Small Universal Spiking Neural P Systems

Andrei Păun[1], Gheorghe Păun[2]

[1] Department of Computer Science
Louisiana Tech University, Ruston
PO Box 10348, Louisiana, LA-71272 USA
apaun@latech.edu

[2] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucharest, Romania
and
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
george.paun@imar.ro, gpaun@us.es

Summary. In search for small universal computing devices of various types, we consider here the case of spiking neural P systems (SN P systems), in two versions: as devices computing functions and as devices generating sets of numbers. We start with the first case and we produce a universal spiking neural P system with 84 neurons. If a slight generalization of the used rules is adopted, namely, we allow rules for producing simultaneously several spikes, then a considerable improvement, to 49 neurons, is obtained. For SN P systems used as generators of sets of numbers, we find a universal system with restricted rules having 76 neurons, and one with extended rules having 50 neurons.

1 Introduction

Looking for small universal computing devices is a natural and well investigated topic in computer science, see, e.g., [4], [11], and the references therein. Recently, this issue started to be considered also in membrane computing; first contributions of this kind can be found in [13] (for tissue P systems with string objects processed by splicing operations) and [1] (for symport/antiport P systems).

In the present paper, we address the case of the recently introduced (see [3], [9], [10]) *spiking neural P systems* (in short, SN P systems), computing models which bring into membrane computing (see [8] for an introduction and [15] for updated information) ingredients from neural computing by spiking (see, e.g., [5], [6]).

In short, an SN P system consists of a set of *neurons* placed in the nodes of a directed graph (representing *synapses*) and sending to each other *spikes*, identical electrical impulses; the distance between consecutive spikes is the main way to

encode information; the neurons contain rules for emitting spikes and for forgetting spikes; one neuron is distinguished as the *output* neuron and its spikes also exit into the environment, thus producing a *spike train*.

Such systems can be used as computing devices in various ways; generating sets of numbers (encoded in the number of steps between consecutive spikes sent into the environment by the output neuron), generating strings (the spike train itself is such a string over the binary alphabet), or computing functions (an input neuron "reads" the arguments of the function from the environment and the output neuron "writes" the function value, in all cases with the numbers being encoded in the distance between consecutive spikes). More precise definitions will be given in Subsection 2.2.

Here we deal with the first and the third cases, of SN P systems generating numbers and computing functions.

Already in [3], the SN P systems used for computing sets of numbers were proved to be computationally complete (able to compute all Turing computable sets of numbers), but no bound on the number of used neurons was found. The proof from [3] is based on simulating register machines with SN P systems, hence the same strategy as that followed in [1] in search of small universal P systems is useful also here: starting from a small universal register machine as those constructed in [4], we can get a small universal SN P system, with the important mentioning that in [4] one works with register machines computing functions. That is why we also start here with this case.

We work with the so-called strong universality (without encodings of the input and output) and we use the standard type of register machines (using ADD and SUB instructions). For this case, a universal register machine with 8 registers and 23 instructions (the halting one included) was constructed in [4]. (The "code" of the particular partial recursive function and the argument of the function are introduced in registers 1 and 2 of the universal machine and the value of the function for that argument, if defined, is found in register 0 when/if the machine halts.) Following then the construction of an SN P system simulating a register machine from [3], with some improvements inspired from [2] and some additional "code optimization", as well as a suitable input module, we get an SN P system with 84 neurons simulating the register machine from [4].

We may formulate this result as follows: there is a universal "brain" (in the form of an SN P system) with only 84 neurons.

Of course, this number needs to be checked for optimality, but it is our expectation that in the framework used here (with the type of SN P systems considered, i.e., with standard rules) it is not possible to significantly decrease the number of neurons.

However, if we allow rules which produce two or more spikes, then a considerable improvement of the previous result is obtained: 49 neurons are sufficient for universality.

We pass then to the case when SN P systems are used for generating sets of numbers, starting from the observation that a set $Q \subseteq \mathbf{N}$ is recursively enumer-

able if and only if the characteristic function of Q (equal to 1 for elements of Q and undefined otherwise) is a partial recursive function. Similar results as for the previous case are obtained: 76 neurons are sufficient for universality when using restricted rules and 50 when using extended rules.

