ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume 26, Number 3–4, 2023, 251–275

Infinite Spike Trains in Spiking Neural P Systems

Gheorghe PĂUN[1], Mario J. PÉREZ-JIMÉNEZ[2, 3, *], and Grzegorz ROZENBERG[4, 5]

[1]Institute of Mathematics of the Romanian Academy, PO Box 1-764, 014700 București, Romania
[2]Research Group on Natural Computing, Department of Computer Science and Artificial Intelligence, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
[3]SCORE Laboratory, I3US, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
[4]Leiden Institute of Advanced Computer Science, LIACS, University of Leiden, Niels Bohr Weg 1, 2333 CA Leiden, The Netherlands
[5]Department of Computer Science, University of Colorado, Boulder, CO 80309-0430, USA
Email: curteadelaarges@gmail.com, marper@us.es*, grozenberg@gmail.com
* Corresponding author

Abstract. We initiate the study of spiking neural P systems associated with infinite sequences, by considering them as computability devices which generate infinite sequences of bits (1 indicates a step when a spike exits the system, and 0 indicates a step when the system does not send a spike to the environment), and as devices which process infinite sequence of bits (for instance, computing Boolean operations or other operations on two input sequences). For both the generating and the transduction case we introduce some basic notions illustrated by numerous examples, establish some basic properties, and formulate a number of research topics.

Key-words: Membrane Computing; Spiking Neural P Systems; Spike Trains; Formal Languages.

1. Introduction

This paper is a direct continuation of [4], where the idea of spiking neurons (see, e.g., [3], [6], [7]) was incorporated in the framework of membrane computing as well as a continuation of [9], where the results of [4] were extended in several ways in order to define the set of numbers computed by a spiking neural P system. The computed numbers are time distances between consecutive spikes which leave the system.

A spiking neural P system (SN P system, for short) consists of a set of *neurons* placed in the nodes of a graph whose edges are called *synapses*. The neurons contain *spikes*, objects of a unique type, which can be processed by either *firing/spiking rules* or by *forgetting rules*. The firing rules consume some spikes and produce a new spike, which is sent to all neurons linked by a synapse to the neuron where the rule was used. The forgetting rules just remove spikes from neurons. One of the neurons is the *output* neuron, and its spikes can also exit into the environment, thus providing a trace of the system evolution. Like in neurobiology, we call this trace (the sequence of moments when spikes are exiting the system) the *spike train*.

In [4] one considers as successful only the computations whose spike trains contain exactly two spikes, and the number of steps that took place between these two spikes is the result of that computation, while in [9] one investigates several other possibilities, such as considering computations with $k \geq 2$ spikes, or even computations with infinitely many spikes. In each case one defines two sets of numbers – the durations of all intervals between two consecutive spikes, or taking only alternately these intervals. In the case of a given number of spikes, both halting and non-halting computations have been considered.

In all these cases similar results were obtained in [4] and in [9]: Turing universality in the general case (neurons without any bound on the number of spikes they contain) and semilinearity in the bounded case.

What naturally remains to be investigated is the behavior of SN P systems as generators and processors/transducers of infinite sequences (of bits) – this is the subject of the present paper.

More specifically, we consider SN P systems: as generators of infinite sequences of bits (1 is associated with moments of time when a spike exits the system, and 0 is associated with moments of time when no spike exits) and as processors of infinite sequences of bits (one spike is introduced into the system when 1 is read and nothing is introduced when we read 0, and, as in the generative case, we output a binary sequence, with 1 marking the steps when a spike exits the system). While in the former case we mainly illustrate the intricate work of these systems, providing a series of interesting examples, in the later case we provide a series of results showing the versatility and the power of SN P systems in handling infinite sequences of bits. For instance, we provide a consistent "tool-kit" of operations which can be realized by SN P systems, and, based on them, we show how any Boolean function and more complex operations (e.g., a repeated crossing-over between two sequences, with the crossings guided by the occurrences of digit 1 in the third sequence, a guiding sequence) can be computed. Moreover, although morphisms which increase the length cannot be computed by our devices, we show that the length preserving functions, defined on blocks of bits can be computed.

In Section 2 we introduce some general notions and notations we need in the sequel. In Section 3, we recall from [4] and [9] the definition of spiking neural P systems and of spike trains they generate. In Section 4, we consider the SN P system as generators of infinite sequences of bits, illustrating the definitions with several examples. In Section 5, we introduce input neurons as a method to transduce an input signal into an output signal. In Section 6, we present a series of specific SN P systems which can be used as modules in more complex systems. They are then used to show that several interesting transformations of infinite sequences of bits can be realized by our systems. Section 7, deals with the (im)possibility of computing morphic images of infinite sequences.

2. Prerequisites

We assume the reader to be familiar with basic language and automata theory, as well as with membrane computing, so we recall only a few definitions here, in order to establish the notation we use. For further details we refer to [12] and [8], respectively (and to [14] for recent information about membrane computing).

For an alphabet V, V^* is the set of all finite strings of symbols from V, with the empty string being denoted by λ. The set of all non-empty strings over V is denoted by V^+. When $V = \{a\}$ is a singleton, we write simply a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$, respectively. The length of a string $x \in V^*$ is denoted by $|x|$.

We also consider infinite sequences of symbols from a given alphabet V, and their set is denoted by V^ω. When $V = \{a\}$ we write a^ω instead of $\{a\}^\omega$. When dealing with finite strings we say simply "strings" and we say "infinite strings/sequences" in the infinite case.

In the rules of spiking neural P systems we use the notion of a *regular expression*. Given an alphabet V, (i) λ and each $a \in V$ is a regular expression over V, (ii) if E_1, E_2 are regular expressions over V, then $(E_1)(E_2)$, $(E_1) \cup (E_2)$, and $(E_1)^+$ are regular expressions over V, and (iii) nothing else is a regular expression over V. The non-necessary parentheses can be omitted, while $E_1^+ \cup \lambda$ can be written as E_1^*. With each expression E we associate the language $L(E)$ as follows: (i) $L(\lambda) = \{\lambda\}$ and $L(a) = \{a\}$, for all $a \in V$, (ii) $L((E_1)(E_2)) = L(E_1)L(E_2)$, $L((E_1) \cup (E_2)) = L(E_1) \cup L(E_2)$ and $L((E_1)^+) = L(E_1)^+$, for any regular expressions E_1, E_2.

The operations used here are the standard operations of union, concatenation, and Kleene $+$. We will also need below the operation of the *right derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$, which is defined by

$$L/x = \{y \in V^* \mid yx \in L\}.$$

3. Spiking Neural P Systems

We recall now from [4] the definition of spiking neural P systems, without recalling the neural motivation, but presenting informally only the basic ideas, especially those which make them essentially different from the usual membrane systems. We work with only one object, denoting a spike, a quanta of energy sent by a neuron along its axon to all neurons with which it is linked through a synapse. This means that we have these neurons (single membranes) placed in the nodes of an arbitrary graph, with one of the neurons called the output neuron. Depending on their contents (number of accumulated spikes), the neurons either fire – and immediately or at a subsequent step spike, send a spike to the neighboring neurons – or forget the spikes they have. In this way we get a sequence of spikes, leaving the system (through its output neuron) at specific moments of times. This is called a *spike train*.

Formally, a *spiking neural P system* (in short, a 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 in the neuron;

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

(1) $E/a^c \to a; d$, where E is a regular expression over O using the symbol a only, $c \geq 1$, and $d \geq 0$;

(2) $a^s \to \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for all rules $E/a^c \to a; d$ 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* among neurons);

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 $E/a^c \to a; d$ can be applied, which means that c spikes are consumed, $k - c$ spikes remain in the neuron, the neuron fires, 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. If $d \geq 1$ and if the rule is used in step t, then in steps $t, t + 1, t + 2, \ldots, t + d - 1$ the neuron is *closed* (we also say *blocked*; this corresponds to the refractory period from neurobiology, [1]), and so it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then the spike is lost). In step $t + d$, the neuron spikes and becomes again open, hence it can receive spikes (which can be used in step $t + d + 1$). A spike emitted by a neuron σ_i goes to all neurons σ_j such that $(i, j) \in syn$.

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.

At each time unit, in each neuron which can use a rule we have to use a rule, either a firing or a forgetting one, the rule has to be used. 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 $E/a^c \to a; d$, $a^s \to \lambda$ have disjoint domains, in the sense that $a^s \notin L(E)$.

Thus, the rules are used in the sequential manner in each neuron, and in parallel for all neurons of the system. It is important to stress that each rule used "covers" completely the neuron: all spikes present in it are taken into consideration by the rule which is applied. (For instance, if a neuron contains 5 spikes and it contains the rules $(aa)^*/a \to a; 0$, $a^3 \to a; 0$, $a^2 \to \lambda$, then none of these rules can be used: a^5 is not in $L((aa)^*)$ and is not equal to either a^3 or a^2. However, the rule $a^5/a^2 \to a; 0$ can be used, and the effect of it is that two spikes are consumed, three remains in the neuron – no rule is applied to them in this step – and one spike is produced and sent immediately to the neurons linked by a synapse to the neuron where the rule was used.

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 by both the numbers of spikes present in each neuron and by the state of each neuron, in the open-closed sense, meaning that if a neuron is closed, then we have to specify the moment when it will become again open.

Using the rules as described 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* by Π. 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*, i.e., the sequence $\langle t_1, t_2, \ldots \rangle$ of natural numbers $1 \leq t_1 < t_2 < \ldots$ corresponding to the moments of time when the output neuron sends a spike out of the system (we also say that the system itself spikes at that time).

