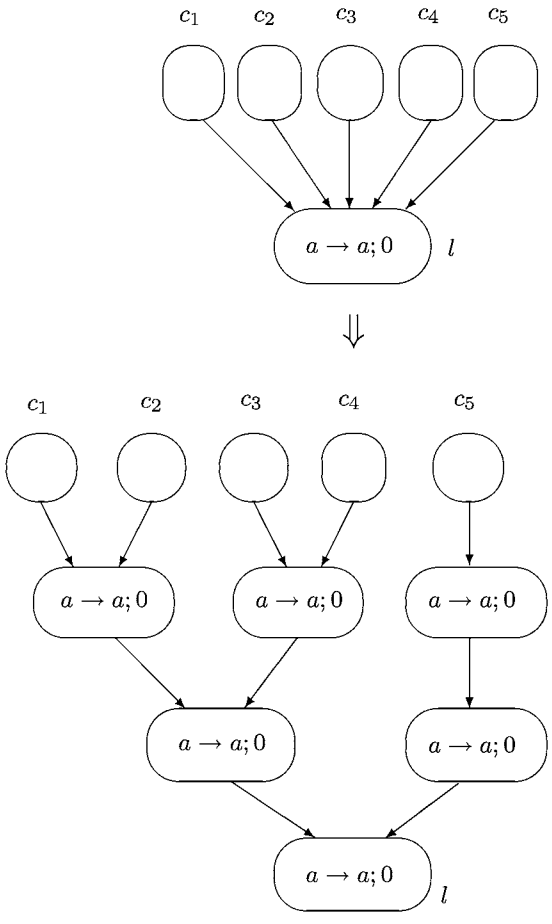

Fig. 9. Decreasing the indegree of neurons $\sigma_l, l \in lab(M)$.

Still, a problem remains to be solved, that of the synchronization of the system. The procedures described above take more steps than initially necessary for the spikes to reach their targets. Let us denote by α the maximal number of steps necessary in any of the previously described procedures for sending the spikes from the input neurons to the output neurons.

First, we add "delaying" neurons to constructions as the one in Figure 9, such that all blocks of this type take exactly α steps for sending the spikes from the input to the output neurons (this is an easy task: just add neurons with the rule $a \to a; 0$ as many times as necessary).

Let us now examine again the modules ADD and SUB as changed after decreasing the indegree.

In module ADD, the way from $\sigma_{c_{i3}}$ and $\sigma_{c_{i4}}$ to σ_r takes now α steps instead of one; in order to re-synchronize the process, we have to add $\alpha - 1$ delaying neurons also along the synapses $(l_i, c_{i1}), (l_i, c_{i2}), (c_{i3}, c_{i6})$. In this way, the paths from σ_{l_i} to $\sigma_{c_{i5}}$ and $\sigma_{c_{i6}}$ will last $\alpha + 1$ steps, as that from σ_{l_i} to σ_r.

Similarly for the SUB modules: we add $\alpha - 1$ delaying neurons along synapses $(l_i, c_{i1}), (l_i, c_{i2})$, and thus the spikes of σ_{l_i} reach all neurons $\sigma_r, \sigma_{ci1}, \sigma_{c_{i2}}$ at the same time, after α steps.

In this way, the system obtained in the end of all these operations is equivalent with Π and has the indegree 2. □

The previous construction can be combined with the construction used in [9] for decreasing the outdegree, hence we get the following combined normal form result:

Corollary 6. $N_2SNP(ind_2, oud_2) = NRE$.

What remains to investigate is the size (and the properties) of families $N_2SNP(ind_i, oud_j)$ for $(i, j) \in \{(1,1), (1,2), (2,1)\}$.

First, let us remark that $NFIN \subseteq N_2SNP(ind_1, oud_1)$: given a finite set $F = \{n_i \mid 1 \leq i \leq k\}$ of natural numbers, for the system

$$\Pi = (\{a\}, (2, \{a^2/a \to a; 0\} \cup \{a \to a; n_i - 1 \mid 1 \leq i \leq k\}), \emptyset, 1)$$

we have $N_2(\Pi) = F$ (we can consider that this system has the indegree and the outdegree zero, as no synapse appears in it).

Then, because the systems with outdegree one cannot increase the number of spikes from their neurons, it follows (as already observed in [13]) that $N_2SNP(ind_2, oud_1) \subseteq NREG$ (the system can be easily simulated by a finite automaton).

A similar result is valid also for the family $N_2SNP(ind_1, oud_2)$. A system with such indegree and outdegree has the synapse graph of a very particular form: a possible cycle, from which starts binary trees. If there is no cycle, then only the tree containing the output neuron is relevant (no other tree contributes to the computations which determine the output), and from it only the synapses going to the output neuron – hence we can reduce the tree to a line. If there is a cycle, and the output neuron is on it, then no tree is relevant. If there is a cycle and the output neuron is on a tree emerging from the cycle, then we can trim all trees different from the one containing the output neuron, as well as all branches of this tree which are not on the way from the cycle to the output neuron, or after the output neuron.

In conclusion, the graph is either a linear tree ended with the output neuron, or a cycle from which emerges a linear tree ended with the output neuron. In the first case, there are only a finite number of possible computations, hence the generated set of numbers is finite. In the second case, the number of spikes cannot increase in the neurons of the cycle, but it can increase in the neurons of the tree, because the cycle can repeatedly introduce spikes in the tree. However, if a neuron of the tree can ever use a rule, then its contents cannot increase unboundedly: after using

a spiking rule or a forgetting rule, the number of spikes in the neuron decreases. From the cycle, we get at most one spike in a step, hence the increase of the number of spikes is one by one; this means that when we reach again the number of spikes which enables the used rule, the rule is used again, decreasing the number of spikes. If a neuron never uses a rule, then it is useless, and can be removed from the system, and then we can remove also all neurons not linked to the output neuron. Consequently, again the contents of neurons is bounded, hence can be controlled by the states of a finite automaton.