In the next section we introduce all necessary prerequisites related to register machines, universality, and standard SN P systems. Section 3 gives the first universal SN P system, and in Section 4 we introduce the extended spiking rules and we produce the universal computing SN P system with 49 rules. In Section 5 we consider the case of universal SN P systems working as generators of sets of numbers.

2 Prerequisites

The reader is assumed to have some elementary knowledge in theoretical computer science as available from the many monographs in the field (e.g., from [12], [14]), so that we specify here only some notations and basic definitions.

For an alphabet V, V^* denotes the set of all strings over V, with the empty string denoted by λ.

A *regular expression* over an alphabet V is defined as follows: (i) λ and each $a \in V$ is a regular expression, (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. Clearly, we assume that the parentheses are not in V; as a matter of fact, we will often omit "unnecessary parentheses". Also, $E_1^+ \cup \lambda$ can be written as E_1^*. With each expression E we associate its language $L(E)$ as follows: (i) $L(\lambda) = \{\lambda\}, L(a) = \{a\}$, for $a \in V$, (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 all regular expressions E_1, E_2.

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

We pass now to introducing the necessary notions related register machines and universality, then the spiking neural P systems.

2.1 Universal Register Machines

We use here register machines given in the form $M = (m, H, l_0, l_h, I)$, where m is the number of registers, H is the set of instruction labels, l_0 is the start label, l_h is the halt label (assigned to instruction `HALT`), and I is the set of instructions; each label from H is associated with only one instruction from I, thus precisely identifying it (therefore, there are as many labels as instructions). The instructions are of the following forms:

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

A register machine M generates a set of numbers in the following way: we start with all registers empty (i.e., storing the number zero), we apply the instruction with label l_0 and we proceed to apply instructions as indicated by the labels (and made possible by the contents of registers); if we reach the halt instruction, then the number n stored at that moment in a specified register r_0 is said to be computed by M; if the computation does not halt, then no number is generated. The set of all numbers generated by M is denoted by $N(M)$. It is known (see, e.g., [7]) that register machines generate all sets of numbers which are Turing computable.

A register machine can also compute any Turing computable function: we introduce the arguments in specified registers $r_1, \ldots, r_k$ (without loss of the generality, we may assume that we use the first k registers), we start with the instruction with label l_0, and if we stop (with the instruction with label l_h), then the value of the function is placed in another specified register, r_t, with all registers different from r_t being empty. The partial function computed in this way is denoted by $M(n_1, n_2, \ldots, n_k)$. In the computing form, the register machines can be considered deterministic, without losing the Turing completeness; then, the ADD instructions $l_i : (\texttt{ADD}(r), l_j, l_k)$ have $l_j = l_k$ (and the instruction is written in the form $l_i : (\texttt{ADD}(r), l_j)$).

In [4], the register machines are used for computing functions, with the universality defined as follows. Let $(\varphi_0, \varphi_1, \ldots)$ be a fixed admissible enumeration of the unary partial recursive functions. A register machine M_u is said to be universal if there is a recursive function g such that for all natural numbers x, y we have $\varphi_x(y) = M_u(g(x), y)$. In [4], several universal register machines are constructed, with the input (the couple of numbers $g(x)$ and y) introduced in registers 1 and 2, and the result obtained in register 0, so that, from now on we also assume that the registers are always numbered from 0 to $m - 1$.

In general, we directly continue here the results from [4], using one of the universal register machines constructed there, implicitly also the recursive function g which encodes the partial recursive function φ_x in the form of the number $g(x)$ as used in the universal register machine.

We give now in the notation introduced above the specific universal register machine from [4] which will be used in Section 3 below: we have $M_u = (8, H, l_0, l_h, I)$, with the instructions (their labels constitute the set H) presented in Figure 1. (The machine from [4] contains a separate check for zero of register 6, of the form "l_8 : if $register(6) = 0$, then go to l_0, else go to l_{10}"; this instruction was replaced in our setup by $l_8 : (\texttt{SUB}(6), l_9, l_0)$, $l_9 : (\texttt{ADD}(6), l_{10})$.) Therefore, there are 8 registers (numbered from 0 to 7) and 23 instructions (hence 23 labels), the last instruction being the halting one. The input numbers (the "code" of the partial

recursive function to simulate and the argument for this function) are introduced in registers 1 and 2, and the result is obtained in register 0.