In the spirit of spiking neurons ([4], [9]) one defines the result of a computation as the number of steps between two consecutive spikes sent out by the output neuron (that is, the differences $t_i - t_{i-1}, i \geq 2$, for a spike train), with a number of possible variants: taking into consideration only the first two spikes or only a given number of spikes, requiring to have only two spikes in the spike train or only a fixed number $k \geq 2$ of spikes, considering alternately the intervals, accepting only halting computations, or only infinite computations, etc. For all these variants two types of results were obtained: Turing computability in the case of neurons without any bound on the number of spikes present inside, and a characterization of semilinear sets of numbers in the case of systems whose neurons have a bound on the number of spikes.

4. Generating Infinite Spike Trains

We depart now from the style of [4] and [9], and consider SN P systems not as number generators, but as devices which handle infinite sequences of symbols 0 and 1. In this section we deal with the generative case, in the forthcoming sections we will deal with the transduction case.

Let us consider a SN P system $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$ and a computation $\sigma = C_0 \Longrightarrow C_1 \Longrightarrow C_2 \Longrightarrow \ldots$ by Π (C_0 is the initial configuration, and $C_{i-1} \Longrightarrow C_i$ is the ith step of the computation). In some steps a spike exits the (output neuron of the) system, and in some steps it does not. For a step when no spike exits the system we write 0, and for a step when the system sends a spike out we write 1. In this way, for the computation σ we obtain a sequence of symbols/bits 0 and 1, which we call the *binary spike train* of σ (note the difference with respect to [4] and [9], where the spike train of a computation was defined as the sequence of moments where a spike exits the system.)

This sequence is denoted by $bst(\sigma)$ (the first letter is used in order to underline that we consider here the **b**inary sequence which describes the spike train, and not the moments of time when the spikes exit the system, as it was the case in [4] and [9]). If the system is *deterministic* (in each neuron, at each moment, at most one rule can be used), then the system proceeds along only one computation. If the system is *non-deterministic*, then the computations can branch. In both cases, even if a computation halts (it reaches a configuration where no rule can be applied), we assume that it generates an infinite spike train, by interpreting all moments of time after the halting step as moments when no spike exits the system, hence the binary spike train continues forever with the sequence of 0. Thus, in this paper, all spike trains are infinite.

In the deterministic case, we denote by $bst(\Pi)$ the unique binary spike train generated by Π. If Π is non-deterministic, the set of binary spike trains is denoted by $BST(\Pi)$.

As in the case of sets of numbers associated with spike trains, also in the case of binary spike trains we can consider sets of sequences generated by systems with a number of neurons bounded by a constant m, using rules which consume a number of spikes and forget a number of spikes bounded by p, q, respectively, maybe also bounding the number of spikes present in the system, and then we can compare such families with families of infinite sequences recognized by finite automata, Turing machines or other types of automata – see [10] or the relevant chapters from [12] for details about such classic families of infinite sequences. However, we do not go here in this direction (which is worth pursuing, hence we formulate it as an interesting research topic for the reader). Instead, we will now illustrate the functioning and the power of our devices by a series of examples both interesting by themselves and useful in the sequel of the paper.

We start with two simple systems which generate deterministically the same binary spike train, $w_k = 1^k0^\omega$, for some $k \geq 1$. The two systems are presented in Figure 1, which also illustrates SN P systems working in the generating mode. Neurons (represented by ovals) are placed in the nodes of a graph whose edges represent synapses, with the output neuron also having an arrow pointing to the environment. Within each neuron we specify the number of existing spikes and the rules. If a firing rule $E/a^c \to a; d$ has $L(E) = \{a^r\}$, then we write it in the simpler form $a^c \to a; d$.

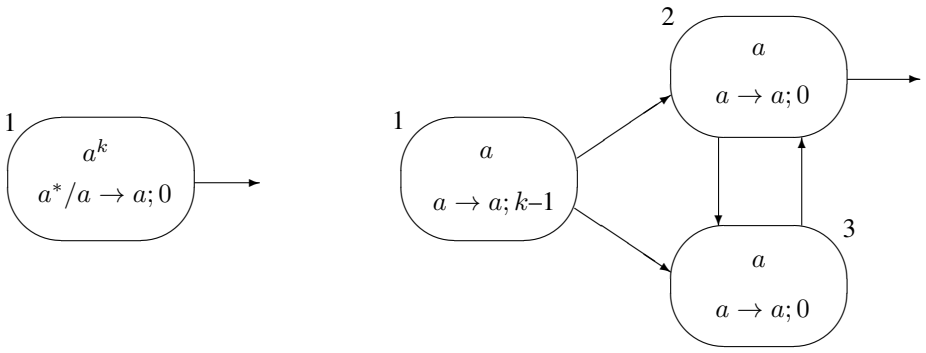

Fig. 1. Two SN P systems generating 1^k0^ω

Let Π_1 be the system on the left side of Figure 1 and let Π_2 be the system on the right side.

Clearly, Π_1 can use the rule $a^*/a \to a; 0$ as long as there is at least one spike in the unique neuron of the system, hence k times.

In Π_2, neurons 2, 3 fire simultaneously as long as each of them contains exactly one spike, and this happens again only for k steps: in the first step, also neuron 1 fires, and its spike arrives in neurons 2 and 3 after $k-1$ steps, hence in step k, thus preventing any further firing of neurons 2 and 3 (the rule $a \to a; 0$ cannot be used if the neuron contains two – or more – spikes).

Consequently, $bst(\Pi_1) = bst(\Pi_2) = 1^k0^\omega$.

A couple of neurons like neurons 2 and 3 in system Π_2, which repeatedly load each other with a spike (thus making possible an infinite collaboration) is worth remembering, because in many constructions in this paper we will make use of this way of continuously providing spikes to other neurons in the system.

The next example is recalled from [9], but we consider it here, as a generator of infinite sequences. It is system Π_3 represented graphically in Figure 2, and it uses two couples of self-

sustaining neurons similar to the couple of neurons 2, 3 from Figure 1. This system is also a generalization of Π_2 above, in the sense that this time we want to obtain the whole set of sequences of the form $1^k 0^\omega$, for all $k \geq 1$ (actually, we will do it only for $k \geq 2$). Thus we have to consider a non-deterministic system, and this is the case for Π_3.

Formally, Π_3 is specified as follows:

$$\begin{aligned}
\Pi_3 &= (O, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, syn, out),\\
O &= \{a\},\\
\sigma_1 &= (1, \{a \to a; 0\}),\\
\sigma_2 &= (1, \{a \to a; 0,\ \ a \to a; 1\}),\\
\sigma_3 &= (0, \{a \to a; 0,\ \ a^2 \to \lambda\}),\\
\sigma_4 &= (1, \{a \to a; 0\}),\\
\sigma_5 &= (1, \{a \to a; 0\}),\\
syn &= \{(1,2), (2,1), (1,3), (2,3), (3,4), (3,5), (4,5), (5,4)\},\\
out &= 4.
\end{aligned}$$

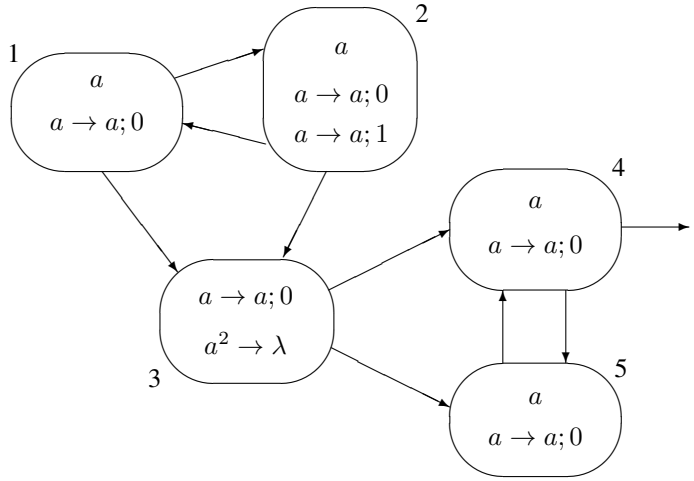

Fig. 2. A non-deterministic SN P system with infinite spike trains

This system contains two "modules" consisting of pairs of neurons which sustain each other: neurons 4 and 5 are in fact neurons 2 and 3 from Π_2, while neurons 1 and 2 form a similar couple as long as neuron 2 fires using the first rule, $a \to a; 0$ (thus, this neuron behaves non-deterministically – and this is the only non-deterministic neuron in Π_3). As long as neuron 4 contains exactly one spike, it fires and sends out a spike. This happens in the beginning, hence the binary spike trains start with a sequence of 1's. The work of neurons 4, 5 can be stopped only if they receive spikes from neuron 3. In turn, this neuron spikes only if it contains only one spike. Thus, as long as both neurons 1 and 2 send a spike to neuron 3, we use here the forgetting rule $a^2 \to \lambda$. Then, neurons 1 and 2 send two spikes to neuron 3 as long as both of them use the rule $a \to a; 0$.

However, at any moment, starting with the first step, neuron 2 can use the rule $a \to a; 1$. This means that at that step it sends no spike to neurons 1 and 3, and it also gets closed. In this way, none of neurons 1 and 2 will contain a spike in the next moment, while neuron 3 has only one spike and can fire in the next step using the rule $a \to a; 0$, hence it sends a spike to each of neurons 4 and 5. Thus, from now on these neurons are idle, because they do not have rules for more than one spike. The spiking out stops, hence the spike train continues to infinity with 0's only.

