Computing with Spiking Neural P Systems: Traces and Small Universal Systems

Mihai Ionescu[1], Andrei Păun[2], Gheorghe Păun[3,4], and Mario J. Pérez-Jiménez[4]

[1] Research Group on Mathematical Linguistics
Universitat Rovira i Virgili
Pl. Imperial Tàrraco 1, 43005 Tarragona, Spain
armandmihai.ionescu@urv.net

[2] Department of Computer Science, Louisiana Tech University
Ruston, PO Box 10348, Louisiana, LA-71272 USA, and
Universidad Politécnica de Madrid – UPM, Faculdad de Informatíca
Campus de Montegancedo s/n, Boadilla del Monte
28660 Madrid, Spain
apaun@latech.edu

[3] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucharest, Romania
george.paun@imar.ro

[4] Department of Computer Science and AI, University of Sevilla
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, marper@us.es

Abstract. Recently, the idea of spiking neurons and thus of computing by spiking was incorporated into membrane computing, and so-called spiking neural P systems (abbreviated SN P systems) were introduced. Very shortly, in these systems neurons linked by synapses communicate by exchanging identical signals (spikes), with the information encoded in the distance between consecutive spikes. Several ways of using such devices for computing were considered in a series of papers, with universality results obtained in the case of computing numbers, both in the generating and the accepting mode; generating, accepting, or processing strings or infinite sequences was also proved to be of interest.

In the present paper, after a short survey of central notions and results related to spiking neural P systems (including the case when SN P systems are used as string generators), we contribute to this area with two (types of) results: (i) we produce small universal spiking neural P systems (84 neurons are sufficient in the basic definition, but this number is decreased to 49 neurons if a slight generalization of spiking rules is adopted), and (ii) we investigate the possibility of generating a language by following the trace of a designated spike in its way through the neurons.

1 Introduction

Spiking neural P systems (in short, SN P systems) were introduced in [6], with the motivation coming from two directions: the attempt of membrane computing to pass from cell-like architectures to tissue-like or neural-like architectures (see [15], [12]), and the intriguing possibility of encoding information in the duration of events, or in the interval of time elapsed between events, as vividly investigated in recent research in neural computing (of "third generation") [8], [9].

This double challenge led to a class of P systems based on the following simple ideas: let us use only one object, the symbol denoting a *spike*, and one-membrane cells (called *neurons*) which can hold any number of spikes; each neuron fires in specified conditions (after collecting a specified number of spikes) and then sends one spike along its axon; this spike passes to all neurons connected by a *synapse* to the spiking neuron (hence it is replicated into as many copies as many target neurons exist); between the moment when a neuron fires and the moment when it spikes, each neuron needs a time interval, and this time interval is the essential ingredient of the system functioning (the basic information carrier – with the mentioning that also the number of spikes accumulated in each moment in the neurons provides an important information for controlling the functioning of the system); one of the neurons is considered the output one, and its spikes provide the output of the computation. The sequence of time moments when spikes are sent out of the system is called a *spike train*. The rules for spiking take into account *all* spikes present in a neuron not only part of them, but not all spikes present in a neuron are consumed in this way; after getting fired and before sending the spike to its synapses, the neuron is idle (biology calls this the refractory period) and cannot receive spikes. There are also rules used for "forgetting" some spikes, rules which just remove a specified number of spikes from a neuron.

In the spirit of spiking neurons, as the result of a computation (not necessarily a halting one) in [6] one considers the number of steps elapsed between the first two spikes of the output neuron. Even in this restrictive framework, SN P systems turned out to be Turing complete, *able to compute all Turing computable sets of natural numbers*. This holds both in the generative mode (as sketched above, a number is computed if it represents the interval between the two consecutive spikes of the output neuron) and in the accepting mode (a number is introduced in the system in the form of the interval of time between the first two spikes entering a designated neuron, and this number is accepted if the computation halts). If a bound is imposed on the number of spikes present in any neuron during a computation, then *a characterization of semilinear sets of numbers is obtained*.

These results were extended in [13] to several other ways of associating a set of numbers with an SN P system: taking into account the interval between the first k spikes of each spike train, or all spikes, taking only alternately the intervals, or all of them, considering halting computations. Then, the spike train itself (the sequences of symbols 0, 1 describing the activity of the output neuron: we write 0 if no spike exits the system in a time unit and 1 if a spike is emitted) was considered as the result of a computation; the infinite case is investigated in [14], the finite one in [2]. A series of possibilities of handling infinite sequences of bits are discussed in [14], while morphic representations of regular and of recursively enumerable languages are found in [2]. The results from [2] are briefly recalled in Section 5 below.

In this paper we directly continue these investigations, contributing in two natural directions. First, the above mentioned universality results (the possibility to compute all Turing computable sets of numbers) do not give an estimation on the number of neurons sufficient for obtaining the universality. Which is the size of the smallest universal "brain" (of the form of an SN P system)? This is both a natural and important (from computer science and, also, from neuro-science point of view) problem, reminding the extensive efforts paid for finding small universal Turing machines – see, e.g., [16] and the references therein.