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

Fig. 1. The universal register machine from [4]

2.2 Spiking Neural P Systems

We introduce now the SN P systems in the form necessary for universality considerations, hence computing functions.

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

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

where:

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

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

   where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained in σ_i;
   b) R_i is a finite set of *rules* of the following two forms:
      (1) $E/a^c \rightarrow a; d$, where E is a regular expression over a and $c \geq 1$, $d \geq 0$;
      (2) $a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from R_i, we have $a^s \notin L(E)$;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn$, $1 \leq i, j \leq m$ (*synapses* between neurons);
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons, respectively.

An SN P system with rules as above is said to be of the *standard* type.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \to a; d \in R_i$ can be applied. This means consuming (removing) c spikes (thus only $k - c$ remain in σ_i; this corresponds to the right derivative operation $L(E)/a^c$), the neuron is fired, and it produces a spike after d time units (as usual in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized). If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, etc. If the rule is used in step t and $d \geq 1$, then in steps $t, t+1, t+2, \ldots, t+d-1$ the neuron is *closed* (this corresponds to the refractory period from neurobiology), so that it cannot receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then that particular spike is lost). In the step $t + d$, the neuron spikes and becomes again open, so that it can receive spikes (which can be used starting with the step $t + d + 1$, when the neuron can again apply rules).

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

If a rule $E/a^c \to a; d$ of type (1) has $E = a^c$, then we write it in the simplified form $a^c \to a; d$.

In each time unit, if a neuron σ_i can use one of its rules, then a rule from R_i *must* be used. Since two firing rules, $E_1/a^{c_1} \to a; d_1$ and $E_2/a^{c_2} \to a; d_2$, 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 in that case, only one of them is chosen non-deterministically. Note however that, by definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

Thus, the rules are used in the sequential manner in each neuron, at most one in each step, but neurons function in parallel with each other. It is important to notice that the applicability of a rule is established based on the *total* number of spikes contained in the neuron.

The initial configuration of the system is described by the numbers $n_1, n_2, \ldots, n_m$, of spikes present in each neuron, with all neurons being open. During the computation, a configuration is described by both the number of spikes present in each neuron and by the state of the neuron, more precisely, by the number of steps to count down until it becomes open (this number is zero if the neuron is already open). Thus, $\langle r_1/t_1, \ldots, r_m/t_m \rangle$ is the configuration where neuron σ_i contains $r_i \geq 0$ spikes and it will be open after $t_i \geq 0$ steps, $i = 1, 2, \ldots, m$; with this notation, the initial configuration is $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$.

A computation in a system as above starts in the initial configuration. In order to compute a function $f : \mathbf{N}^k \longrightarrow \mathbf{N}$, we introduce k natural numbers $n_1, \ldots, n_k$ in the system by "reading" from the environment a binary sequence $z = 10^{n_1-1}10^{n_2-1}1 \ldots 10^{n_k-1}1$. This means that the input neuron of Π receives a spike in each step corresponding to a digit 1 from the string z and no spike

otherwise. Note that we input exactly $k+1$ spikes, i.e., after the last spike we assume that no further spike is coming to the input neuron. The result of the computation is also encoded in the distance between two spikes: we impose the restriction that the system outputs exactly two spikes and halt (immediately after the second spike), hence producing a train spike of the form $0^b10^{r-1}1$, for some $b \geq 0$ and with $r = f(n_1, \ldots, n_k)$ (the system outputs no spike a non-specified number of steps from the beginning of the computation until the first spike).

In Section 5 the SN P systems are used as number generators – in a way which will be specified there.

An SN P system can be represented graphically in a very natural way: the neurons are placed in the nodes of a graph representing the synapses, with the rules and the initial spikes mentioned in each neuron; the input neuron has an incoming arrow and the output neuron has an outgoing arrow, suggesting their communication with the environment. In the next sections we will always present the systems we construct in this way, which is much more transparent (easy to understand) than the symbolic one.