Note that the internal work of the system does not stop after having neurons 4, 5 blocked: the spike of neuron 2, produced by the rule $a \to a; 1$, will be sent to both neurons 1 and 3 one step after using the rule, and both these neurons fire again; neuron 1 sends spikes to both neurons 2 and 3, which also fire, and the process continues forever, sending further spikes to neurons 4 and 5, which never fire again.

Even if neuron 2 fires in the first step using the rule $a \to a; 1$, neurons 4 and 5 are blocked only from step 3 on, hence all spike trains start with at least two 1's. Consequently,

$$BST(\Pi_3) = \{1^k 0^\omega \mid k \geq 2\}.$$

Let us now consider a more complex system, Π_4, whose initial configuration is shown in Figure 3.

The system is deterministic. At the beginning, it contains spikes in neurons 1 and 8, which fire and spike immediately. Their spikes reach neurons 2 and 3, which can now fire using the rule $a^2 \to a; 0$. As long as both neurons 2 and 3 fire, their spikes sent to neuron 4 are forgotten through the use of the rule $a^2 \to \lambda$.

Let us assume that neuron 1 contains $2n + 1$ spikes, for some $n \geq 0$, this is the case, initially for $n = 0$. Each use of the rule $a^3(aa)^*/a^2 \to a; 0$ decreases by 2 the number of spikes, and this must be done as long as at least three spikes are present. The spike from neuron 1 reaches both neurons 2 and 3, which, using also the spike which they receive from each other, fire using the rule $a^2 \to a; 0$. Each of these neurons sends one spike to neuron 5. Thus, two spikes are consumed in neuron 1 and two are added to neuron 5, which cannot use its rules, because all of them (like in neuron 1) can be used only if the number of spikes there is odd.

This moving of spikes from neuron 1 to neuron 5 continues for n steps, until only one spike remains in neuron 1. In that step, while neurons 2 and 3 fire again adding two more spikes to neuron 5, neuron 1 fires for the last time (during this cycle), using the rule $a \to a; 0$. In the next step, neurons 2 and 3 send again two spikes to neuron 5, thus setting the contents of neuron 5 to $2n + 2$ spikes, but now, because neuron 1 does not fire (it is empty), neurons 2 and 3 will contain only one spike, the one received from the partner neuron. Neuron 2 has to forget this spike, while neuron 3 fires. Its single spike reaches at the same time neurons 4 and 5. This time, neuron 4 has to fire, and it sends a spike to each of the neurons 6, 7, and 9. In the next step one spike is sent out of the system (hence after $n + 4$ steps since we have started to move the $2n + 1$ spikes from neuron 1 to neuron 5), the spikes sent to neurons 6 and 7 trigger a procedure similar to the previous one, of moving the contents of neuron 5 to neuron 1, with the help of neurons 6, 7. This is done exactly as above, with the difference that the number of spikes moved from neuron 5 to neuron 1 is not changed, because in the last move we use the rule $a^3 \to a; 0$ from neuron 5 (in neuron 1 we have used the rule $a \to a; 0$). In the next step, both neurons 6 and 7 fire for the last time (in this cycle), and one step later only neuron 7 fires. This has the same effect as the firing of neuron 3 at the end of moving the contents of neuron 1 into neuron 5, enabling neuron 8 to

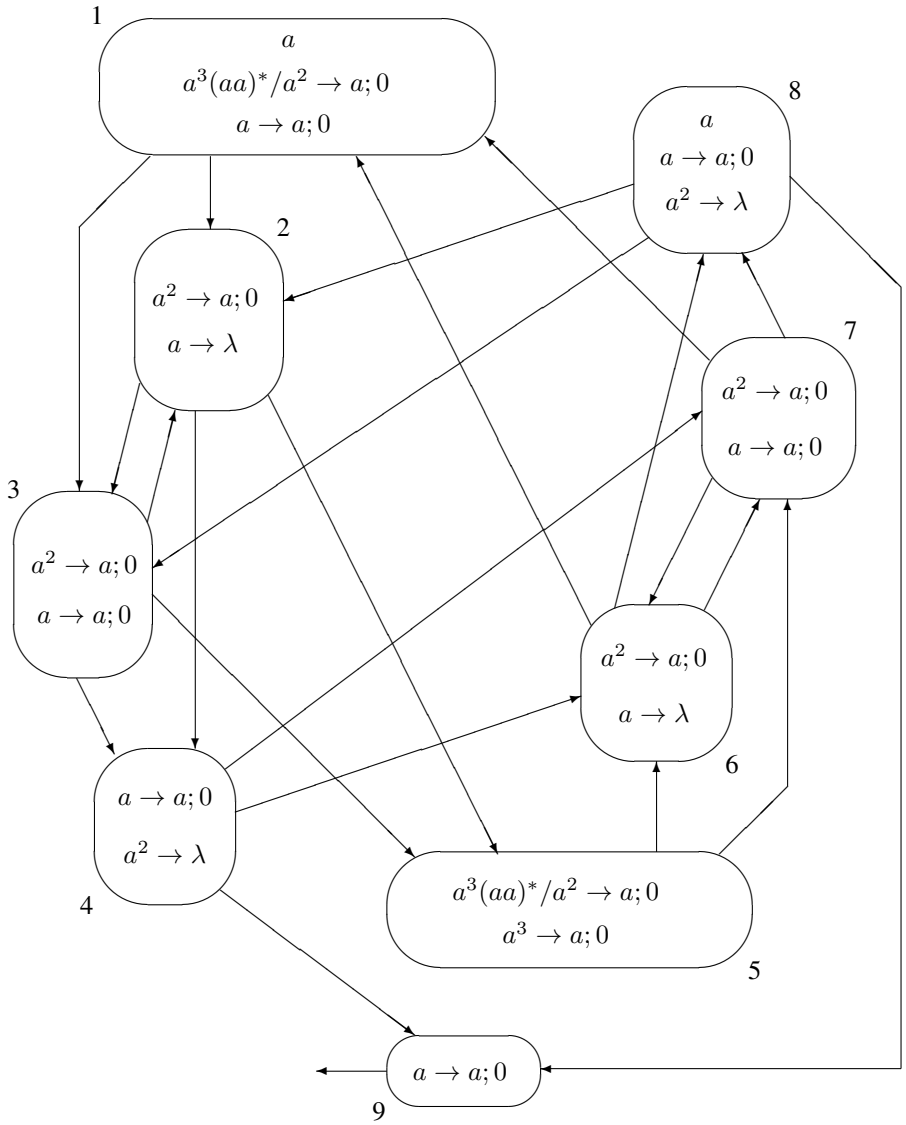

Fig. 3. A SN P system computing a complex infinite sequence

fire. This means that again a spike is sent to the output neuron, while triggering spikes are sent to neurons 2 and 3, like at the beginning of the computation. Therefore, on the one hand, the system sends one spike out, $(n + 1) + 1$ steps after the previous spike, and, on the other hand, the procedure of moving the contents of neuron 1 into neuron 5 is restarted.

The process is iterated, always increasing by 2 the number of spikes when passing from neuron 1 to neuron 5, and spiking after completing each move. Thus, the spike train generated

in this way is

$$bst(\Pi_4) = 0^4 10^3 10^5 10^4 10^6 10^5 10^7 10^6 \ldots,$$

that is, an infinite sequence of blocks of the form $0^{2i}10^{2i-1}10^{2i+1}10^{2i}1$ with $i \geq 2$.

We can write this sequence also in the following manner. Consider the partial function $g : \{0,1\}^* \longrightarrow \{0,1\}^*$ defined by

$$g(0^i 10^j 1) = 0^{i+1}10^{j+1}1,$$
$$g(w10^i 10^j 1) = 0^{i+1}10^{j+1}1,$$

for all $w \in \{0,1\}^*$ and $i, j \geq 1$, and then define the infinite sequence f_ω as the limit of the following sequence of strings:

$$f_0 = 0^4 10^3 1,$$
$$f_{n+1} = f_n \; g(f_n), \text{ for } n \geq 0.$$

The equality $bst(\Pi_4) = f_\omega$ can be easily proved.

This above example suggests the interesting research question of comparing the sequences generated by SN P systems with sequences of bits from the literature (see [13]), in particular, with the so-called self-reading sequences. The sequence f_ω above can be considered as such (it grows by appending to its right end a string which can be interpreted as a reading of the previous stage of the sequence itself (although in our case the reading is partial, as the mapping g takes into consideration only a well specified suffix of the previous sequence). Also, the relationship to D0L sequences or other sequences from the Lindenmayer systems area (see, e.g., [11]) is worth investigating – with the mentioning that, as we will see in Section 7., there are several limitations in computing (sequences obtained by iterated) morphisms.

We consider two more examples, which will be useful in Section 7., namely generating the periodic strings of the forms x^k and x^ω, respectively, for some $x \in \{0,1\}^+$ and $k \geq 1$.

Actually, it is sufficient to find a system Π which generates x^ω, because we can supplement Π with a module which stops its spiking out after a specified number of spikes.

Such a module is given in Figure 4. As long as neuron c_1 has not accumulated k spikes, it does not fire, but after having k spikes, received from the output neuron i_0 of Π, it sends two spikes to neuron out, which will never spike again.

Now, because we know how many occurrences of 1 are in x, having a system Π such that $bst(\Pi) = x^\omega$, we can construct a system Π_k such that $bst(\Pi_k) = x^k$.

A SN P system generating x^ω for a given string x is given in Figure 5. The construction includes two modules, the one-step-one-spike loader composed of neurons 1 and 2, already used in some of the previous examples, and the module LOADER(s, n),given in Figure 6 which provides s spikes in each step $1, n+1, 2n+1, \ldots$.

