Spiking Neural dP Systems

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

[1] University of Piteşti
Str. Târgu din Vale, nr. 1, 110040 Piteşti
Romania
armandmihai.ionescu@gmail.com
[2] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucharest, Romania
[3] Research Group on Natural Computing
Department of Computer Science and AI
University of Sevilla
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, marper@us.es
[4] Department of Mathematics, School of Education
Waseda University, 1-6-1 Nishi-waseda, Shinjuku-ku
Tokyo 169-8050, Japan
yokomori@waseda.jp

Summary. We bring together two topics recently introduced in membrane computing, the much investigated spiking neural P systems (in short, SN P systems), inspired from the way the neurons communicate through spikes, and the dP systems (distributed P systems, with components which "read" strings from the environment and then cooperate in accepting their concatenation). The goal is to introduce SN dP systems, and to this aim we first introduce SN P systems with the possibility to input, at their request, spikes from the environment; this is done by so-called *request rules*. A preliminary investigation of the obtained SN dP systems (they can also be called *automata*) is carried out. As expected, request rules are useful, while the distribution in terms of dP systems can handle languages which cannot be generated by usual SN P systems. We always work with extended SN P systems; the non-extended case, as well as several other natural questions remain open.

1 Introduction

We combine here two ideas recently considered in membrane computing, the spiking neural P systems (in short, SN P systems) introduced in [9], and the dP systems introduced in [13].

For the reader's convenience, we shortly recall that an SN P system consists of a set of neurons placed in the nodes of a graph and sending signals (spikes)

along synapses (edges of the graph), under the control of firing rules. One neuron is designated as the *output* neuron of the system and its spikes can exit into the environment, thus producing a *spike train*. Two main kinds of outputs can be associated with a computation in an SN P system: a set of numbers, obtained by considering the number of steps elapsed between consecutive spikes which exit the output neuron, and the string corresponding to the sequence of spikes which exit the output neuron. This sequence is a binary one, with 0 associated with a step when no spike is emitted and 1 associated with a step when a spike is emitted.

Actually, we use *extended* SN P systems, that is, we allow rules of the form $E/a^c \rightarrow a^p$, with the following meaning: if the content of the neuron is described by the regular expression E, then c spikes are consumed and p are produced and sent to the neurons to which there exist synapses leaving the neuron where the rule is applied (more precise definitions will be given in Section 3). In this way, strings over arbitrary alphabets can be obtained: if i spikes are sent out at the same time, then we say that the symbol b_i was generated. The languages generated by SN P systems in this way were investigated in [2] and [3]; see also [4].

In turn, a dP systems consists of several "modules" (usual P systems), which communicate among them by means of antiport rules like in tissue P systems. When only symport/antiport rules are used inside modules, then we can define a language accepted by such a system: each component takes from the environment sequences of symbols, they work separately and communicate through antiport rules and, if the computation halts, then the concatenation of the input strings is accepted. See [6], [13], [15], [16] for a series of results about such machineries; in particular, [6] investigates the language families characterized by P and dP automata, their relationships and place in Chomsky hierarchy.

There is an apparent difference between the two classes of P systems: SN P systems *generate* strings, while dP systems *accept* them. There were considered SN P systems also working in the accepting mode, with a spike train (a number is encoded as the distance between two consecutive spikes) introduced in a specified neuron of the system in the first steps of the computation. However, a more natural way to proceed, more similar to the way the P automata take symbols from the environment, is to consider a new type of rules in SN P systems, able to take spikes from the environment. We consider such rules of a form rather similar to spiking rules, namely, of the form $E/\lambda \leftarrow a^r$: if the contents of the neuron is described by the regular expression E, then r spikes are brought from the environment. Such a rule can be applied as a usual spiking rule (its use lasts one time unit, with the spikes brought from the environment being added to the contents of the neuron and ready to be used in the next step).

Now, the definition of an SN P system with request rules is rather natural: the neurons which contain request rules are supposed as having a "synapse" also with the environment such that they can take spikes from the environment, depending on the number of spikes they contain. The computation proceeds as usual, starting from the initial configuration. Both input and output spike trains can be associated with the neurons linked with the environment. Also the step to SN dP systems

(we can call them *automata*) is natural: take "modules" (or components) consisting of neurons, with only one neuron of each component also having a synapse with the environment (hence able to take an input); each component can be linked with another component through synapses among their neurons (for simplicity, we consider only the case when at most one synapse is available in each direction). The components take strings from the environment and the concatenation of these strings is accepted if the computation of the system halts.

Many questions arise in this framework. Compare usual SN P systems with SN P systems having request rules, both in the generative and the accepting mode. Do the additional facilities provided by the request rules help, e.g., from the point of view of the descriptional complexity? What about imposing a bound on the environment (considering that in the beginning it only contains a given number of spikes, and further spikes can be there only if the system sends spikes out)? More interesting: which is the power of SN dP systems? As expected, it is larger than that of SN P systems (with the mentioning that we compare languages *accepted* by SN dP systems with languages *generated* by SN P systems).