3 A Small Universal SN P System

We pass now to constructing the first of the four announced universal SN P systems, namely the one for computing functions and using rules of the standard type. To this aim, we follow the way used in [3] to simulate a deterministic register machine by an SN P system. This is done as follows: neurons are associated with each register and with each label of an instruction of the machine; if a register contains a number n, then the associated neuron will contain $2n$ spikes; modules as in Figures 2 and 3 are associated with the ADD and the SUB instructions (each of these modules contains two neurons which do not correspond to registers or to labels of instructions, namely the neurons with labels l_i', l_i'').

The work of the system is triggered by introducing two spikes in the neuron σ_{l_0} (associated with the starting instruction of the register machine). In general, the simulation of an ADD or SUB instruction starts by introducing two spikes in the neuron with the instruction label (we say that this neuron is *activated*). We do not describe here in detail the (pretty transparent) way the modules from Figures 2 and 3 work (if necessary, the reader can consult [3] in this respect), but we only remark that after incrementing, respectively decrementing (if possible) the value of register r (the number of spikes in neuron σ_r), we activate exactly one neuron, σ_{l_j} or σ_{l_k}, as requested by the instruction.

Starting with neurons σ_1 and σ_2 already loaded with $g(x)$ and y spikes, respectively, and introducing two spikes in neuron σ_{l_0}, we can compute in our system in the same way as the universal register machine M_u from Figure 1; if the computation halts, then neuron σ_0 will contain the double of $\varphi_x(y)$ spikes.

There are two additional tasks to solve: to introduce the mentioned spikes in the neurons σ_{l_0}, σ_1, σ_2, and to output the computed number. The first task is

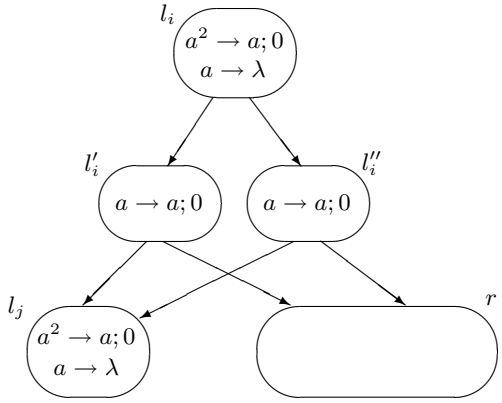

Fig. 2. Module ADD (simulating $l_i : (\texttt{ADD}(r), l_j)$)

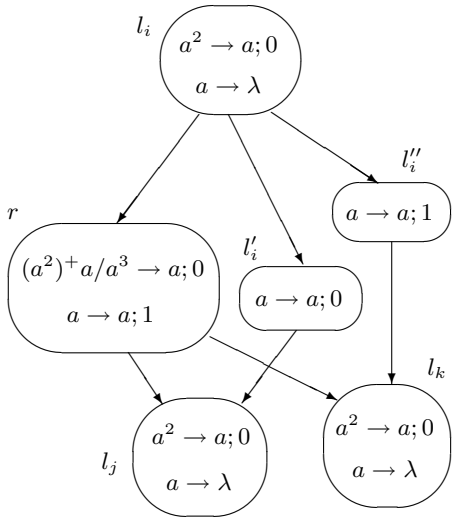

Fig. 3. Module SUB (simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$)

covered by module INPUT presented in Figure 4. The neurons $\sigma_{c_1}, \sigma_{c_2}$ fire only after receiving the third spike, and then they send two spikes to neuron σ_{l_0}, thus starting the simulation of M_u. At that moment, neurons σ_1 and σ_2 are already loaded: neurons σ_{c_3} and σ_{c_4} send to neuron σ_1 as many pairs of spikes as the number of steps between the first two input spikes, and after that they get "over flooded" by the second input spike and are blocked; in turn, neurons $\sigma_{c_5}, \sigma_{c_6}$ start working only after collecting two spikes and stop working after receiving the third spike.