Our answer is rather surprising/encouraging: *84 neurons ensure the universality* in the basic setup of SN P systems, as they were defined in [6], while this number is decreased to 49 if slightly more general spiking rules are used (rules with the possibility to produce not only one spike, *but also two or more spikes at the same time* – such rules are called *extended*). The proof is based on simulating a small universal register machine from [7]. (The full details for the proof of these results about small universal SN P systems will be provided elsewhere – see [11].)

Extended rules are also useful when generating strings: we associate a symbol b_i with a step when the system outputs i spikes and in this way we obtain a string over an arbitrary alphabet, not only on the binary one, as in the case of standard rules. Especially flexible is the case when we associate the empty string with a step when no spike is sent out of the system we associate (that is, b_0 is interpreted as λ). Results from [3], concerning the power of extended SN P systems as language generators, are also recalled in Section 5.

Then, another natural issue is to bring to the SN P systems area a notion introduced for symport/antiport P systems in [5]: mark a spike and follow its path through the system, recording the labels of the visited neurons until either the marking disappears or the computation halts. Because of the very restrictive way of generating strings in this way, there are simple languages which cannot be computed, but, on the other hand, there are rather complex languages which can be obtained in this framework.

Due to space restrictions, we do not give full formal details in definitions and proofs (we refer to the above mentioned papers for that); such details are or will be available in separate papers to be circulated/announced through [19].

2 Formal Language Theory Prerequisites

We assume the reader to be familiar with basic language and automata theory, e.g., from [17] and [18], so that we introduce here only some notations and notions used later in the paper.

For an alphabet V, V^* denotes the set of all finite strings of symbols from V; the empty string is denoted by λ, and the set of all nonempty strings over V is denoted by V^+. When $V = \{a\}$ is a singleton, then we write simply a^* and a^+ instead of $\{a\}^*, \{a\}^+$. If $x = a_1a_2 \dots a_n$, $a_i \in V$, $1 \leq i \leq n$, then the mirror image of x is $mi(x) = a_n \dots a_2a_1$.

A morphism $h : V_1^* \longrightarrow V_1^*$ such that $h(a) \in \{a, \lambda\}$ for each $a \in V_1$ is called a projection, and a morphism $h : V_1^* \longrightarrow V_2^*$ such that $h(a) \in V_2 \cup \{\lambda\}$ for each $a \in V_1$ is called a weak coding.

If $L_1, L_2 \subseteq V^*$ are two languages, the left and right quotients of L_1 with respect to L_2 are defined by $L_2 \backslash L_1 = \{w \in V^* \mid xw \in L_1 \text{ for some } x \in L_2\}$, and respectively $L_1/L_2 = \{w \in V^* \mid wx \in L_1 \text{ for some } x \in L_2\}$. When the language L_2 is a singleton, these operations are called left and right derivatives, and denoted by $\partial_x^l(L) = \{x\}\backslash L$ and $\partial_x^r(L) = L/\{x\}$, respectively.

A Chomsky grammar is given in the form $G = (N, T, S, P)$, where N is the nonterminal alphabet, T is the terminal alphabet, $S \in N$ is the axiom, and P is the finite set of rules. For regular grammars, the rules are of the form $A \rightarrow aB, A \rightarrow a$, for some $A, B \in N, a \in T$.

We denote by FIN, REG, CF, CS, RE the families of finite, regular, context-free, context-sensitive, and recursively enumerable languages; by MAT we denote the family of languages generated by matrix grammars without appearance checking. The family of Turing computable sets of numbers is denoted by NRE (these sets are length sets of RE languages, hence the notation).

Let $V = \{b_1, b_2, \ldots, b_m\}$, for some $m \geq 1$. For a string $x \in V^*$, let us denote by $val_m(x)$ the value in base $m+1$ of x (we use base $m+1$ in order to consider the symbols $b_1, \ldots, b_m$ as digits $1, 2, \ldots, m$, thus avoiding the digit 0 in the left hand of the string). We extend this notation in the natural way to sets of strings.

All universality results of the paper are based on the notion of a register machine. Such a device – in the non-deterministic version – is a construct $M = (m, H, l_0, l_h, I)$, where m is the number of registers, H is the 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:

– $l_i : (\texttt{ADD}(r), l_j, l_k)$ (add 1 to register r and then go to one of the instructions with labels l_j, l_k non-deterministically chosen),
– $l_i : (\texttt{SUB}(r), l_j, l_k)$ (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),
– $l_h : \texttt{HALT}$ (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., [10]) that register machines generate all sets of numbers which are Turing computable.

A register machine can also be used as a number accepting device: we introduce a number n in some register r_0, we start working with the instruction with label l_0, and if the machine eventually halts, then n is accepted (we may also assume that all registers are empty in the halting configuration). Again, accepting register machines characterize NRE.