In this context, we introduce a refinement in the way of defining languages associated with (extended) SN P systems. In between the restricted (in each step, one symbol is produced) and the non-restricted case we can consider the languages generated/accepted in the k-restricted way: between reading or producing two symbols which are considered in the string, the system can work at most k steps without reading or sending symbols out.

2 Formal Language and Automata Theory Prerequisites

We assume the reader to be familiar with basic language and automata theory, e.g., from [18] and [19], 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 $x = a_1a_2 \ldots a_n$, $a_i \in V$, $1 \leq i \leq n$, then $mi(x) = a_n \ldots a_2a_1$.

We denote by REG, RE the families of regular and recursively enumerable languages. The family of Turing computable sets of numbers is denoted by NRE (these sets are length sets of RE languages, hence the notation).

In most universality proofs in membrane computing, in particular, in the SN P systems area, one uses register machines (several registers can contain natural numbers; they are increased by one or decreased by one – the latter operation only after checking whether the number stored in the register is different from zero –, by means of labeled instructions; if the computation starting in an initial label with all registers empty halts by reaching a special halting label, then the number stored in the halting configuration by the first register is said to be computed/generated by that computation.

Because our SN P systems can read at the same time several spikes, an operation which corresponds to reading a symbol from an alphabet with several elements, we will use in the universality proof a device more adequate to this case: *counter automata with an input tape*. The version we choose is one with as simple as possible instructions: check for zero, increment, decrement (by one in both cases), read a symbol from the input tape, halt. Formally, such a device is a construct $M = (n, V, H, l_0, l_h, I)$, where n is the number of registers/counters, V is the alphabet of the input tape, H is the set of labels, each one uniquely associated with an instruction, l_0 is the initial label, l_h is the halt label, and I is the set of instructions of the following forms:

1. $(l_i : \texttt{check}(r), l_j, l_k)$: when label l_i is reached, the contents of register r is compared to zero; if the register is empty, then the next label is l_j, if the contents of the register is strictly positive, then the next label is l_k;
2. $(l_i : \texttt{add}(r), l_j)$: add 1 to the contents of register r and pass from label l_i to label l_j;
3. $(l_i : \texttt{sub}(r), l_j)$: subtract 1 from the contents of register r and pass from label l_i to label l_j (the operation is supposed to be possible, meaning that previously the register was checked for zero by an instruction of the first type);
4. $(l_i : \texttt{read}(b), l_j)$: read the symbols $b \in V$ from the tape and pass from label l_i to label l_j;
5. $l_h : \texttt{halt}$: when reaching l_h, the computation halts; this is the only instruction labeled by l_h.

Without loss of the generality, we may assume that in all instructions the involved labels are mutually different (this can be easily achieved by introducing intermediate labels, involved in additional instructions performing “dummy” operations, of the form “add 1 to register r, then subtract 1 from register r”.

We start with label l_0 (by applying the instruction with label l_0), with all counters empty (storing the number 0), with the “reading head” in front of the first symbol of the input tape, where a string is written. We proceed as specified by the instructions, under the control of labels. The process is deterministic and the only branching is based on the check for zero of the registers. When the label l_h (hence the instruction $l_h : \texttt{halt}$) is reached, the computation stops and the sequence of symbols read from the input tape is the string accepted by the counter machine. The language of all such strings is denoted by $L(M)$.

It is known that counter machines (with a small number of counters, but this is not of interest below) with an input tape can recognize all recursively enumerable languages. Details (variants and proofs) can be found in several places: [11], [5], [7].

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 with Request Rules

We directly introduce the type of SN P systems we investigate in this paper; the reader can find details about the standard definition in [9], [14], [2], etc.

An *(extended) spiking neural P system* (abbreviated as *SN P system*) with *request rules*, of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \dots, \sigma_m, syn, i_0),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \dots, \sigma_m$ 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 forms
      (i) $E/a^c \to a^p$, where E is a regular expression over a and $c \geq p \geq 1$ (spiking rules);
      (ii) $E/\lambda \leftarrow a^r$, where E is a regular expression over a and $r \geq 1$ (request rules);
      (iii) $a^s \to \lambda$, with $s \geq 1$ (forgetting rules) such that there is no rule $E/a^c \to a^p$ of type (i) or $E/\lambda \leftarrow a^r$ of type (ii) with $a^s \in L(E)$;
3. $syn \subseteq \{1, 2, \dots, m\} \times \{1, 2, \dots, m\}$ with $i \neq j$ for each $(i, j) \in syn$, $1 \leq i, j \leq m$ (*synapses* between neurons);
4. $i_0 \in \{1, 2, \dots, m\}$ indicates the *output neuron* (σ_{i_0}) of the system.

A rule $E/a^c \to a^p$ is applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule can *fire*, and its application means consuming (removing) c spikes (thus only $k - c$ remain in σ_i) and producing p spikes, which will exit immediately the neuron. A rule $E/\lambda \leftarrow a^r$ is used if the neuron contains k spikes and $a^k \in L(E)$; no spike is consumed, but r spikes are added to the spikes in σ_i. In turn, a rule $a^s \to \lambda$ is used if the neuron contains exactly s spikes, which are removed ("forgotten"). A global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized.