We synthesize all these observations in the following theorem:

Theorem 7. *$NFIN \subseteq N_2SNP(ind_1, oud_1) \subseteq N_2SNP(ind_i, oud_j) \subseteq NREG$, for each $(i,j) \in \{(1,2),(2,1)\}$.*

We *conjecture* that all families $N_2SNP(ind_i, oud_j)$ from the previous result are equal to $NFIN$.

6. Closing Remarks

This overview was a quick one, meant only to let the reader have a general idea about spiking neural P systems, the basic notions and results. As suggested in the introduction, there are several other directions of research which were not mentioned above. For instance, an important issue concerns 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 which 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 [3] 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 [3] for `SAT`: an SN P system is constructed, depending on `SAT`, and it is initiated for a given instance γ by introducing spikes in $2nm$ neurons, where n is the number of variables and m is the number of clauses of γ; in four steps, the system decides whether or not γ 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.

A lot of open problems and research topics are formulated in the papers mentioned in the previous sections (and listed in the bibliography). Standard questions concern the optimality of the computational completeness results (number of neurons, rules, spikes consumed, forgotten or produced, etc.), characterizations of other families of numbers or of languages than those considered in Theorems 1, 2, 3, the passage to sets of vectors (with a series of results already reported in [11], [10]), combinations of the normal forms (or proofs that they cannot be combined without losing power). A long list of problems concerns the use of SN P systems for generating infinite sequences of bits or for processing such sequences – see [22].

In what concerns the indegree/outdegree normal forms, it remains as a research topic to see whether other graph-theoretic restrictions might be of interest for SN P systems. Cycle structure of the graphs and planarity could be studied. As regards planarity, it is interesting to note that most of the examples from [13] and [21] deal, indeed, with planar SN P systems, but this is not the case with the systems used in the proofs. In turn, the ADD, SUB, and OUTPUT modules from the proof of Theorem 5 are also planar, but their combination in the system is not necessarily so, because this depends on the relations between the instructions of the starting register machine.

The study of SN P systems is rather recent, but the related bibliography is already comprehensive and continuously growing. We believe that this area is worth investigating, not only because of the importance of neural computing based on spiking (one speaks about "neural computing of third generation" in this respect), but also because of the mathematical (computability) interest and the possible ground for initiating a study of complexity classes based on pre-computed resources. The reader is advised to watch [24] for future developments in this area.

References

[1] M. Cavaliere, R. Freund, A. Leitsch, Gh. Păun: Event-related outputs of computations in P systems. *Proc. Third Brainstorming Week on Membrane Computing*, Sevilla, 2005, RGNC Report 01/2005, 107–122.

[2] H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In [8], Vol. I, 169–194.

[3] H. Chen, M. Ionescu, T.-O. Ishdorj: On the efficiency of spiking neural P systems. In [8], Vol. I, 195–206.

[4] H. Chen, M. Ionescu, A. Păun, Gh. Păun, B. Popa: On trace languages generated by spiking neural P systems. In [8], Vol. I, 207–224, and *Proc. DCFS2006*, Las Cruces, NM, June 2006.

[5] H. Chen, T.-O. Ishdorj, Gh. Păun: Computing along the axon. In [8], Vol. I, 225–240.

[6] H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with extended rules. In [8], Vol. I, 241–265.

[7] W. Gerstner, W Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity.* Cambridge Univ. Press, 2002.

[8] M.A. Gutiérrez-Naranjo et al., eds.: *Proceedings of Fourth Brainstorming Week on Membrane Computing*, Febr. 2006, Fenix Editora, Sevilla, 2006.

[9] O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosik, S. Woodworth: Normal forms for spiking neural P systems. In [8], Vol. II, 105–136.

[10] O.H. Ibarra, S. Woodworth: Characterizations of some restricted spiking neural P systems. In *Pre-proceedins of Seventh Workshop on Membrane Computing, WMC7*, Leiden, The Netherlands, July 2006, 387–396.

[11] O.H. Ibarra, S. Woodworth, F. Yu, A. Păun: On spiking neural P systems and partially blind counter machines. In *Proceedings of Fifth Unconventional Computation Conference, UC2006*, York, UK, September 2006.

[12] M. Ionescu, A. Păun, Gh. Păun, M.J. Pérez-Jiménez: Computing with spiking neural P systems: Traces and small universal systems. In *Proceedings of 12th DNA Based Computing Conference, DNA12*, Seul, June 2006, 32–42.

[13] M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.

[14] I. Korec: Small universal register machines. *Theoretical Computer Science*, 168 (1996), 267–301.

[15] W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.

[16] W. Maass, C. Bishop, eds.: *Pulsed Neural Networks*, MIT Press, Cambridge, 1999.

[17] M. Minsky: *Computation – Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.

[18] A. Păun, Gh. Păun: Small universal spiking neural P systems. In [8], Vol. II, 213–234.

[19] Gh. Păun: *Membrane Computing – An Introduction*. Springer, Berlin, 2002.

[20] Gh. Păun: Languages in membrane computing. Some details for spiking neural P systems. In *Proceedings of Developments in Language Theory Conference, DLT 2006*, Santa Barbara, CA, June 2006, LNCS 4036, Springer, Berlin, 2006, 20–35.

[21] Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Spike trains in spiking neural P systems. *Intern. J. Found. Computer Sci.*, 17, 4 (2006), 975–1002.

[22] Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted 2005.

[23] G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*, 3 volumes. Springer-Verlag, Berlin, 1997.

[24] The P Systems Web Page: `http://psystems.disco.unimib.it`.