Furthermore, register machines can compute all Turing computable functions: we introduce the numbers $n_1, \ldots, n_k$ in some specified registers $r_1, \ldots, r_k$, we start with the instruction with label l_0, and when we stop (with the instruction with label l_h) the value of the function is placed in another specified register, r_t, with all registers different from r_t being empty. Without loss of generality we may assume that $r_1, \ldots, r_k$ are the first k registers of M, and then the result of the computation is denoted by $M(n_1, \ldots, n_k)$.

In both the accepting and the computing case, the register machines can be *deterministic*, i.e., with the ADD instructions of the form $l_i : (\texttt{ADD}(r), l_j)$ (add 1 to register r and then go to the instruction with label l_j).

In the following sections, when comparing the power of two language generating/accepting devices the empty string λ is ignored.

3 Spiking Neural P Systems

We give here the basic definition we work with, introducing SN P systems in the form considered in the small universal SN P systems, hence computing functions (which, actually, covers both the generative and accepting cases).

A computing *spiking neural membrane system* (abbreviated SN P system), of degree $m \geq 1$, is a construct of the form

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

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 in σ_i;
   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[1] over a, $c \geq 1$, and $d \geq 0$;
      (2) $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from R_i, we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses* between neurons);
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons of Π.

[1] The regular language defined by E is denoted by $L(E)$.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \rightarrow a; d \in R_i$ can be applied. This means consuming (removing) c spikes (thus only $k - c$ remain in σ_i), the neuron is fired, and it produces a spike after d time units (as usual in membrane computing, 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, etc. 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* (this corresponds to the refractory period from neurobiology), 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). In the step $t + d$, the neuron spikes and becomes again open, so that it can receive spikes (which can be used starting with the step $t + d + 1$).

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

If a rule $E/a^c \rightarrow a; d$ of type (1) has $E = a^c$, then we will write it in the following simplified form: $a^c \rightarrow a; d$. If all spiking rules are of this form, then the system is said to be *finite* (it can handle only a bounded number of spikes in each of its neurons).

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, $E_1/a^{c_1} \rightarrow a; d_1$ and $E_2/a^{c_2} \rightarrow a; d_2$, 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 in that case, only one of them is chosen non-deterministically. Note however that, by definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other (the system is synchronized).

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 the neuron, more precisely, by 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 $f : \mathbf{N}^k \longrightarrow \mathbf{N}$, we introduce k natural numbers $n_1, \ldots, n_k$ in the system by "reading" from the environment a binary sequence $z = 0^b 1 0^{n_1 - 1} 1 0^{n_2 - 1} 1 \ldots 1 0^{n_k - 1} 1 0^f$, for some $b, f \geq 0$; 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 train spike of the form $0^{b'} 1 0^{r-1} 1 0^{f'}$, for some $b', f' \geq 0$ and with $r = f(n_1, \ldots, n_k)$.

If we use an SN P system in the generative mode, then no input neuron is considered, hence no input is taken from the environment; we start from the initial configuration and the distance between the first two spikes of the output neuron (or other numbers, see the discussion in the Introduction) is the result of the computation. Dually, we can ignore the output neuron, we input a number in the system as the distance between two spikes entering the input neuron, and if the computation halts, then the number is accepted.

We do not give here examples, because in the next section we show the four basic modules of our small universal SN P system.

4 Two Small Universal SN P Systems

In both the generating and the accepting case, SN P systems are universal, they compute the Turing computable sets of numbers. The proofs from [6], [13] are based on simulating register machines, which are known to be equivalent to Turing machines when computing (generating or accepting) sets of numbers, [10]. In [7], the register machines are used for computing functions, with the universality defined as follows. Let $(\varphi_0, \varphi_1, \ldots)$ be a fixed admissible enumeration of the set of unary partial recursive functions. A register machine M_u is said to be universal if there is a recursive function g such that for all natural numbers x, y we have $\varphi_x(y) = M_u(g(x), y)$. In [7], the input is introduced in registers 1 and 2, and the result is obtained in register 0 of the machine.

$l_0 : (\mathtt{SUB}(1), l_1, l_2)$,	$l_1 : (\mathtt{ADD}(7), l_0)$,
$l_2 : (\mathtt{ADD}(6), l_3)$,	$l_3 : (\mathtt{SUB}(5), l_2, l_4)$,
$l_4 : (\mathtt{SUB}(6), l_5, l_3)$,	$l_5 : (\mathtt{ADD}(5), l_6)$,
$l_6 : (\mathtt{SUB}(7), l_7, l_8)$,	$l_7 : (\mathtt{ADD}(1), l_4)$,
$l_8 : (\mathtt{SUB}(6), l_9, l_0)$,	$l_9 : (\mathtt{ADD}(6), l_{10})$,
$l_{10} : (\mathtt{SUB}(4), l_0, l_{11})$,	$l_{11} : (\mathtt{SUB}(5), l_{12}, l_{13})$,
$l_{12} : (\mathtt{SUB}(5), l_{14}, l_{15})$,	$l_{13} : (\mathtt{SUB}(2), l_{18}, l_{19})$,
$l_{14} : (\mathtt{SUB}(5), l_{16}, l_{17})$,	$l_{15} : (\mathtt{SUB}(3), l_{18}, l_{20})$,
$l_{16} : (\mathtt{ADD}(4), l_{11})$,	$l_{17} : (\mathtt{ADD}(2), l_{21})$,
$l_{18} : (\mathtt{SUB}(4), l_0, l_h)$,	$l_{19} : (\mathtt{SUB}(0), l_0, l_{18})$,
$l_{20} : (\mathtt{ADD}(0), l_0)$,	$l_{21} : (\mathtt{ADD}(3), l_{18})$,
$l_h : \mathtt{HALT}$.	