If a rule $E/a^c \to a^p$ has $E = a^c$, then we will write it in the simplified form $a^c \to a^p$.

The spikes emitted by a neuron σ_i go to all neurons σ_j such that $(i, j) \in syn$, i.e., if σ_i has used a rule $E/a^c \to a^p$, then each neuron σ_j receives p spikes. The spikes produced by a rule $E/\lambda \leftarrow a^r$ are added to the spikes in the neuron and to those received from other neurons, hence they are counted/used in the next step of the computation.

If several rules can be used at the same time, then the one to be applied is chosen non-deterministically.

During the computation, a configuration of the system is described by the number of spikes present in each neuron; thus, the initial configuration is described by the numbers $n_1, n_2, \ldots, n_m$.

Using the rules as described above, one can define transitions among configurations. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where no rule can be used.

There are many possibilities to associate a result with a computation, in the form of a number (the distance between two input spikes or two output spikes) or of a string. Like in [3], we associate a symbol b_i with a step of a computation when i spikes exit the system, thus generating strings over an alphabet $\{b_0, b_1, \ldots, b_m\}$, for some $m \geq 1$. When one neuron is distinguished as an input neuron, then the sequence of symbols b_i associated as above with the spikes taken from the environment by this input neuron also forms a string. In both cases, we can distinguish two possibilities: to interpret b_0 as a symbol or to simply ignore a step when no spike is read or sent out. The second case provides a considerable freedom, as the computation can proceed inside the system without influencing the result, and this adds power to our devices.

In what follows, we also consider an intermediate case: the system can work inside for at most a given number of steps, k, before reading or sending out a symbol. (Note the important detail that this is a *property* of the system, not a *condition* about the computations: all halting computations observe the restriction to work inside for at most k steps, this is not a way to select some computations as correct and to discard the others. This latter possibility is worth investigating, but we do not examine it here.) The obtained languages, in the accepting and the generating modes, are denoted $L_k^a(\Pi), L_k^g(\Pi)$, respectively, where $k \in \{0, 1, 2, \ldots\} \cup \{\infty\}$. Then, L_0^g corresponds to the restricted case of [3] and L_∞^g to the non-restricted case (denoted L_λ in [3]).

The respective families of languages associated with systems with at most m neurons are denoted by $L_k^\alpha SNP_m$, where $\alpha \in \{g, a\}$ and k is as above; if k is arbitrary, but not ∞, then we replace it with $*$; if m is arbitrary, then we replace it with $*$. (Note that we do not take here into account the descriptional complexity parameters usually considered in this framework: number of rules per neuron, numbers of spikes consumed or forgotten, etc.)

By the definitions, we have the following inclusions for all $\beta \in \{1, 2, \ldots\} \cup \{*\}$:

$$L_0^\alpha SNP_\beta \subseteq L_1^\alpha SNP_\beta \subseteq L_2^\alpha SNP_\beta \subseteq \ldots L_*^\alpha SNP_\beta \subseteq L_\infty^\alpha SNP_\beta \subseteq RE.$$

4 Some Preliminary Results for the Generating Case

In general, the results which were obtained for SN P systems without request rules are expected to hold – maybe with simplified proofs – also for SN P systems with request rules. However, some differences exist. For instance, it is observed in

[2] that the language $\{0,1\}$ cannot be generated by an SN P system (the output neuron cannot choose between spiking or not spiking in the first step), but this language can be generated by the system

$$(\{a\}, (1, \{a \to a,\ a/\lambda \leftarrow a\}), \emptyset, 1),$$

because of the possibility of choosing between spiking or bringing a spike inside (the system halts after the first step). However, this is mainly due to the definition – allowing a nondeterministic choice between spiking and forgetting rules will lead to a similar result also for SN P systems without request rules.

Returning to the extension of results to the new class of SN P systems, we consider here three results from [3], as they are significant below.

Lemma 1. *The number of configurations reachable after n steps by an extended SN P system with request rules of degree m is bounded by a polynomial $g(n)$ of degree m.*

Proof. The same as in [3], with the observation that the number of spikes in the system is increased in two cases: when a neuron spikes, hence it introduces a well defined number of spikes in all neurons with which it has synapses, or when it brings spikes from the environment; in this case, the spikes, again a well defined number, are introduced in the neuron which has used the request rule. The rest of the argument remains the same as in [3]. □

Theorem 1. *If $f : V^+ \longrightarrow V^+$ is an injective function, $card(V) \geq 2$, then there is no extended SN P system Π with request rules such that $L_f(V) = \{x\, f(x) \mid x \in V^+\} = L^g_*(\Pi)$.*