Let us write the string x as

$$x = 0^{n_1}10^{n_2}1 \ldots 0^{n_r}10^{n_r+1},$$

with $n_i \geq 0$ for $1 \leq i \leq r+1$, and let

$$\sum_{i=1}^{r+1} n_i = N.$$

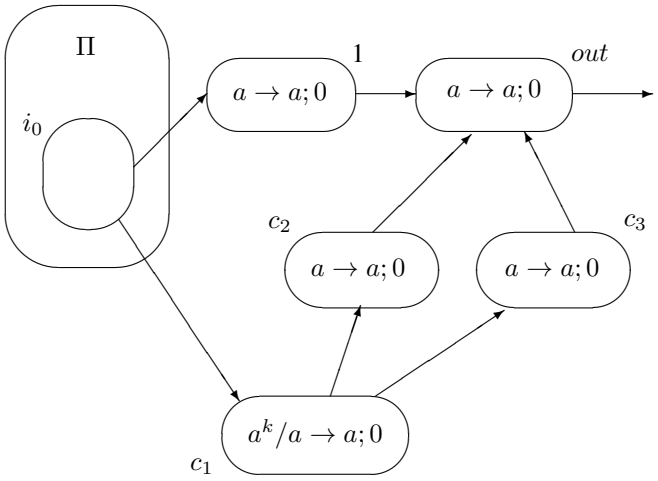

Fig. 4. A module which stops the spiking out of system Π after k spikes

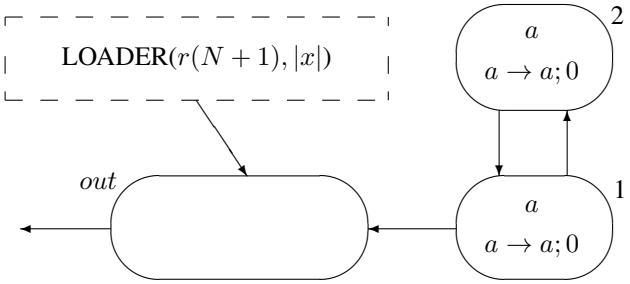

Fig. 5. A SN P system generating x^ω

We introduce in neuron out the following rules:

$$a^{\alpha_j}/a^{N+1} \to a; 0,$$

for all $j = 1, 2, \ldots, r$, where

$$\alpha_j = (r - j + 1)(N + 1) + (j - 1) + \sum_{i=1}^{r} n_i,$$

as well as the rule

$$a^{N+r} \to \lambda.$$

The neuron out of the system in Figure 5 contains initially no spike. In step 1 it receives $r(N + 1)$ spikes from the loader and one from neuron 1, and it gets one spike from neuron 1 in each step. After n_1 spikes coming from neuron 1 were introduced, neuron *out* contains α_1

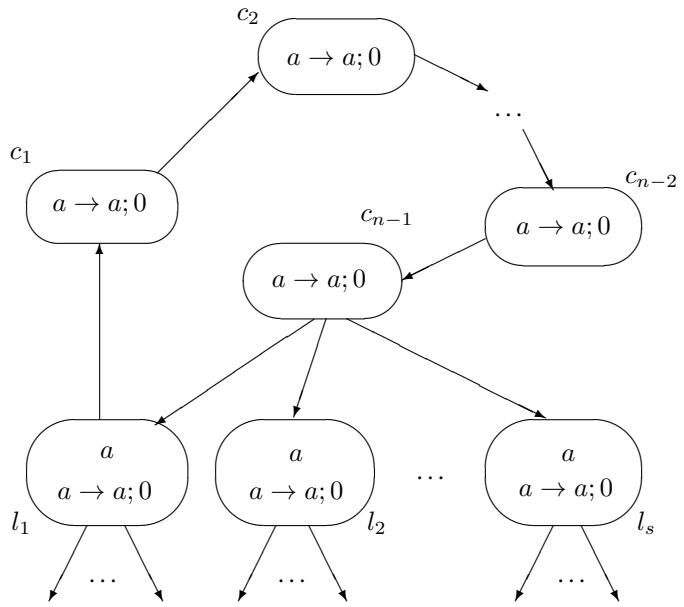

Fig. 6. The module LOADER(s, n)

spikes, hence the rule $a^{\alpha_1}/a^{N+1} \to a; 0$ can be used. It removes $N + 1$ spikes, and at the same time it gets one more spike from neuron 1. We continue n_2 steps by receiving spikes from neuron 1, and at that moment we have in neuron out the following number of spikes:

$r(N + 1) + n_1$	at the moment of the first spike,
$-(N + 1)$	used by the first firing rule used, and
$+1 + n_2$	introduced by neuron 1 in the meantime,

Hence, altogether we have in neuron out

$$r(N + 1) - (N + 1) + 1 + (n_1 + n_2) = \alpha_2$$

spikes. Thus, the rule $a^{\alpha_2}/a^{N+1} \to a; 0$ has to be used, and no other rule was applicable before. It is clear that continuing in this way, the system spikes at the right moments, thus producing the string x symbol by symbol, until the rule $a^{\alpha_r}/a^{N+1} \to a; 0$ is used. This process is correct, because we have:

1. $n_j < N + 1$ for all $j = 1, 2, \ldots, r + 1$, hence
2. $\alpha_j > \alpha_{j+1}$ for all $j = 1, 2, \ldots, r$ (easy to check), which implies that
3. all $\alpha_j, j = 1, 2, \ldots, r$, are different from each other.

Now, let us count the number of spikes present in the system after introducing the last digit 1 of x. We introduce in the neuron out $r(N + 1)$ spikes from the loader and $|x| - n_{r+1}$ spikes from neuron 1, in total $r(N + 1) + r + N - n_{r+1}$ spikes, and we consume $r(N + 1)$ of them by using the firing rules. Hence at the end we have $r + N - n_{r+1}$ spikes. No rule can be applied until

adding step by step further n_{r+1} spikes, thus completing the string x. At that moment, we have in neuron out $N + r$ spikes, hence we can use the rule $a^{N+r} \to \lambda$ and in this way we forget all these spikes. This happens in step $|x| + 1$, hence neuron out is again empty as at the beginning of the computation, and the process can be iterated.

Consequently, $bst(\Pi) = x^\omega$, as desired.

5. SN P Systems as Transducers

In this section we consider SN P systems as transducers of infinite sequences. To this aim, we need to equip those systems with a way to get inputs, in the form of sequences of bits, with 1 indicating that a spike is introduced from the environment into the system, and with 0 indicating a step when no spike comes from the environment.

The idea is rather simple: besides the output neuron out we also consider a neuron in (which can be the same as out, in which case we denote it by in/out), where the input spikes are introduced. The only restriction we impose is that neuron in can never be closed, that is, all its firing rules are of the form $E/a^c \to a; d$ with $d = 0$. We call such a device an *SN P transducer*. In graphical representations of SN P transducers, we will indicate the input neuron with an incoming arrow, pointing from the environment to neuron in.

The translation of a sequence $w \in \{0, 1\}^\omega$ by an SN P transducer Π is done as follows. In each step, one digit of w is "read" and if the currently read digit is 1, then one spike is introduced in neuron in (this spike is added to the possible spikes already existing in neuron in or received at the same step from other neurons), while if the current digit of w is 0, then no spike is introduced into neuron *in* from the environment. At each step Π applies its rules to the spikes existing in its neurons, and it either sends a spike into the environment (through neuron out) or it does not. The infinite sequence of digits describing the firing moments of the output neuron is the *translation* of the input sequence w, and it is denoted by $\Pi(w)$. If Π is deterministic, then $\Pi(w)$ is a unique sequence, otherwise $\Pi(w)$ is the set of all possible translations of w, through all non-deterministic behaviors of Π. Note that we do not rule out the possibility that Π gets stuck and hence never sends out a further spike, in such a case $\Pi(w)$ continues with $000\ldots$, irrespective of the input sequence w.

In all translations considered below we have a delay between the moment when a digit is read (either a spike enters the system or no spike enters) and the moment when its translation is sent to the output: even if the system has only one neuron, a spike which enters in moment 1 can exit only in moment 2. This means that the translation of a sequence starts with some occurrences of 0 which do not correspond to "real" input symbols.

We see two possible solutions to this "time-difference-problem" (delay between the input and the output sequences): (1) to accept – and then ignore – a prefix of the form 0^d of the output, with a specified d which represents the input-output delay, or (2) to design the transducer in such a way that the first digits of the output are relevant and correct, meaning that even if we need some steps for processing the first input digit, we work in an anticipatory manner, by outputting some tentative bits and then checking whether they are correct with respect to the input. Because we do not have an implementation of this second idea (it assumes a way to abort the translation in the case of "wrong guess"), in what follows we will use a dummy prefix 0^d which is explicitly mentioned whenever necessary.

The simplest SN P transducer for which we can illustrate this idea is the one with only one

neuron and $a \to a; 0$ as its rule. It takes a spike and sends it out in the *next* step, hence it computes the identity, modulo one additional 0 in front of each translation: $\Pi(0^k 1w) = 0^{k+1}1w$ for all $k \geq 0$ and all $w \in \{0,1\}^\omega$.

A similar meaningless initial 0 appears in the translations done by the system in the following example, which computes the Boolean function NOT (negation). The system is given in Figure 7 and it functions as follows. At each moment, one spike is sent to neuron in/out from neuron 1. If at the same time a spike comes from the environment, then the two spikes are forgotten, but if no spike comes from the environment, then neuron in spikes. Therefore, for any $w \in \{0,1\}^\omega$ we get $\Pi(w) = 0\,\mathtt{NOT}(w)$, where NOT is the morphism defined by $\mathtt{NOT}(0) = 1,\ \mathtt{NOT}(1) = 0$.