Fig. 1. The universal register machine from [7]

The constructions from [6] do not provide a bound on the number of neurons, but such a bound can be found if we start from a specific universal register machine. We will use here the one with 8 registers and 23 instructions from [7] – for the reader convenience, this machine is recalled in Figure 1, in the notation and the setup introduced in the previous section.

Theorem 1. *There is a universal SN P system with 84 neurons.*

Proof. (Outline) We follow the way used in [6] to simulate a register machine by an SN P system. This is done as follows: neurons are associated with each register (r) and with each label (l_i) of the machine; if a register contains a number n, then the associated neuron will contain $2n$ spikes; modules as in Figures 2 and 3 are associated with the ADD and the SUB instructions (each of these modules contains two neurons – with primed labels – which do not correspond to registers and labels of the simulated machine).

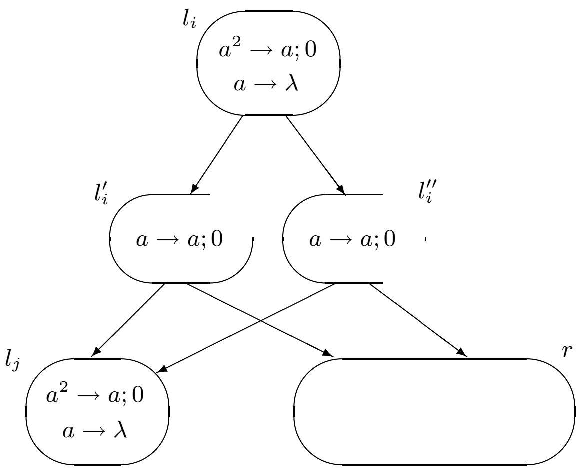

Fig. 2. Module ADD (simulating $l_i : (\mathtt{ADD}(r), l_j)$)

The work of the system is triggered by introducing two spikes in the neuron σ_{l_0} (associated with the starting instruction of the register machine). In general, the simulation of an ADD or SUB instruction starts by introducing two spikes in the neuron with the instruction label. We do not describe here in detail the (pretty transparent) way the modules from Figures 2 and 3 work – the reader can consult [6] in this respect.

Starting with neurons σ_1 and σ_2 already loaded with $2g(x)$ and $2y$ spikes, respectively, and introducing two spikes in neuron σ_{l_0}, we can compute in our system in the same way as M_u; if the computation halts, then neuron σ_0 will contain $2\varphi_x(y)$ spikes. What remains to do is to construct input and output modules, for reading a sequence of bits and introducing the right number of spikes in the neurons corresponding to registers 1 and 2, and, in the end of the computation, to output the contents of register 0. Modules of these types are given in Figures 4, 5, having seven and two additional neurons, respectively.

After this direct construction, we get a system with 91 neurons (9 for the registers of the starting register machine – one further register is necessary for technical reasons, 25 for its labels, 24×2 for the ADD and SUB instructions, 7 in the input module, and 2 in the output module). However, some "code optimization" is possible, based on certain properties of the register machine from

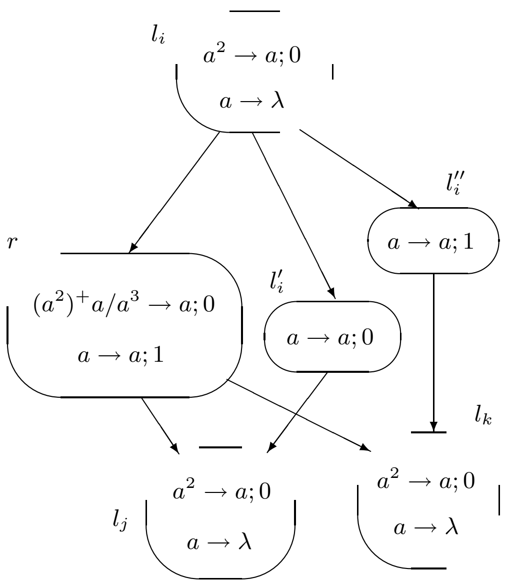

Fig. 3. Module SUB (simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$)

[7] (for instance, consecutive ADD instructions can be simulated by a specific module, smaller than two separate ADD modules); we skip the technical details and we only mention that the final SN P system will contain 84 neurons.