Proof. Assume that there is an extended SN P system Π of degree m, with request rules, such that $L^g_k(\Pi) = L_f(V)$ for some f and V as in the statement of the theorem and some $k \geq 1$. According to the previous lemma, there are only polynomially many configurations of Π which can be reached after n steps. Take some n of the form $n = km$. The string generated after n steps is of length at least m. However, there are $card(V)^m \geq 2^m = 2^{n/k}$ strings of length m in V^+. Therefore, for large enough m there are two strings $w_1, w_2 \in V^+, w_1 \neq w_2$, such that after n steps the system Π reaches the same configuration when generating the strings $w_1\, f(w_1)$ and $w_2\, f(w_2)$, hence after step n the system can continue any of the two computations. This means that also the strings $w_1\, f(w_2)$ and $w_2\, f(w_1)$ are in $L^g_k(\Pi)$. Due to the injectivity of f and the definition of $L_f(V)$ such strings are not in $L_f(V)$, hence the equality $L_f(V) = L^g_k(\Pi)$ is contradictory. □

Corollary 1. *The following two languages are not in $L^g_*SNP_*$ (in all cases, $card(V) = k \geq 2$):*

$$L_1 = \{x\, mi(x) \mid x \in V^+\},$$
$$L_2 = \{xx \mid x \in V^+\}.$$

Note that language L_1 above is a non-regular minimal linear one and L_2 is context-sensitive non-context-free.

Theorems 3 and 4 from [3] (characterizing regular languages, modulo a symbol added to their strings), and Theorem 5 (generating non-semilinear languages), always in the restricted mode, can now be written in the form:

Theorem 2. $L^g_\infty SNP_2 \subseteq REG \subset L^g_1 SNP_3$.

5 The Accepting Case

In order to have a definition of the language accepted by a given SN P system with request rules, we have to distinguish a neuron as the input one, and only the sequence of symbols taken from the environment by that neuron is introduced in the language. The other neurons are allowed to bring spikes from the environment, but those spikes are not defining symbols for the processed string.

First, let us notice that Lemma 1 and Theorem 1 remains true also in the accepting case, hence the languages in Corollary 1 cannot be recognized.

Also the characterization of RE from [3] remains true. However, this proof is rather complex: one takes symbols from the environment, in the form of packages of spikes, but for a string w over an alphabet with k symbols, one passes to $val_{k+1}(w)$, the numerical value of w when considered as a number written in base $k+1$, and one accepts w if and only if $val_{k+1}(w)$ is accepted (a language L is in RE if and only if $val_{k+1}(L)$ is in NRE). Then, handling numbers is reduced to simulating register machines (basically, counter machines without an input tape). Here we will proceed in a direct way, proving that SN P systems with request rules can recognize all RE languages by starting from counter automata with input tape.

Theorem 3. $L^a_\infty SNP_* = RE$.

Proof. Let us consider a counter automaton $M = (n, V, H, l_o, l_h, I)$ as introduced in Section 2, with $V = \{b_1, \dots, b_k\}$. We construct modules simulating instruction of the first four types mentioned in Section 2 – no halting module is necessary, we just ignore the halting instruction. Let us denote by Π the SN P system we construct.

For each counter r of M, Π contains a neuron $\sigma_r, 1 \leq r \leq n$; if the value of the counter will be at some moment m, then the associated neuron will contain $2m$ spikes. For each label $l \in H$, we also introduce in Π a neuron σ_l. All neurons of the system are empty in the beginning of the computation, except neuron σ_{l_0}, which contains one spike. Having a spike inside, this neuron is *active*; in general, when a neuron $\sigma_l, l \in H$, receives one spike, then it is active, the associated module will start working, simulating the instruction identified by the label l. There is a unique input neuron, with the label in, and σ_{in} is the only neuron of Π which contains request rules. Several other auxiliary neurons are involved in the modules. They and the synapses among all these neurons are specified below.

We do not give formally the modules, but in a graphical form.

Let us start with a CHECK instruction, $(l_i : \texttt{check}(r), l_j, l_k)$. We construct the module shown in Figure 1.