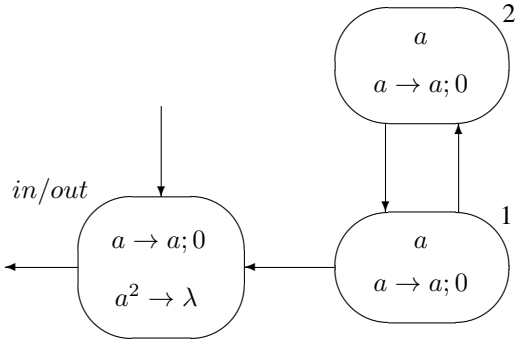

Fig. 7. A SN P transducer computing the Boolean NOT

The NOT function is unary, so one input neuron suffices to provide the argument for it. The situation is different when one considers functions of two (or more) variables. In general, one would need two (or more) input neurons, each of which represents (inputs) one variable. One input neuron cannot distinguish between the bits coming from two input sequences, unless the considered function is commutative.

Because this is the case for OR, AND, XOR and NAND functions, each of them can be computed by SN P transducers with only one input neuron. For the first three operations, we need transducers with only one neuron, the in/out one, empty at the beginning and containing the following rules:

Function	Rules
OR	$a \to a; 0,\ \ a^2 \to a; 0$
AND	$a^2 \to a; 0,\ \ a \to \lambda$
XOR	$a \to a; 0,\ \ a^2 \to \lambda$

As discussed already, the output sequence always starts with one meaningless 0, independently of the input bits.

In the case of NAND we need to resolve a small problem, because for two inputs equal to 0 we have to output 1, and this is possible only if we have a spike inside. Such a spike is not provided by the input sequences, but it can be provided by a couple of neurons such as 1 and 2 in Figure 7. Actually, the system looks exactly the same as the one in Figure 7, with neuron in/out containing the following rules:

$$a \to a; 0,\ \ a^2 \to a; 0,\ \ a^3 \to \lambda.$$

For **non-commutative functions** one has to distinguish between the inputs coming from various sequences (variables), and one has to consider several input neurons. Thus, for computing a function $f : \{0,1\}^k \longrightarrow \{0,1\}$ we provide k input neurons, labeled by $in_1, in_2, \ldots, in_k$, where neuron in_i receives the ith argument of the function.

Then, not surprisingly, the following general result holds.

Theorem 1. *Any function $f : \{0,1\}^k \longrightarrow \{0,1\}$ can be computed by an SN P transducer with k input neurons (and additional $2^k + 2$ neurons, one of which is the output neuron).*

Proof. Given a function f as above, we consider the system consisting of the following neurons:

1. k *input neurons* $in_i, 1 \leq i \leq k$, each of them containing the rule $a \to a; 0$;

2. each neuron in_i is linked by synapses to 2^{i-1} neurons, $1 \leq i \leq k$, each of them containing the rule $a \to a; 0$; we call them *intermediate neurons*;

3. two *auxiliary neurons*, similar to neurons 1 and 2 from Figure 7, which use at each time unit the rule $a \to a; 0$, thus providing each other with a spike;

4. two *delaying neurons*, placed in between one of the auxiliary neurons and the output neuron, which ensure that the spike of the auxiliary neuron reaches neuron out with a delay of three steps;

5. all the intermediate neurons as well as one of the auxiliary neurons have a synapse to the *output neuron*, neuron out, which contains the following 2^k rules:

$$a^{\sum_{i=0}^{k-1} 2^i \cdot b_{i+1}+1} \to a; 0, \text{ if } f(b_1, b_2, \ldots, b_k) = 1, b_j \in \{0,1\}, 1 \leq j \leq k,$$
$$a^{\sum_{i=0}^{k-1} 2^i \cdot b_{i+1}+1} \to \lambda, \text{ if } f(b_1, b_2, \ldots, b_k) = 0, b_j \in \{0,1\}, 1 \leq j \leq k.$$

All neurons except for the auxiliary neurons are empty at the beginning of the translation (and they become empty after using the local rules, because all these rules consume or forget all existing spikes), while each of the two auxiliary neurons contains initially one spike.

The equality $\Pi(w_1, \ldots, w_k) = 0^3 f(w_1, \ldots, w_k)$, for all $w_j \in \{0,1\}^\omega, 1 \leq j \leq k$, is guaranteed by the fact that each input neuron in_i sends 2^{i-1} spikes to neuron out if and only if the ith input is 1, and by the fact that all numbers $\sum_{i=0}^{k-1} 2^i \cdot b_{i+1}$ are distinct for distinct vectors $(b_1, \ldots, b_k)$. The auxiliary neurons are necessary in the case when $f(0, 0, \ldots, 0) = 1$: a spike can be produced and sent out of the system only if a spike already exists in the output neuron.

The prefix 0^3 of any translation is due to the three steps necessary for both the input spikes and for spikes emitted by the auxiliary neuron to reach the output neuron.

The above construction is illustrated in Figure 8 showing a system which computes the function $f : \{0,1\}^3 \longrightarrow \{0,1\}$ defined by

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

□

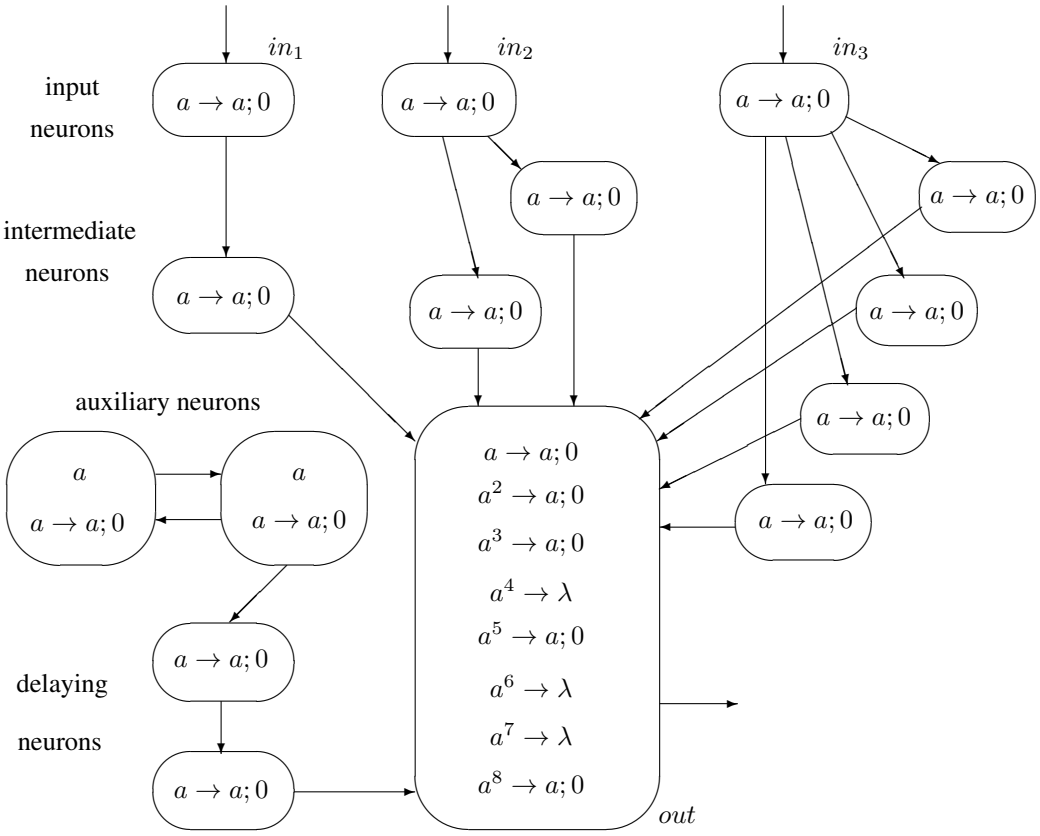

Fig. 8. A SN P transducer computing a Boolean function of three variables

6. A Tool-Kit for Handling Infinite Sequences

In the previous section, it was not essential that an input is infinite sequence, the input bits were processed one by one without taking into account the adjacent bits in the sequence. In this section we consider operations on infinite sequences of bits, which are realized/implemented by SNP transducers.

We begin with an example which uses one of our previous ideas of considering only **the prefix of length** k of an input sequence, for some $k \geq 1$. This means that after preserving the first k bits as they are, all other bits are mapped to 0. This amounts to applying the AND operation to the input sequence and the sequence 1^k0^ω, $k \geq 1$, generated by the systems Π_1 and Π_2 from Figure 1.

Note that a meaningless 0 appears in front of the output. In what follows we will not specify such occurrences of 0, except for cases when it is of a particular interest.

The same idea can be used when we consider **any prefix of length at least 2**, not just a prefix of a given length. We have then to combine the system Π_3 from Figure 2 with an SN P transducer

which implements the AND operation.

Now, rather than preserving prefixes we will consider **removing prefixes** (which can be also seen as **taking suffixes**, where the suffix of an infinite sequence w is the infinite sequence obtained by removing a prefix of w), which now means to map all bits until a given position to 0 and to preserve all other bits as they are. If we want to do this for a fixed number of bits, then we can use again the systems from Figure 1: we apply the operation AND to the input sequence and the negation of $bst(\Pi_1)$. This transforms the first k bits of the input sequence into 0s, and it can be implemented by SN P systems.

For the case of non-deterministically removing an arbitrary prefix we can use the same idea, now using the system Π_3 from Figure 2.