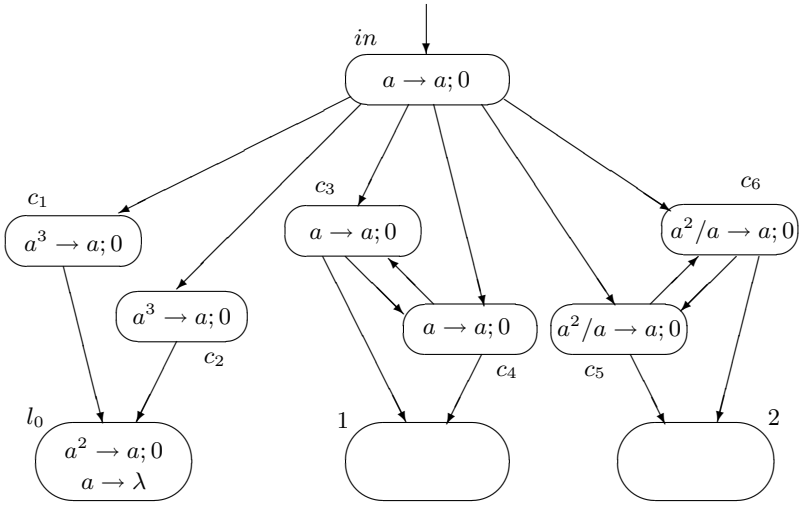

Fig. 4. Module INPUT

The second task needs a modification in the universal register machine: the construction from [3] does not allow subtraction operations on the neuron where we place the result, but register 0 of M_u is subject of such operations. That is why we have to add a further register – we label it with 8 – and we replace the halt instruction of M_u with the following instructions:

$$l_h : (\texttt{SUB}(0), l_{22}, l'_h), \quad l_{22} : (\texttt{ADD}(8), l_h), \quad l'_h : \texttt{HALT}.$$

In this way, we have 9 registers, 24 ADD and SUB instructions, and 25 labels.

We denote by M'_u the obtained register machine.

Having the result of the computation in register 8, which is never decremented during the computation, we can output the result by means of the module OUTPUT from Figure 5. When neuron $\sigma_{l'_h}$ receives two spikes, it fires and sends a spike to neuron σ_8; in this way, the number of spikes of this neuron becomes odd, hence its rule can be applied. A spike arrives in the output neuron, which spikes. From the next step on until exhausting the spikes from neuron σ_8, the output neuron receives two spikes (one from σ_8 and one from σ_{d_1}), hence neuron σ_{out} collects an even number of spikes and cannot fire. When neuron σ_8 has only one spike, it cannot fire anymore, but one further spike comes in neuron σ_{out} from neuron σ_{d_1}, and thus the output neuron spikes for the second (and last) time.

The overall design of the system is given in Figure 6.

Thus, we have

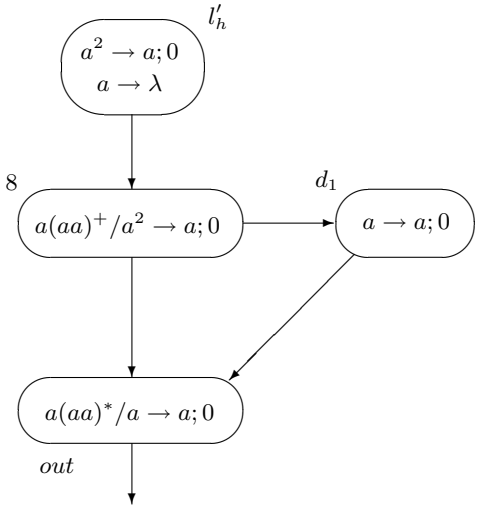

Fig. 5. Module OUTPUT

9 neurons	for the 9 registers,
25 neurons	for the 25 labels,
48 neurons	for the 24 ADD and SUB instructions,
7 neurons	in the INPUT module,
2 neurons	in the OUTPUT module,

which comes to a total of 91 neurons.

This number can be slightly decreased, by some "code optimization", exploiting some particularities of the register machine M'_u.