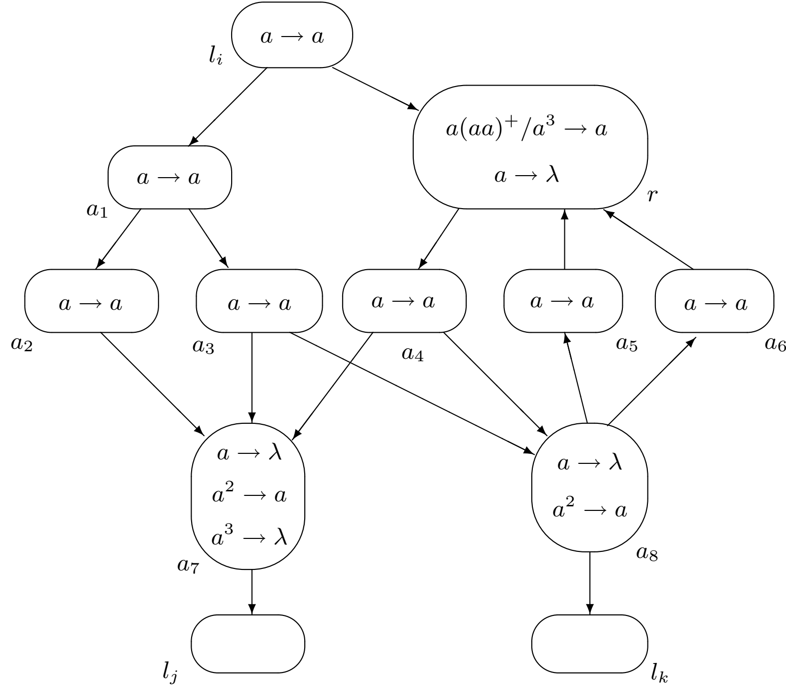

Fig. 1. The CHECK module

As long as the contents of a neuron σ_r is an even number, no rule can be applied in it. If σ_{l_i} becomes active, it sends a spike to σ_r, hence the number of spikes becomes odd. If this number is 1, hence the counter was empty, then a spike will eventually arrive in σ_{l_j}. If the counter was non-zero, then only σ_{a_8} will spike. In this way, it activates the neuron σ_{l_k} and also restores the contents of counter r, putting back the two spikes consumed by the rule $a(aa)^+/a^3 \to a$. The continuation is correct in both cases.

The new neurons, $\sigma_{a_s}, 1 \leq s \leq 8$, are uniquely associated with this module (it would be more rigorous to label them, say, by $(l_i, s), 1 \leq s \leq 8$, but we prefer the simple writing). All the three label neurons in Figure 1 can have incoming synapses from various other neurons, but each such neuron has only one associated module which is triggered by it. These remarks are valid for all modules constructed below.

Because several CHECK instructions (also several SUB instructions – see below) can act on the same counter r, it means that σ_r can have synapses to several

neurons of type σ_{a_4}, in various modules. However, this entails nothing wrong, because the spike produced by σ_{a_4} will be erased in both neurons $\sigma_{a_7}, \sigma_{a_8}$.

The modules for the ADD and SUB instructions are rather simple – they are given in Figure 2 (for $(l_i : \texttt{add}(r), l_j)$) and Figure 3 (for $(l_i : \texttt{sub}(r), l_j)$; remember that, before activating a SUB instruction, we assume that we have checked whether the operation can be done, that is, whether the counter is non-zero, hence we can assume that always the operation asked for by the instruction $(l_i : \texttt{sub}(r), l_j)$ is possible).

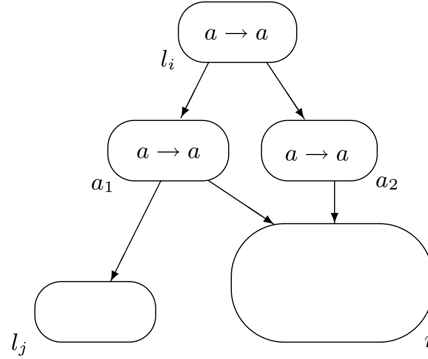

Fig. 2. The ADD module

In the SUB case, we have reproduced both rules of σ_r used in the CHECK module, as the first one is also used in the SUB case (but the produced spike is "lost", as explained before when discussing the CHECK module); the second rule, $a \to \lambda$, cannot be used, as we know in advance that the substraction is possible, hence the neuron is not empty.

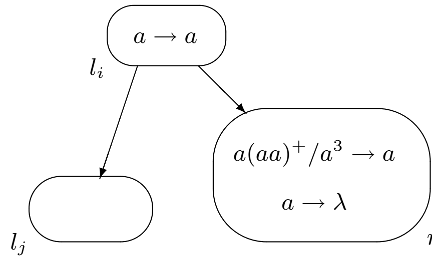

Fig. 3. The SUB module

Only the READ instructions $(l_i : \texttt{read}(b_s), l_j), 1 \leq s \leq k$, remains to be considered. For such an instruction, we build a module like in Figure 4. Note that the module contains several neurons, depending on s, and that the input neuron, labeled with in, is unique for the whole system.

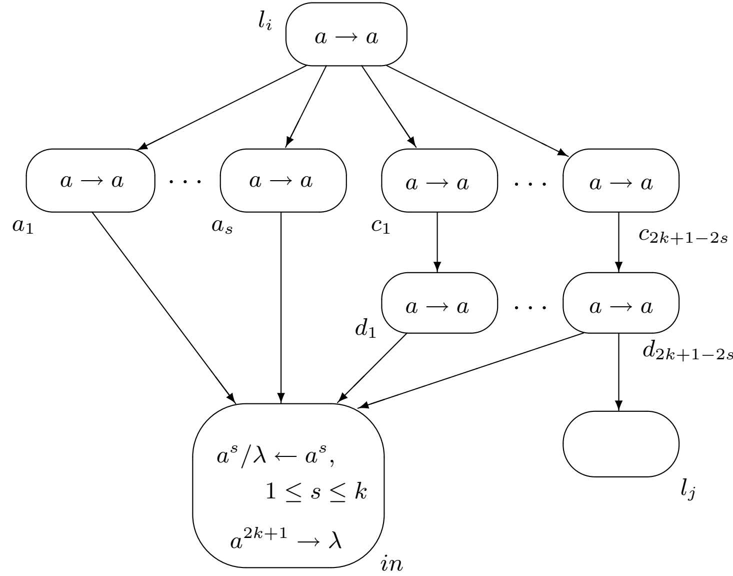

Fig. 4. The READ module

After activating σ_{l_i}, s spikes reach σ_{in}, and in this way the input neuron can take from the environment the correct number of spikes (reading in this way the symbol b_s). Because the input neuron can be used in any further step, it should be left empty after simulating the instruction $(l_i : \texttt{read}(b_s), l_j)$, and to this aim the use of the forgetting rule $a^{2k+1} \to \lambda$ is made possible, after receiving from neurons $\sigma_{d_t}, 1 \leq t \leq 2k+1-2s$, the corresponding number of spikes. One of these neurons, the last one, also sends a spike to σ_{l_j}, in order to continue the computation as requested by the READ instruction.