This is a small number (a small "brain", compared to the human one; it would be nice to know where in the evolution scale there are animals with about 84 neurons in their brain), but we do not know whether it is optimal or not. Anyway, we believe that in the previous setup, we cannot significantly decrease the number of neurons from a universal SN P system.

However, we can do better starting from the following observation. In many modules mentioned above we need pairs of intermediate neurons for duplicating the spike to be transmitted further (this is the case for neurons $\sigma_{l_i'}, \sigma_{l_i''}$ in Figure 2), and this suggests to consider a slight extension of the rules of SN P systems: to allow spiking rules of the form $E/a^c \to a^p; d$, where all components are as usual, and $p \geq 1$. The meaning is that c spikes are consumed and p spikes are produced. To be "realistic", we impose the restriction $c \geq p$ (the number of produced spikes is not larger than the number of consumed spikes).

Theorem 2. *There is a universal SN P system with 49 neurons, using rules of the form $E/a^c \to a^p; 0$, with $p \geq 1$.*

(Note that the delay is zero in the rules of the extended form used in the theorem.) As above, we do not know whether this result is optimal, but we again believe that it cannot be significantly improved (without, maybe, changing the definition of SN P systems in an essential way).

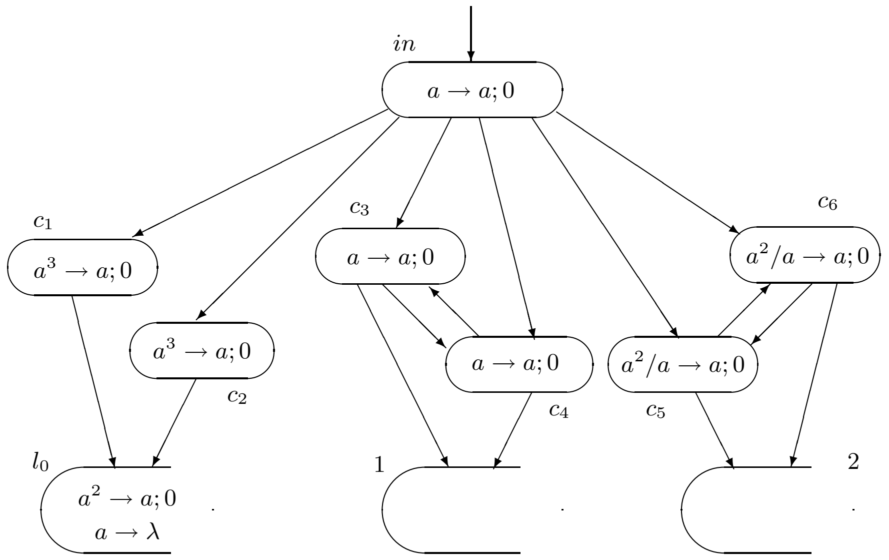

Fig. 4. Module INPUT

5 SN P Systems as String Generators

Following [2] we can also consider as the result of a computation the spike train itself, thus associating a language with an SN P system. Specifically, like in [2], we can consider the language $L_{bin}(\Pi)$ of all binary strings associated with halting computations in Π: the digit 1 is associated with a step when one or more spikes exit the output neuron, and 0 is associated with a step when no spike is emitted by the output neuron. We denote $B = \{0, 1\}$.

Because (in the case of extended systems) several spikes can exit at the same time, we can also work on an arbitrary alphabet: let us associate the symbol b_i with a step when the output neuron emits i spikes. We have two cases: interpreting b_0 (hence a step when no spike is emitted) as a symbol or as the empty string. In the first case we denote the generated language by $L_{res}(\Pi)$ (with "res" coming from "restricted"), in the latter one we write $L_\lambda(\Pi)$.

The respective families are denoted by $L_\alpha SN^eP_m(rule_k, cons_p, prod_q)$, where $\alpha \in \{bin, res, \lambda\}$ and parameters m, k, p, q are as above. We omit the superscript e and the parameter $prod_q$ when working with standard rules (in this case we always have $q = 1$).

We recall from [2] the following results:

Theorem 3. (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 with restricted rules, but for any $L \in FIN$, $L \subseteq B^+$, we have $L\{1\} \in L_{bin}SNP_1(rule_*, cons_*)$, and if $L = \{x_1, x_2, \ldots, x_n\}$, then we also have $\{0^{i+3}x_i \mid 1 \leq i \leq n\} \in L_{bin}SNP_*(rule_*, cons_*)$.*