In the above examples we have not considered the specific number of bites 1 from the prefix, but this can be easily achieved by the following system, which **forgets the first k spikes**, for a given $k \geq 1$. The system consists of a unique neuron, used both as input and output neuron, having no spike at the beginning, and the rule $a^{k+1}/a \to a; 0$. Clearly, before accumulating k spikes the rule cannot be used, but from the spike $k+1$ on all spikes are reproduced as the output.

A related operation is that of **preserving one spike and forgetting the next k, repeatedly**, for some $k \geq 1$, which can be realized by a system with only one neuron, containing initially k spikes, as well as the rule $a^{k+1} \to a; 0$. The first spike entering the neuron completes the $k+1$ spikes necessary for using the rule, all spikes are consumed, hence next k spikes are accumulated without spiking, and the cycle is repeated.

A somewhat more complex operation is the **alternate filling with 1**, defined as follows. For sequences of bits starting with 0: from the beginning of the input sequence until the first 1 we output 1, then we output 0 until the next 1, and we repeat this operation (the $(2k+1)$th substring of the form 0^i1 of the input is replaced with 1^{i+1}, for all $k \geq 0$). This operation is realized by the system from Figure 9. We start with one spike in both neurons 1 and out, hence the system spikes in the first step. Neurons 1 and out exchange the spikes, hence neuron out will spike continuously until receiving a spike from neuron in. None of neurons 1 and out have rules for two spikes, hence as long as no further spike comes into the system we output no spike. When the second spike enters the system, we can apply the rule $a^3 \to a; 0$ and again both neurons 1 and out contain only one spike, hence the system will output 1 until receiving one further spike. These operations are repeated indefinitely.

The above operations, and the associated SN P transducers, can be composed in various ways in order to compute more complex operations.

For instance, consider two infinite sequences of bits, w_1 and w_2. The following expression

$$(bst(\Pi_1)\ \texttt{AND}\ w_1)\ \texttt{OR}\ (\texttt{NOT}\ bst(\Pi_1)\ \texttt{AND}\ w_2) \tag{1}$$

is computable by a SN P transducer and its result is *the concatenation of a prefix of length k of sequence w_1 with the sequence obtained by removing the prefix of length k of w_2*. If instead of Π_1 we use the system Π_3, then we obtain all infinite sequences which are the concatenation of a prefix of w_1 with a suffix of w_2, without necessarily jumping from the ith bit of w_1 to the $(i+1)$th bit of w_2.

A variant of expression (1) can specify the operation of **guided crossing over** of two sequences under the control of another sequence.

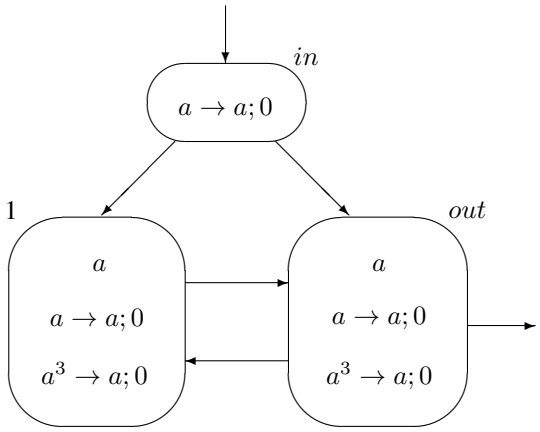

Fig. 9. A SN P transducer filling alternately with 1 the input sequence

Indeed, let us consider three sequences $w_0, w_1, w_2 \in \{0,1\}^\omega$ and define the sequence w_3 obtained as follows. We start to read step by step the three sequences, moving each bit of w_1 to w_3 as long as w_0 has in the corresponding positions the bit 0. Then at the first bit 1 from w_0 we switch to w_2 and move its bits to w_3 as long as no further 1 appears in w_0, and when this happens, we switch back to w_1, and so on. Therefore, w_3 is composed of blocks of w_1 and w_2 which correspond to substrings 0^i1 of w_0 of maximal length, jumping alternately from w_1 to w_2 and back.

Formally, this operation is specified by the following expression, where ALT is the operation of alternately filling with 1 the substrings 0^i1 of a string, as done by the transducer from Figure 9:

$$(\texttt{ALT}(w_0)\ \texttt{AND}\ w_1)\ \texttt{OR}\ (\texttt{NOT}\ \texttt{ALT}(w_0)\ \texttt{AND}\ w_2). \tag{2}$$

As all operations involved in expression 2 can be realized by SN P transducers, also this whole operation can be realized. Actually, a SN P transducer simpler than the one provided by expression (2) can be designed for the guided crossing over operation. For example, such a transducer is presented in Figure 10, where input neuron in_i reads the sequence w_i, $i = 0, 1, 2$.

7. Computing Morphisms

A standard way to process (finite and) infinite strings is by means of morphisms. In our case, because we work with the binary alphabet, we consider the following classes of morphisms.

1. *Length preserving* (also called *codings*). They are Boolean operations of four sorts: constant 0, constant 1, identity, and negation. All of them can easily be realized by SN P transducers (we have already discussed the case of negation).

2. *Erasing*. If, say, $h(0) = \lambda$ and $h(1) \neq \lambda$, then for any sequence $w \in \{0,1\}^\omega$ we have either $h(w) = h(1)^k$, if w contains only k occurrences of 1, or $h(w) = h(1)^\omega$, if w contains infinitely many occurrences of 1. Therefore, the translation of w can be done

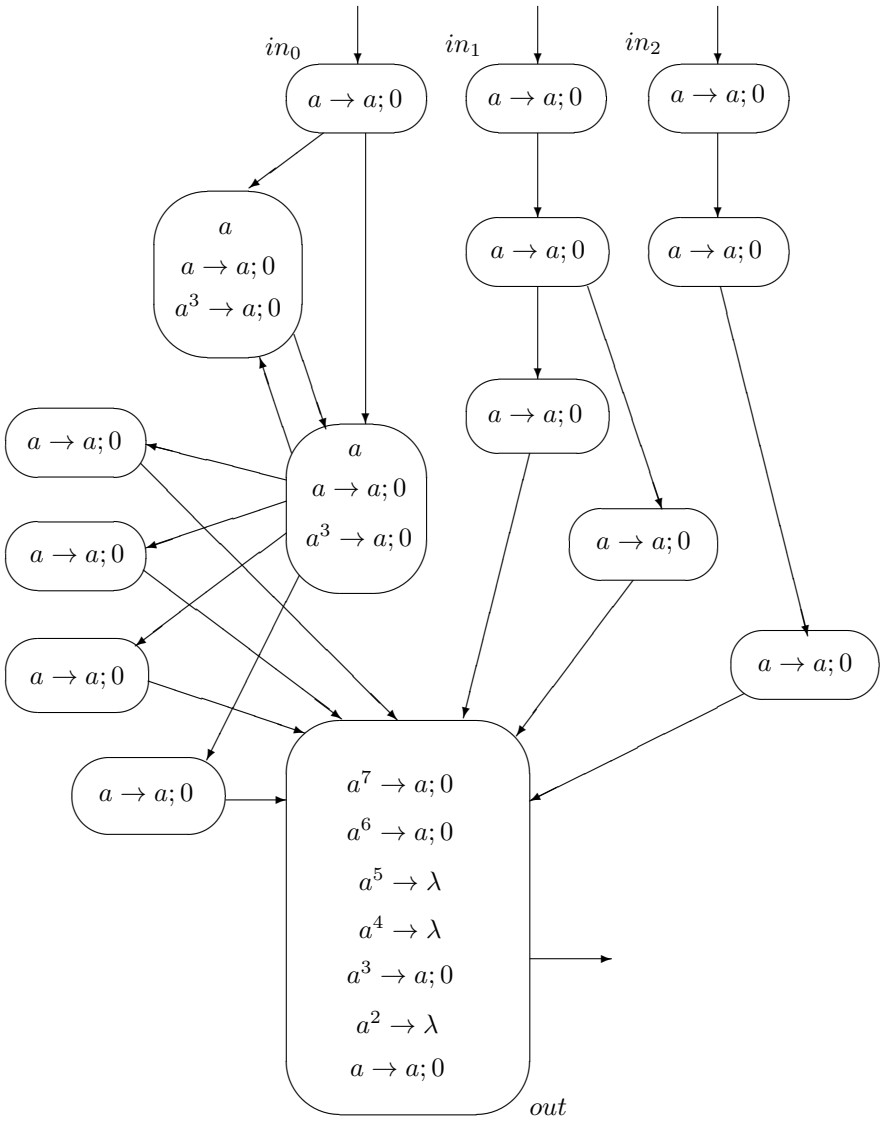

Fig. 10. A SN P transducer computing the guided crossing over

independently of the symbols of w, taking only into account the number of occurrences of 1 in w, which amounts to computing strings of the forms x^k and x^ω. Such strings can be generated by SN P systems, as was already shown at the end of Section 4..

3. *Non-erasing and non-length-preserving*. In this case both $h(0)$ and $h(1)$ are nonempty, and at least one of them has the length at least 2. If $h(0) = u^i$ and $h(1) = u^j$ for some $u \in \{0,1\}^+$ and $i, j \geq 1$, then $h(w) = u^\omega$, as in the erasing case above, and this sequence

can be generated by an SN P system irrespectively of its input. If, however, $h(0)$ and $h(1)$ are not powers of the same word, then the mapping h cannot be computed by an SN P transducer – see the theorem below.

Theorem 2. *Let $h : \{0,1\}^* \longrightarrow \{0,1\}^+$ be a morphism such that:*

1. *$|h(1)| = r \geq 2$,*
2. *$h(0) = u^i$ and $h(1) = u^j$ for some $u \in \{0,1\}^+$ and $i, j \geq 1$ does not hold.*

Then, there is no SN P transducer Π such that $\Pi(w) = 0^s\, h(w)$ for any given $s \geq 0$ and all $w \in \{0,1\}^* \cup \{0,1\}^\omega$.