By repeatedly using these modules, the computation in M is correctly simulated; as pointed out above, there is no misleading interaction between modules. When the label l_h is reached in M, the unique neuron associated with a label which has a spike inside is σ_{l_h}, where we provide no rule, hence the computation halts also in Π. Consequently, the recognized string is the same, $L(M) = L^a_\infty(\Pi)$, and this completes the proof. $\square$

Note the interesting fact that in the previous construction we only use non-extended spiking rules, always producing only one spike. If we allow the use of

extended rules, then some of the constructions can be slightly simplified (this is the case, for instance, with the modules CHECK).

6 SN dP Systems

We pass now to the main goal of our paper, introducing the SN P systems counterpart of dP systems from [13]. We directly introduce the definition of the systems we investigate.

An *SN dP system* is a construct

$$\Delta = (O, \Pi_1, \ldots, \Pi_n, esyn),$$

where (1) $O = \{a\}$ (as usual, a represents the spike), (2) $\Pi_i = (O, \sigma_{i,1}, \ldots, \sigma_{i,k_i}, syn, in_i)$ is an SN P system with request rules present only in neuron σ_{in_i} ($\sigma_{i,j} = (n_{i,j}, R_{i,j})$, where $n_{i,j}$ is the number of spikes initially present in the neuron and $R_{i,j}$ is the finite set of rules of the neuron, $1 \leq j \leq k_i$), and (3) $esyn$ is a set of *external synapses*, namely between neurons from different systems Π_i, with the restriction that between two systems Π_i, Π_j there exist at most one link from a neuron of Π_i to a neuron of Π_j and at most one link from a neuron of Π_j to a neuron of Π_i. We stress the fact that we allow request rules only in neurons σ_{in_i} of each system Π_i – although this restriction can be removed; the study of this extension remains as a task for the reader. The systems $\Pi_i, 1 \leq i \leq n$, are called *components* (or *modules*) of the system Δ.

As usual in dP automata, each component can take an input (by using request rules), work on it by using the spiking and forgetting rules in the neurons, and communicate with other components (along the synapses in $esyn$); the communication is done as usual inside the components: when a spiking rule produces a number of spikes, they are sent simultaneously to all neurons, inside the component or outside it, in other components, provided that a synapse (internal or external) exists to the destination.

As above, when r spikes are taken from the environment, a symbol b_r is associated with that step, hence the strings we consider introduced in the system are over an alphabet $V = \{b_0, b_1, \ldots, b_k\}$, with k being the maximum number of spikes introduced in a component by a request rule.

A halting computation with respect to Δ accepts the string $x = x_1 x_2 \ldots x_n$ over V if the components $\Pi_1, \ldots, \Pi_n$, starting from their initial configurations, working in the synchronous (in each time unit, each neuron which can use a rule should use one) non-deterministic way, bring from the environment the substrings $x_1, \ldots, x_n$, respectively, and eventually halts.

Hence, the SN dP systems are synchronized, a universal clock exists for all components and neurons, marking the time in the same way for the whole system.

In what follows, like in the communication complexity area, see, e.g., [8], we ask the components to take equal parts of the input string, modulo one symbol. (One

also says that the string is distributed *in a balanced way*. The study of the unbalanced (free) case remains as a research issue.) Specifically, for an SN dP system Δ of degree n we define the language $L(\Delta)$, of all strings $x \in V^*$ such that we can write $x = x_1 x_2 \dots x_n$, with $||x_i| - |x_j|| \leq 1$ for all $1 \leq i, j \leq n$, each component Π_i of Δ takes as input the string x_i, $1 \leq i \leq n$, and the computation halts. Moreover, we can distinguish between considering b_0 as a symbol or not, like in the previous sections, thus obtaining the languages $L_\alpha(\Delta)$, with $\alpha \in \{0, 1, 2, \dots\} \cup \{\infty, *\}$.

Let us denote by $L_\alpha SNdP_n$ the family of languages $L_\alpha(\Delta)$, for Δ of degree at most n and $\alpha \in \{0, 1, 2, \dots\} \cup \{\infty, *\}$. An SN dP system of degree 1 is a usual SN P system with request rules working in the accepting mode (with only one input neuron), as considered in Section 5. Thus, the universality of SN dP systems is ensured, for the case of languages $L_\infty(\Delta)$.

In what follows, we prove the usefulness of distribution, in the form of SN dP systems, by proving that one of the languages in Corollary 1, can be recognized by a simple SN dP system (with two components), even working in the L_k mode.

Proposition 1. $\{ww \mid w \in \{b_1, b_2, \dots, b_k\}^*\} \in L_{k+2} SNdP_2$.