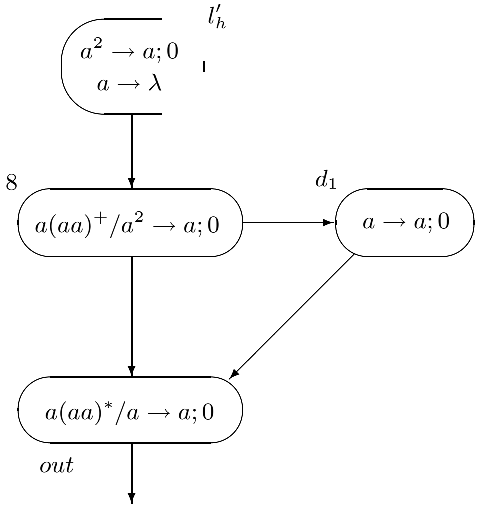

Fig. 5. Module OUTPUT

(ii) *The family of languages generated by finite non-extended 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) $L_{bin}SNP_*(rule_*, cons_*) \subset REC$*, but for every alphabet* $V = \{b_1, b_2, \ldots, b_s\}$ *there are 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 non-extended 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, and this again naturally suggests the idea of extended rules, with the possibility of having λ as output in steps when no spike exits the system. As we will see immediately, this possibility considerably enlarges the generated families of languages.

The next results were obtained in [3], as counterparts of the results from Theorem 3; as expected, the extended rules are useful, the obtained families of languages are larger, and finite, regular, and recursively enumerable can be directly obtained, without additional symbols and squeezing mechanisms.

We consider at the same time both the restricted case (with b_0 associated with a step when no spike is sent out) and the non-restricted one (with b_0 interpreted as λ); V is the alphabet $\{b_1, \ldots, b_s\}$:

Theorem 4. (i) $FIN = L_\alpha SN^eP_1(rule_*, cons_*, prod_*), \alpha \in \{res, \lambda\}$*, and this result is sharp, because* $L_{res}SN^eP_2(rule_2, cons_3, prod_3)$ *contains infinite languages.*

(ii) *If* $L \in REG$, *then* $\{b_0\}L \in L_{res}SN^eP_4(rule_*, cons_*, prod_*)$ *and* $L\{b_0\} \in L_{res}SN^eP_3(rule_*, cons_*, prod_*)$, *but there are minimal linear languages which are not in the family* $L_{res}SN^eP_*(rule_*, cons_*, prod_*)$.

(iii) $L_\lambda SN^eP_2(rule_*, cons_*, prod_*) \subseteq REG \subset L_\lambda SN^eP_3(rule_*, cons_*, prod_*)$; *the second inclusion is proper, because* $L_\lambda SN^eP_3(rule_4, cons_4, prod_2)$ *contains non-regular languages; actually, the family* $L_\lambda SN^eP_3(rule_3, cons_6, prod_4)$ *contains non-semilinear languages.*

(iv) $RE = L_\lambda SN^eP_*(rule_*, cons_*, prod_*)$.

It is an open problem to find characterizations (even only representations) of other families of languages in the Chomsky hierarchy.

6 Following the Traces of Spikes

We have seen above that SN P systems can be used also for generating or accepting languages, and even infinite sequences [14], by just taking the spike trains as generated strings/sequences. Here we consider yet another idea for defining a language, taking into account the traces of a distinguished spike through the system. This is a direct counterpart of trace languages from [5], and also has some similarity with the idea of "computing by observing", as recently considered in [1].

Specifically, in the initial configuration of the system we "mark" one spike from a specified neuron – the intuition is that this spike has a "flag" – and we follow the path of this flag during the computation, *recording the labels of the neurons where the flag is present in the end of each step.* Actually, for neuron σ_i we consider the symbol b_i in the trace string.

The previous definition contains many delicate points which need clarifications – and we use a simple example to do this.

Assume that in neuron σ_i we have three spikes, one of them marked; we write aaa' to represent them. Assume also that we have a spiking rule $aaa/aa \to a; 0$. When applied, this rule consumes two spikes, one remains in the neuron and one spike is produced and sent along the synapses going out of neuron σ_i. Two cases already appear: the marked spike is consumed or not. If not consumed, then it remains in the neuron. If consumed, then the flag passes to the produced spike. Now, if there are two or more synapses going out of neuron σ_i, then again we can have a branching: only one spike is marked, hence only on one of the synapses (i, j), non-deterministically chosen, we will transmit a marked spike. If σ_j is an open neuron, then the marked spike ends in this neuron. If σ_j is a closed neuron, then the marked spike is lost, and the same happens if the marked spike exits in the environment. Anyway, if the marked spike is consumed, at the end of this step it is no longer present in neuron i; it is in neuron σ_j if $(i, j) \in syn$ and neuron σ_j is open, or it is removed from the system in other cases.

Therefore, if in the initial configuration of the system neuron σ_i contains the marked spike, then the trace can start either with b_i (if the marked spike is not consumed) or with b_j (if the marked spike was consumed and passed to neuron σ_j); if the marked spike is consumed and lost, then we generate the empty string, which is ignored in our considerations. Similarly in later steps.

If the rule used is of the form $aaa/aa \to a; d$, for some $d \geq 1$, and the marked spike is consumed, then the newly marked spike remains in neuron σ_i for d steps, hence the trace starts/continues with b_i^d. Similarly, if no rule is used in neuron σ_i for k steps, then the trace records k copies of b_i.