First, let us observe that the sequence of two consecutive ADD instructions

$$l_{17} : (\texttt{ADD}(2), l_{21}), \quad l_{21} : (\texttt{ADD}(3), l_{18}),$$

without any other instruction addressing the label l_{21}, can be simulated by the module from Figure 7, and in this way we save the neuron associated with l_{21} and instead of four auxiliary neurons used in the two separate ADD instructions we use two neurons.

A similar operation is possible for the following two sequences of ADD – SUB instructions, where again we can save the intermediate labels (l_6 and l_{10}), as well as one auxiliary neuron for each pair:

$$l_5 : (\texttt{ADD}(5), l_6), \quad l_6 : (\texttt{SUB}(7), l_7, l_8),$$
$$l_9 : (\texttt{ADD}(6), l_{10}), \quad l_{10} : (\texttt{SUB}(4), l_0, l_{11}).$$

Instead of the modules ADD and SUB as in Figures 2 and 3, for each couple of instructions as above we can use a module as that from Figure 8, where the labeling of neurons corresponds to the first couple of instructions mentioned above.

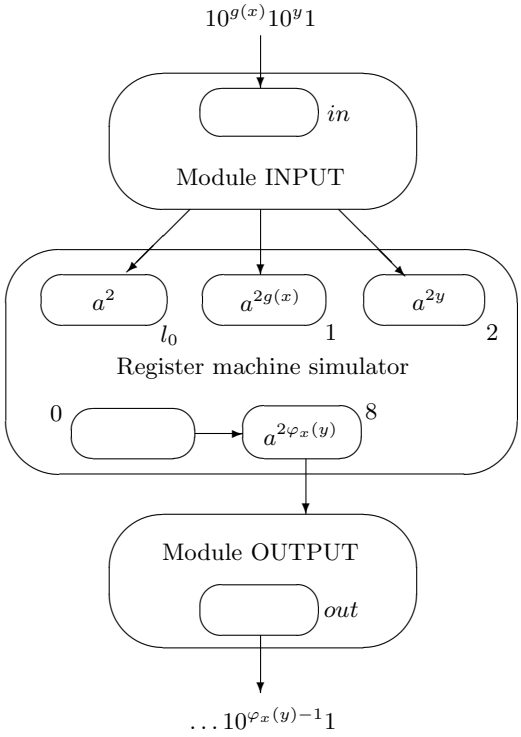

Fig. 6. The general design of the universal SN P system

A similar module is used for the second pair of instructions. In each case, we save two neurons; together with the three neurons saved by the module in Figure 7, we get the improvement from 91 to 84 neurons.

We state this result in the form of a theorem in order to stress its importance:

Theorem 1. *There is a universal computing SN P system with standard rules having 84 neurons.*

In the register machine M'_u there are also couples of SUB – ADD instructions with the label of the second instruction used only in the first instruction, but we were not able to find a coupled module for this case with a smaller number of neurons than in the two SUB and ADD modules given in Figures 2 and 3.

4 Using More General Rules

In several of the modules constructed in the previous section we need pairs of intermediate neurons for duplicating the spike which is transmitted further (such