Proof. Let us assume that there is an SN P transducer Π which computes h for all $w \in \{0,1\}^* \cup \{0,1\}^\omega$. Consider an infinite sequence w which is of the form $w = 1^k w'$ for some k and $w' \in \{0,1\}^\omega$. We have $h(w) = h(1^k)h(w') = \Pi(1^k)\Pi(w')$, with $|\Pi(1^k)| = k|h(1)| = rk$, possibly with an additional prefix 0^s of $\Pi(1^k w')$ for a fixed $s \geq 0$ which does not depends on the input sequence. This means that when reading the kth occurrence of 1 we have already in the system the information about the first $rk + s$ output digits. Consider the possible continuations of w, after 1^k, until the position $rk + s$. There are $(r-1)k + s$ positions to fill in with the two symbols 0, 1, hence there are $2^{(r-1)k+s}$ strings $z \in \{0,1\}^*$ of length $(r-1)k+s$ such that $1^k z z'$ can be the input of Π. Their symbols should be already read by the transducer at the moment $rk + s$ (there is no step when we do not advance in the input sequence), hence the system should "remember" which were the $(r-1)k + s$ symbols read after 1^k. This reasoning is illustrated in Figure 11.

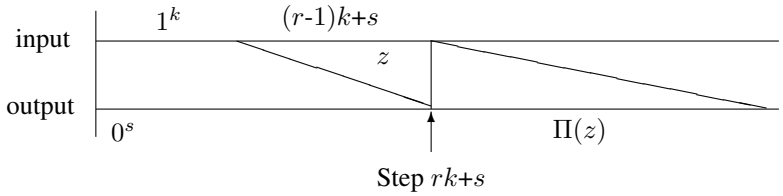

Fig. 11. The reasoning in the proof of Theorem 2

The information about the input symbols can be encoded in the system configurations, so let us estimate the number of configurations of the system after $rk + s$ steps.

A configuration is described by the number of spikes in each neuron and by the open-close status of each neuron, in the sense of the time interval until the neuron will be open again after using a firing rule $E/a^c \to a; d$ with $d \geq 1$.

We start with a total of n_0 spikes present initially in the system. During the $rk + s$ steps we have received at most further $rk + s$ spikes from the input sequence. Each rule which is used in the system consumes at least one spike, but it may produce one spike (the case of the firing/spiking rules). The newly produced spikes can also be replicated, in the case of multiple synapses leaving a neuron. Let o be the maximal out-degree of the synapse graph. Each spiking rule consumes at least one spike and introduces in the system at most o spikes, hence the increase is of at most $o - 1$ spikes for each application of a rule. In each step, at most m rules are applied, where m is the number of neurons in Π. This means that in each step one produces at most

$m(o-1)$ new spikes. For $rk+s$ steps this means a total of at most $m(d-1)(rk+s)$ spikes. Adding the initial and the input spikes, we get at most $n_0 + rk + s + m(d-1)(rk+s)$ spikes. Let us denote this number by $N(k)$.

Therefore, Π can contain any number of spikes from 0 to $N(k)$. The number of possible distributions of q spikes in the m neurons of the system is bounded by q^m. Thus, the number of spikes distributions is bounded by

$$\sum_{i=0}^{N(k)} i^m < \sum_{i=0}^{N(k)} N(k)^m = N(k)^m(N(k)+1).$$

In order to obtain a bound on the number of configurations, we also have to take into account the moments until some neurons are open after using spiking rules with delays. Let D be the maximal d such that $E/a^c \to a; d$ is a rule in Π. The spikes can be placed in any moment from 0 to d after using such a rule, which means at most $(D+1)^m$ possibilities for m neurons.

Consequently, the number of possible configurations is bounded by

$$(D+1)^m \cdot N(k)^m \cdot (N(k)+1),$$

which is a polynomial in k (all parameters D, m, r, o, s are fixed, depending on Π). Therefore, there is k such that the number of configurations of the system in step $rk+s$ is strictly smaller that the number of strings $z \in \{0,1\}^*$ of length $(r-1)k+s$. This means that there are two strings z_1, z_2 of this length, $z_1 \neq z_2$, such that the system Π reaches the same configuration after $rk+s$ steps. Consider then the infinite sequences $w_1 = 1^k z_1 0^\omega$ and $w_2 = 1^k z_2 0^\omega$. Because the system is in the same configuration after reading $1^k z_1$ as it is after reading $1^k z_2$, and after that the input sequence continues in the same way in w_1 and w_2, it follows that $\Pi(w_1) = \Pi(w_2)$. However, the condition that $h(0)$ and $h(1)$ are not powers of the same word implies that $h(z_1) \neq h(z_2)$ (if $h(z_1) = h(z_2)$ for $z_1 \neq z_2$, then, as a consequence of the so-called defect theorem, see [2] or [5], it follows that $h(z_1)$ and $h(z_2)$ are powers of the same word, which is assumed not to be the case). Therefore we have $h(w_1) = h(1)^k h(z_1) h(0^\omega) \neq h(1^k) h(z_2) h(0^\omega) = h(w_2)$, which contradicts the fact that $\Pi(w_j) = 0^s\, h(w_j), j = 1, 2$. Hence the assumption from the beginning of the proof that Π exists leads to a contradiction. Thus the theorem holds. □

Because length preserving morphisms can be computed by SN P transducers, it is natural to consider other length preserving functions, for instance, defined for blocks of a given length. More specifically, for a given $k \geq 1$, a *k-block morphism* is a function $f : \{0,1\}^k \longrightarrow \{0,1\}^k$ extended to a function $\hat{f} : \{0,1\}^\omega \longrightarrow \{0,1\}^\omega$ by

$$\hat{f}(x_1 x_2 \dots) = f(x_1) f(x_2) \dots,$$

for all $x_1, x_2, \dots \in \{0,1\}^k$ (for $k = 1$ we get the usual notion of a morphism).

We consider now the case of $k = 2$.

Theorem 3. *For each 2-block morphism $f : \{0,1\}^2 \longrightarrow \{0,1\}^2$ there is an SN P transducer Π such that $\Pi(w) = 0^5 \hat{f}(w)$, for all $w \in \{0,1\}^\omega$.*

Proof. Let f be as in the statement of the theorem. Let Π be a SN P transducer shown in Figure 12, where some of the "sub-systems" are given only by their names. This is the case for two types of modules, which we describe separately.

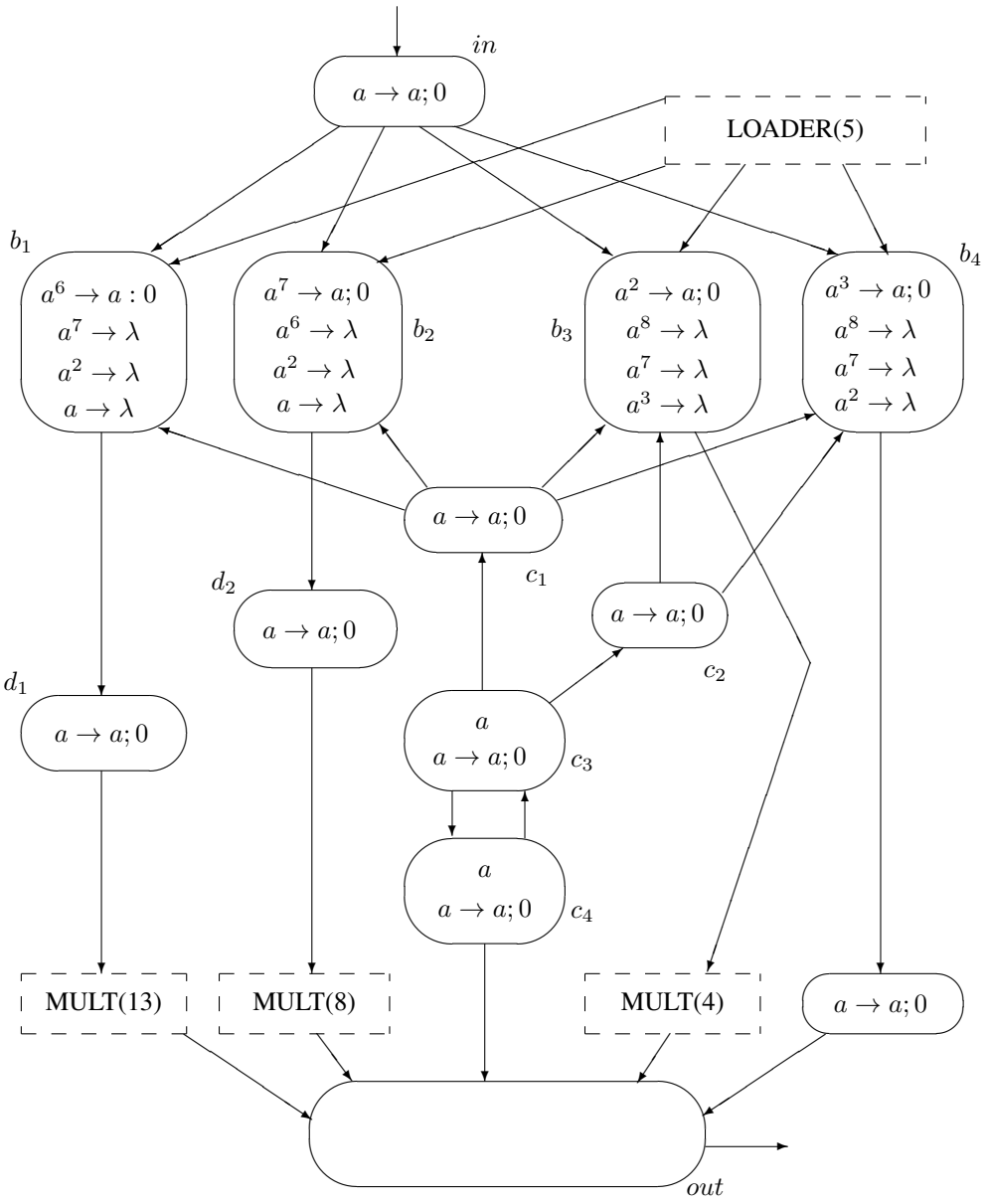

Fig. 12. A SN P transducer which computes a 2-block morphism

The module LOADER(5) is almost the same as module LOADER(5,1) in Figure 6 with a small difference: the first 5 spikes should arrive in neurons b_1, b_2, b_3, b_4 in step 2. Because of this difference and because the module we need now is simpler than the one from Figure 6, we present its construction in Figure 13, for the general case of sending in each step s spikes to specified

neurons, starting with step 2. The functioning of this module is obvious: the neurons 1 and 2 just reload each other continuously, also sending one spike to each of the neurons $l_1, \ldots, l_s$, which, in turn, send as many spikes (at most s) to any other neuron in the system.