If a forgetting rule is used in the neuron where the marked spike is placed, then the trace string stops (and no symbol is recorded for this step).

Therefore, when considering the possible branchings of the computation, we have to take into account the non-determinism not only in using the spiking rules, but also in consuming the marked spike and in sending it along one of the possible synapses.

The previous discussion has, hopefully, made clear what we mean by *recording the labels of the neurons where the flag is present in the end of each step*, and why choosing the end of a step and not the beginning: in the latter case, all traces would start with the same symbol, corresponding to the input neuron, which is a strong – and artificial – restriction.

Anyway, we take into account only halting computations: irrespective whether or not a marked spike is still present in the system, the computation should halt (note that it is possible that the marked spike is removed and the computation still continues for a while – but this time without adding further symbols to the trace string).

For an SN P system Π we denote by $T(\Pi)$ the language of all strings describing the traces of the marked spike in all halting computations of Π. Then, we denote by $TSNP_m(rule_k, cons_p, forg_q)$ the family of languages $T(\Pi)$, generated by systems Π with at most m neurons, each neuron having at most k rules, each of the spiking rules consuming at most p spikes, and each forgetting rule removing at most q spikes. As usual, a parameter m, k, p, q is replaced with $*$ if it is not bounded.

We pass now to investigating the relationship with the families of languages from Chomsky hierarchy, starting with a counterexample result (whose simple proof is omitted).

Lemma 1. *There are singleton languages which are not in $TSNP_*(rule_*, cons_*, forg_*)$.*

Theorem 5. *The family of trace languages generated by SN P systems by means of computations with a bounded number of spikes present in their neurons is strictly included in the family of regular languages.*

The inclusion follows from the fact that the transition diagram associated with the computations of an SN P system which use a bounded number of spikes is finite and can be interpreted as the transition diagram of a finite automaton. The fact that the inclusion is proper is a consequence of Lemma 1.

As expected, also non-regular languages can be generated – as well as much more complex languages.

Theorem 6. *Every unary language $L \in RE$ can be written in the form $L = h(L') = (b_1^* \backslash L') \cap b_2^*$, where $L' \in TSNP_*(rule_*, cons_*, forg_*)$ and h is a projection.*

Proof. (Sketch) This result is a consequence of the fact that SN P systems can simulate register machines. Specifically, starting from a register machine M, we construct an SN P system Π which halts its computation with $2n$ spikes in a specified neuron σ_{out} if and only if n can be generated by the register machine M; in the halting moment, a neuron σ_{l_h} of Π associated with the label of the halting instruction of M gets two spikes and fires. The neuron σ_{out} contains no rule used in the simulation of M (the corresponding register is only incremented, but never decremented – see the details of the construction from [6], as well as Figures 2 and 3).

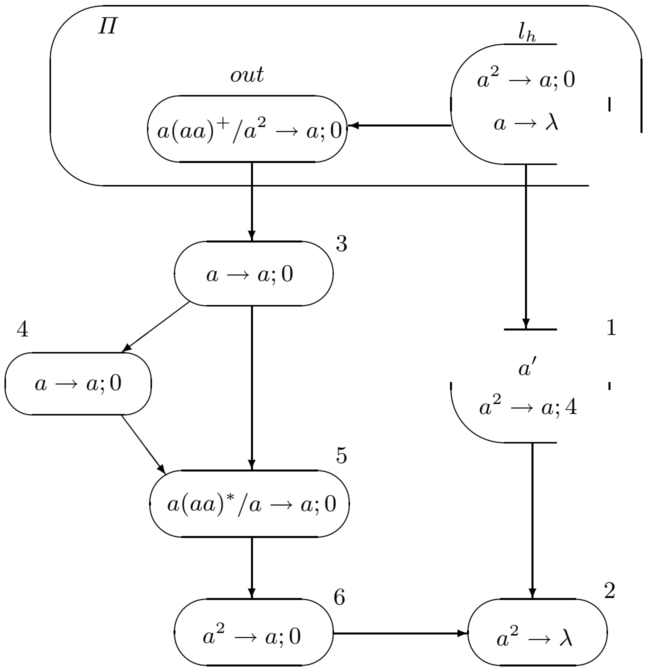

Fig. 6. The SN P system from the proof of Theorem 6

Now, consider a language $L \subseteq b_2^*, L \in RE$. There is a register machine M such that $n \in N(M)$ if and only if $b_2^n \in L$. Starting from such a machine M, we construct the system Π as in [6], having the properties described above. We append to the system Π six more neurons, as indicated in Figure 6. There is a marked spike in neuron σ_1, and it will stay here during all the simulation of M. In the moment when neuron σ_{l_h} of Π spikes, its spike goes both to neuron σ_{out} and to neuron σ_1.

Neurons $\sigma_3, \sigma_4, \sigma_5, \sigma_6$ send a spike to neuron σ_2 only when neuron σ_{out} has finished its work (this happens after n steps of using the rule $a(aa)^+/a^2 \to a; 0$, for $2n$ being the contents of neuron σ_{out} in the moment when neuron σ_{l_h} spikes).