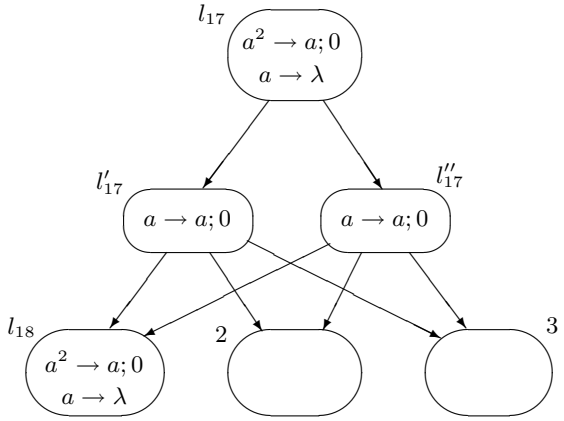

Fig. 7. A module simulating two consecutive ADD instructions

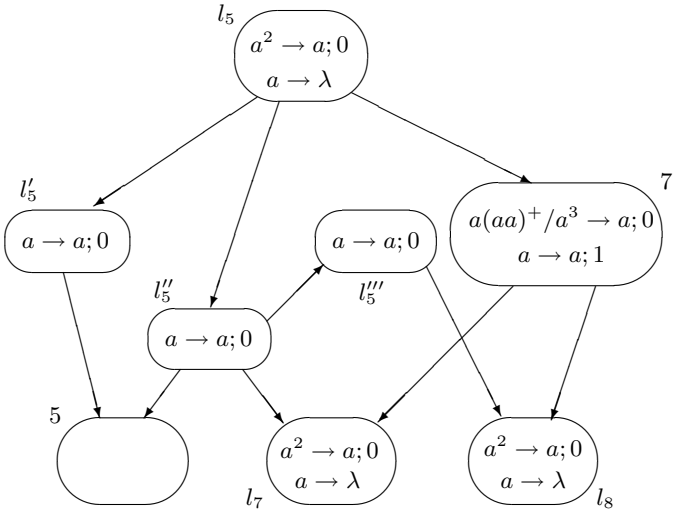

Fig. 8. A module simulating consecutive ADD – SUB instructions

as $\sigma_{l'_i}$ and $\sigma_{l''_i}$ in Figure 2), and this suggests to consider an extension of the rules of SN P systems: to allow spiking rules of the form $E/a^c \to a^p; d$, where all components are as usual, and $p \geq 1$. To be "realistic" (conservative), we impose the restriction $c \geq p$ (the number of produced spikes is not larger than the number of consumed spikes).

The rules of this type are said to be *extended*.

For reasons which will become apparent below, in this section we return to the initial universal register machine M_u, hence having 8 registers (with the result placed in register 0) and 23 instructions.

Using extended rules, the modules for the simulation of ADD and SUB instructions become smaller in the number of neurons, but more complex in the functioning – see Figures 9, 10. Moreover, the delay from firing to spiking is always zero, that is why from now on we do not mention the delay when writing the rules.

This time, the encoding of the value n of a register r is done by means of placing $6n$ spikes in the neuron σ_r associated with the register. That is why, the neuron σ_{l_i} corresponding to an ADD instruction $l_i : (\texttt{ADD}(r), l_j)$ has to produce 6 spikes, which are sent both to neuron σ_r and to the neuron with the label l_j. If l_j is a label which also appears in a SUB instruction, then, as one can see in Figure 10, neuron σ_{l_j} also fires when receiving 10 or 11 spikes, that is why we also have the rules $a^{10} \to a^6$ and $a^{11} \to a^6$ present in neuron σ_{l_i}. Similarly, because the neurons $\sigma_{l_i}, \sigma_{l_j}$ can be involved in SUB instructions, we have considered the forgetting rules $a^7 \to \lambda$, $a^8 \to \lambda$, $a^9 \to \lambda$, and $a^{13} \to \lambda$ in both these neurons in Figure 9. The significance/value of $\delta(l_j)$ will be specified immediately.

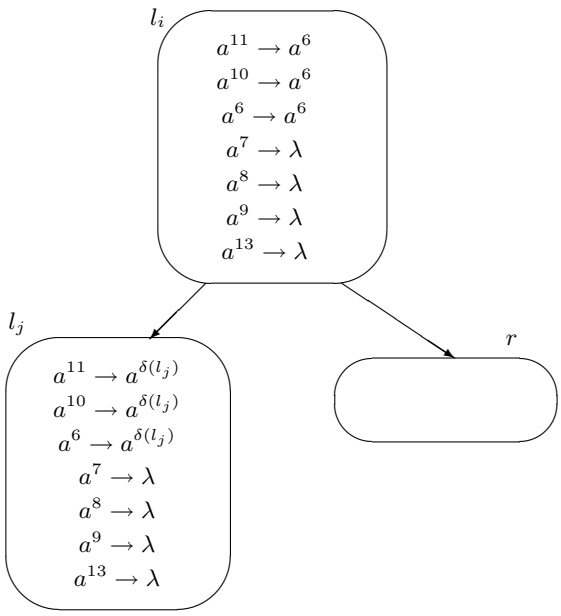

Fig. 9. Module ADD (simulating $l_i : (\texttt{ADD}(r), l_j)$) for extended rules

The simulation of a SUB instruction $l_i : (\texttt{SUB}(r), l_j, l_k)$ proceeds as follows. Assume that at some moment, neuron σ_{l_i} receives either 6 spikes (if l_i is the "output" label of an ADD instruction), or 10 or 11 spikes (if l_i is an "output" label of a SUB instruction), and fires, producing 4 spikes. These spikes are sent at the same time to all neurons $\sigma_r, \sigma_{l'_i}, \sigma_{l_j}$, and σ_{l_k} from Figure 10. There is no rule for handling 4 spikes in the last two neurons, hence these spikes wait here one step.