In Figure 12, the LOADER(5) is used to introduce in each step, starting with the second one (hence simultaneously with the arrival of the input spike – if any), five spikes to each of the neurons b_1, b_2, b_3, and b_4.

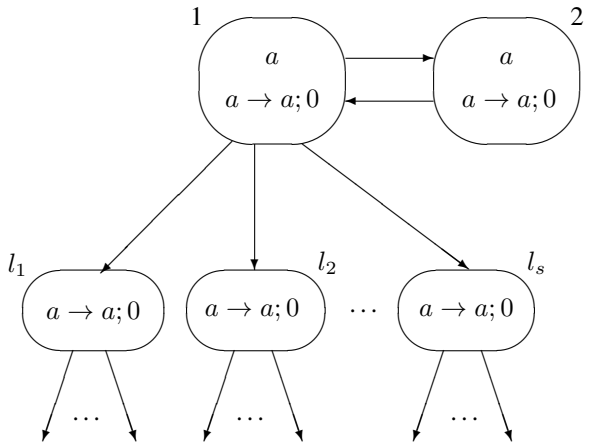

Fig. 13. The module LOADER(s)

The module MULT(s) is a multiplier, used already in previous constructions, including the one in Figure 13. It consists of s neurons $m_1, \ldots, m_s$ which receive simultaneously a spike from the same neuron, thus replicating this spike in s copies, which then can be transmitted to other neurons.

In the system in Figure 12, we multiply in this way the spikes sent by neurons b_1, b_2, b_3, b_4 towards neuron out. More specifically, the spike from b_1 is replicated in 13 copies, the one from b_2 is replicated in 8 copies, the one from b_3 in four copies, and the spike from b_4 is replicated in only one copy.

Let us briefly examine the work of this transducer. The basic idea is that neurons b_1 and b_2 "memorize" the first bit of each block of length 2, and the neurons b_3, b_4 "memorize" the second bit. Moreover, b_1 fires if the first bit is 0, b_2 fires if the first bit is 1, b_3 fires if the second bit is 0, and b_4 fires if the second bit is 1.

The neurons distinguish between the values of the respective bits with the help of the spikes received from the loader, which sends 5 spikes at each second moment of time, and the spikes received from neurons c_1 and c_2. Note that each of b_1 and b_2 receives one spike, each of b_3, b_4 receives two spikes from c_1 and c_2, and this happens in each time unit.

In the first step, a bit is read. If it is 1, then a spike reaches the neurons b_1, b_2, b_3, and b_4 at the same time with the spikes coming from the loader and neurons c_1, c_2. If the first bit is 0, then only b_1 fires and spikes: b_1 and b_2 have six spikes each, while b_3 and b_4 have 7 spikes each. If the first bit is 1, then neurons b_1, b_2 have 7 spikes each, hence b_2 fires and spikes, while b_3 and b_4 have 8 spikes each – they forget the spikes.

Therefore, when reading the first bit, one of b_1, b_2 fires, and all other neurons from b_1, b_2, b_3, and b_4 get empty.

When reading the second bit, only c_1, c_2 send spikes to the four neurons b_1, b_2, b_3, and b_4, not also the loader. If the second bit is 0, then neurons b_1 and b_2 have only one spike and forget it, while neurons b_3 and b_4 contain 2 spikes, and b_3 fires. Similarly, if the second bit is 1, the neuron b_4 fires, while b_1, b_2, and b_3 forget their spikes.

In the next step, hence at the beginning of a new block of two bits, the loader sends again 5 spikes to all neurons b_i, hence the cycle is repeated.

Now, let us follow the path of the spikes through the system. The spikes of the neurons b_1 and b_2 need three steps for reaching the output neuron, while the spikes of neurons b_3 and b_4 need only two steps. This means that these spikes arrive in neuron out at the same time – multiplied by the respective multipliers in 13, 8, 4, 1 copies, respectively.

Because only one of b_1, b_2 and only one of b_3, b_4 can fire for each block, neuron out receives from neurons b_1, b_2, b_3, and b_4 one of the following combinations of numbers of spikes:

1. $13 + 4 = 17$,
2. $13 + 1 = 14$,
3. $8 + 4 = 12$,
4. $8 + 1 = 9$.

In each time unit, neuron out also receives a spike from neuron c_4.

We introduce in neuron out the rule

$$a \to \lambda,$$

hence all spikes coming from c_4 are forgotten, except for those which arrive at the same time with the spikes from the multipliers. Therefore, we accumulate in the output neuron one of the following numbers of spikes:

$$18, 15, 13, 10.$$

These numbers identify precisely the block which was just read. Depending on the number of spikes present in neuron out, we output either 0 or 1, using rules as follows.

Knowing the function f, we know for which two of the values $18, 15, 13, 10$ (if any) we have to output 1, and for those values x_1, x_2 we introduce the rules

$$a^{x_1}/a^7 \to a; 0, and$$
$$a^{x_2}/a^7 \to a; 0.$$

This means that two of the values $18, 15, 13, 10$ will be decreased by 7, thus they belong to the set $\{11, 8, 6, 3\}$. Simultaneously, neuron c_4 has sent one more spike to the output neuron, hence these numbers were increased by one. If we do not have to output 1, then none of the rules above is used, hence the number of spikes remains as before, and it is increased by one because of the spike from c_4. This means that the possible number of spikes in neuron out belongs to the set $\{12, 9, 7, 4, 19, 16, 14, 11\}$. For at most two of these values we have to output 1, as requested by the mapping f, and for those values x_3, x_4 we introduce in neuron out the rules

$$a^{x_3} \to a; 0, and$$
$$a^{x_4} \to a; 0.$$

For all other values z from the set $\{12, 9, 7, 4, 19, 16, 14, 11\}$ we introduce the rules

$$a^z \to \lambda.$$

In this way, all spikes from neuron out are removed (consumed by spiking rules or forgotten), while the output is as requested by the mapping $\hat{f}$.

In the meantime, new spikes have left neurons b_1, b_2, b_3, and b_4, but they do not interfere with those emitted for the previous block, because of the alternating firing of either one of b_1, b_2 or one of b_3, b_4 when reading an input bit.

After reading the first bit and before possibly spiking for the first time, Π needs five steps, that is why we have $\Pi(w) = 0^5\hat{f}(w)$, for all $w \in \{0, 1\}^\omega$. □

Acknowledgement. Thanks are due to Ion Petre for useful bibliographical hints.

References

[1] B. ALBERTS, A. JOHNSON, J. LEWIS, M. RAFF, K. ROBERTS, P. WALTER, *Molecular Biology of the Cell*, 4th ed. Garland Science, New York, 2002.

[2] C. CHOFFRUT, J. KARHUMÄKI, *Combinatorics on words*, In [12], vol. I, pp. 329–438.

[3] W. GERSTNER, W. KISTLER, *Spiking Neuron Models. Single Neurons, Populations, Plasticity*, Cambridge Univ. Press, 2002.

[4] M. IONESCU, Gh. PĂUN, T. YOKOMORI, *Spiking neural P systems*, Fundamenta Informaticae, **71** (2-3), pp. 279–308.

[5] M. LOTHAIRE, *Combinatorics on Words*, Addison-Wesley, Reading, Mass., 1983.

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

[7] W. MAASS, C. BISHOP, *Pulsed Neural Networks*, MIT Press, Cambridge, 1999.

[8] Gh. PĂUN, *Membrane Computing – An Introduction*, Springer-Verlag, Berlin, 2002.

[9] Gh. PĂUN, M.J. PÉREZ-JIMÉNEZ, G. ROZENBERG, *Spike trains in spiking neural P systems*, International Journal of Foundations of Computer Science, **17** (4), pp. 975–1002, 2006.

[10] D. PERRIN, J.-E. PIN, *Infinite Words*, Elsevier, Amsterdam, 2004.

[11] G. ROZENBERG, A. SALOMAA, *The Mathematical Theory of L Systems*. Academic Press, New York, 1980.

[12] G. ROZENBERG, A. SALOMAA, *Handbook of Formal Languages*, 3 volumes. Springer-Verlag, Berlin, 1997.

[13] N.J.A. SLOANE, S. PLOUFFE, *The Encyclopedia of Integer Sequences*, Academic Press, New York, 1995.

[14] The P Systems Web Page at PPage: `http://ppage.psystems.eu/index.php/Home/`.