The marked spike leaves neuron σ_1 four steps after using the rule $a^2 \rightarrow a; 4$, hence five steps after the spiking of neuron σ_{l_h}. This means that the marked spike waits in neuron σ_2 exactly n steps. When the spike of neuron σ_6 reaches neuron σ_2, the two spikes present here, the marked one included, are forgotten.

Thus, the traces of the marked spike are of the form $b_1^r b_2^n$, for some $r \geq 1$ and $n \in N(M)$. By means of the left derivative with the regular language b_1^* we can remove prefixes of the form b_1^k and by means of the intersection with b_2^* we ensure that the maximal prefix of this form is removed. Similarly, the projection $h : \{b_1, b_2\}^* \longrightarrow \{b_1, b_2\}^*$ defined by $h(b_1) = \lambda$, $h(b_2) = b_2$, removes all occurrences of b_1. Consequently, $L = (b_1^* \backslash T(\Pi)) \cap b_2^* = h(T(\Pi))$.

Corollary 1. *The family $TSNP_*(reg_*, cons_*, forg_*)$ is incomparable with each family of languages FL which contains the singleton languages, is closed under left derivative with regular languages and intersection with regular languages, and does not contain all unary recursively enumerable languages.*

Families FL as above are FIN, REG, CF, CS, MAT etc.

7 Final Remarks

After a brief informal survey of main results related to SN P systems as number generating or accepting devices, we have produced small universal SN P systems (with 84 and 49 neurons, depending on the type of spiking rules used), and we have introduced and preliminarily investigated the possibility of using SN P systems as language generators by following the trace of a marked spike across the neurons. Many topics remain open for further research, and other suggestions from biology are worth considering.

Acknowledgements

Thanks are due to Takashi Yokomori for many discussions about spiking neural P systems, as well as for many comments about a previous version of this paper.

The authors gratefully acknowledge the following (partial) support of their research. M. Ionescu: Programa Nacional para la Formación del Profesorado Universitario from the Spanish Ministry of Education. A. Păun: LA BoR RSC grant LEQSF (2004-07)-RD-A-23, and NSF Grants IMR-0414903 and CCF-0523572. Gh. Păun and M.J. Pérez-Jiménez: Project TIN2005-09345-C03-01 of Ministerio de Educación y Ciencia of Spain, cofinanced by FEDER funds, and Project of Excellence TIC 581 of Junta de Andalucia.

References

1. M. Cavaliere, P. Leupold: Evolution and observation – A new way to look at membrane systems. In *Membrane Computing. Intern. Workshop'WMC 2003, Tarragona, Spain, July 2003. Revised Papers* (C. Martin-Vide, G. Mauri, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 2933, Springer, Berlin, 2004, 70–87.

2. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In *Proc. of Fourth. Brainstorming Week on Membrane Computing*, Sevilla, 2006, vol. I, 169–193 (also available at [19]).
3. H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with extended rules. In *Proc. of Fourth. Brainstorming Week on Membrane Computing*, Sevilla, 2006, vol. I, 241–266 (also available at [19]).
4. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosik, S. Woodworth: Normal forms for spiking neural P systems. In *Fourth Brainstorming Week on Membrane Computing*, Febr. 2006, Fenix Editora, Sevilla, 2006, vol. II, 105–136 (also available at [19]).
5. M. Ionescu, C. Martin-Vide, A. Păun, Gh. Păun: Membrane systems with symport/antiport: (unexpected) universality results. In *Proc. 8th International Meeting of DNA Based Computing* (M. Hagiya, A. Ohuchi, eds.), Japan, 2002, 151–160.
6. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
7. I. Korec: Small universal register machines. *Theoretical Computer Science*, 168 (1996), 267–301.
8. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
9. W. Maass, C. Bishop, eds.: *Pulsed Neural Networks*, MIT Press, Cambridge, 1999.
10. M. Minsky: *Computation – Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, NJ, 1967.
11. A. Păun, Gh. Păun: Small universal spiking neural P systems. *BioSystems*, to appear.
12. Gh. Păun: *Membrane Computing – An Introduction.* Springer-Verlag, Berlin, 2002.
13. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Spike trains in spiking neural P systems. *Intern. J. Found. Computer Sci.*, to appear (also available at [19]).
14. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted, 2006.
15. Gh. Păun, Y. Sakakibara, T. Yokomori: P systems on graphs of restricted forms. *Publicationes Mathematicae Debrecen*, 60 (2002), 635–660.
16. Y. Rogozhin: Small universal Turing machines. *Theoretical Computer Science*, 168 (1996), 215–240.
17. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*, 3 volumes. Springer-Verlag, Berlin, 1997.
18. A. Salomaa: *Formal Languages.* Academic Press, New York, 1973.
19. The P Systems Web Page: `http://psystems.disco.unimib.it`.