Proof. The SN dP system which recognizes the language in the proposition is the following:

$$\begin{aligned}
\Delta &= (\{a\}, \Pi_1, \Pi_2, \{((2,1),(1,3)),\ ((1,5),(2,1))\}), \text{ with the components}\\
\Pi_1 &= (\{a\}, \sigma_{(1,1)}, \dots, \sigma_{(1,7)}, syn_1, (1,1)),\\
&\quad \sigma_{(1,1)} = (3, \{a^3/\lambda \leftarrow a^r \mid 1 \leq r \leq k\} \cup \{a^4 a^+/a \to a,\ a^4 \to a^3\}),\\
&\quad \sigma_{(1,2)} = (0, \{a \to a,\ a^3 \to a^3\}),\\
&\quad \sigma_{(1,3)} = (0, \{a \to a,\ a^3 \to a^3\}),\\
&\quad \sigma_{(1,4)} = (0, \{a^2 \to \lambda,\ a^6 \to \lambda,\ a \to a,\ a^4 \to a\}),\\
&\quad \sigma_{(1,5)} = (0, \{a^2 \to \lambda,\ a^6 \to a,\ a^6 \to a^3\}),\\
&\quad \sigma_{(1,6)} = (0, \{a^+/a \to a\}),\\
&\quad \sigma_{(1,7)} = (0, \{a^+/a \to a\}),\\
&\quad syn_1 = \{((1,1),(1,2)),\ ((1,5),(1,1)),\ ((1,2),(1,4)),\ ((1,4),(1,6)),\\
&\qquad\qquad ((1,4),(1,7)),\ ((1,6),(1,7)),\ ((1,7),(1,6)),\ ((1,2),(1,5)),\\
&\qquad\qquad ((1,3),(1,4)),\ ((1,3),(1,5))\},\\
\Pi_2 &= (\{a\}, \sigma_{(2,1)}, \emptyset, (2,1)),\\
&\quad \sigma_{(2,1)} = (3, \{a^3/\lambda \leftarrow a^r \mid 1 \leq r \leq k\} \cup \{a^4 a^+/a \to a,\ a^4 \to a^3\}).
\end{aligned}$$

For an easier reference, the system is also presented graphically, in Figure 5.

Assume that we are in a configuration where both neurons $\sigma_{(1,1)}$ and $\sigma_{(2,1)}$ contain three spikes, as in the beginning of the computation. Each neuron can bring spikes inside, by using the rules $a^3/\lambda \leftarrow a^r$; if they bring the same number of spikes, then the system will return to a configuration as the one we started with, hence the computation can continue, otherwise the system will never halt.

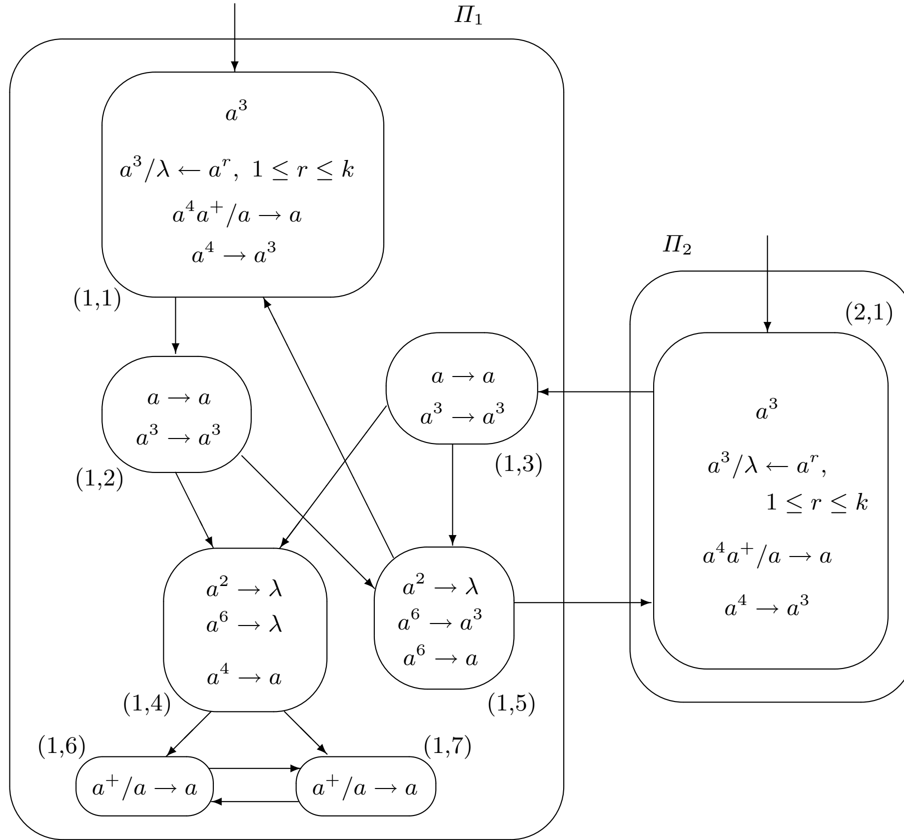

Fig. 5. The SN dP system from the proof of Proposition 1

For instance, assume that $\sigma_{(1,1)}$ brings inside r_1 spikes and $\sigma_{(2,1)}$ brings r_2 spikes. These spikes are moved one by one (at most k steps in total, where k is the cardinality of the alphabet) to neurons $\sigma_{(1,2)}$ and $\sigma_{(1,3)}$, respectively, by using the rules $a^4a^+/a \to a$, and from here, duplicated, in neurons $\sigma_{(1,4)}, \sigma_{(1,5)}$, where they are removed by the forgetting rules $a^2 \to \lambda$. If $r_1 = r_2$, then the neurons $\sigma_{(1,1)}, \sigma_{(2,1)}$ use at the same time the rules $a^4 \to a^3$, and after one further step six spikes reach both $\sigma_{(1,4)}$ and $\sigma_{(1,5)}$; in the former neuron the spikes are forgotten, in the latter one can use the rule $a^6 \to a^3$, which will send three spikes to $\sigma_{(1,1)}, \sigma_{(2,1)}$. The process can continue, one reads one further symbol of the string. If $r_1 \neq r_2$, then one of $\sigma_{(1,1)}, \sigma_{(2,1)}$ produces one spike and the other three, hence four spikes arrive in neuron $\sigma_{(1,4)}$; this neuron sends a spike to $\sigma_{(1,6)}$ and $\sigma_{(1,7)}$, and these neurons will exchange forever spikes, hence the computation never halts.

If, instead of the rule $a^6 \to a^3$, neuron $\sigma_{(1,5)}$ uses the rule $a^6 \to a$, then the computation stops, because the input neurons cannot fire having inside only one spike. This is the only way to stop the computation, hence the strings read by the two components are equal.

Note that after introducing some r spikes in each component of the system, we need $r-1$ steps for using the rules $a^4a^+/a \to a$, one step for the rule $a^4 \to a^3$ (at most k steps in total), then two more steps for sending three spikes (one in the end) to neurons $\sigma_{(1,1)}$ and $\sigma_{(2,1)}$. Therefore, $\{ww \mid w \in \{b_1, b_2, \ldots, b_k\}^*\} \in L_{k+2}SNdP_2$, and the proposition is proved. □

7 Final Remarks

Many problems can be formulated for SN P systems with request rules and for SN dP systems. Several were already mentioned in the previous sections. Let us close by recalling the fact that besides the synchronized (sequential in each neuron) mode of evolution, there were also introduced other modes, such as the exhaustive one, [10], and the non-synchronized one, [1]. Universality was proved for these types of SN P systems, but only for the extended case. Can universality be proved for non-extended SN P systems also using request rules?

Acknowledgements

The work of M. Ionescu was possible due to CNCSIS grant RP-4 12/01.07.2009. The work of Gh. Păun was supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200.

References

1. M. Cavaliere, E. Egecioglu, O.H. Ibarra, M. Ionescu, Gh. Păun, S. Woodworth: Asynchronous spiking neural P systems. *Theoretical Computer Science*, 410, 24-25 (2009), 2352–2364.
2. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. *Fundamenta Informaticae*, 75, 1-4 (2007), 141–162.
3. H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with extended rules. In *Proc. Fourth Brainstorming Week on Membrane Computing*, Sevilla, 2006, RGNC Report 02/2006, 241–265.
4. H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Handling languages with spiking neural P systems with extended rules. *Romanian J. Information Sci. and Technology*, 9, 3 (2006), 151–162.
5. P.C. Fischer: Turing machines with restricted memory access. *Information and Control*, 9 (1966), 364–379.

6. R. Freund, M. Kogler, Gh. Păun, M.J. Pérez-Jiménez: On the power of P and dP automata. *Annals of Bucharest University. Mathematics-Informatics Series*, 63 (2009), 5–22.
7. J.E. Hopcroft, J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Mass., 1979.
8. J. Hromkovic: *Communication Complexity and Parallel Computing: The Application of Communication Complexity in Parallel Computing*. Springer, Berlin, 1997.
9. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
10. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems with exhaustive use of rules. *Intern. J. Unconventional Computing*, 3, 2 (2007), 135–154.
11. M. Minsky: *Computation – Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, NJ, 1967.
12. Gh. Păun: *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
13. Gh. Păun, M.J. Pérez-Jiménez: Solving problems in a distributed way in membrane computing: dP systems. *Int. J. of Computers, Communication and Control*, 5, 2 (2010), 238–252.
14. 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.
15. Gh. Păun, M.J. Pérez-Jiménez: P and dP automata: A survey. *Lecture Notes in Computer Science*, 6570, in press.
16. Gh. Păun, M.J. Pérez-Jiménez: An infinite hierarchy of languages defined by dP systems. *Theoretical Computer Sci.*, in press.
17. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *Handbook of Membrane Computing*. Oxford University Press, 2010.
18. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*. 3 volumes, Springer, Berlin, 1998.
19. A. Salomaa: *Formal Languages*. Academic Press, New York, 1973.
20. The P Systems Website: `http://ppage.psystems.eu`.