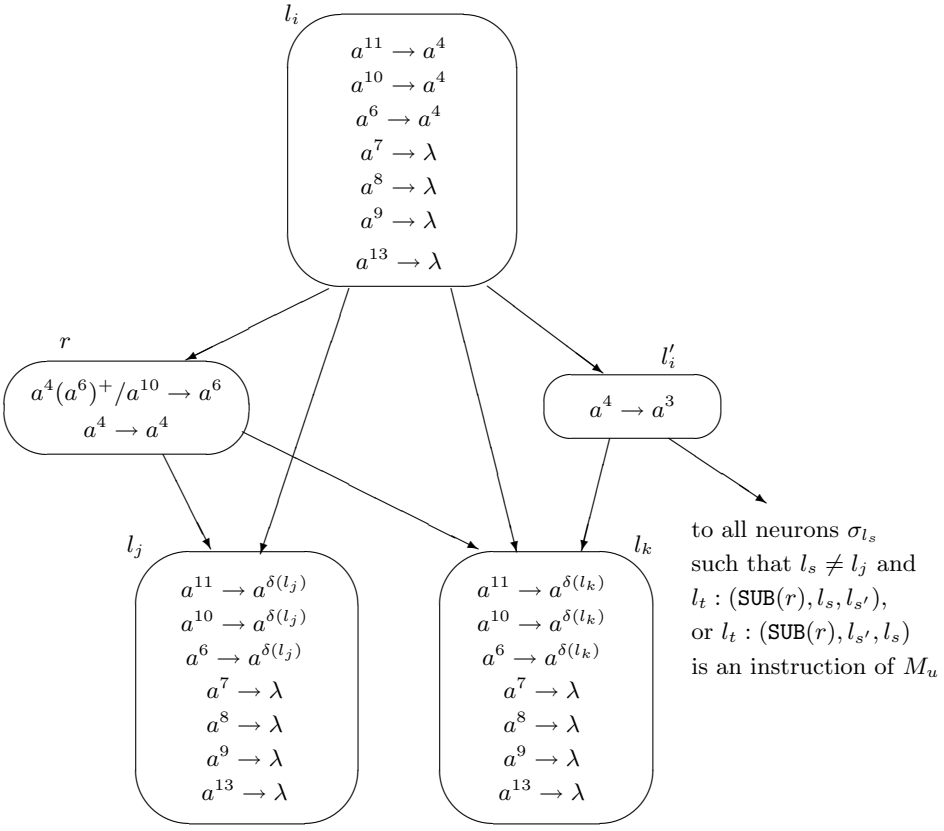

Fig. 10. Module SUB (simulating $l_i : (\texttt{SUB}(r), l_j, l_k)$) for extended rules

In the next step, neuron $\sigma_{l'_i}$ fires and sends three spikes to all neurons σ_{l_s} of the system such that SUB instructions $l_t : (\texttt{SUB}(r); l_s, l_{s'})$ or $l_t : (\texttt{SUB}(r); l_{s'}, l_s)$ exist in the register machine M, with the exception of σ_{l_j} (hence to all neurons associated with output labels of SUB instructions which decrement register r, but not to σ_{l_j}). Simultaneously, σ_r spikes, because its rules are now applicable, the number of spikes it holds is no longer a multiple of 6, but it is of the form $6m+4$ for some

$m \geq 0$. If $m \geq 1$, this means that the register r was non-empty; the subtraction is done by the rule $a^4(a^6)^+/a^{10} \to a^6$. Note that in this way the number of spikes remaining in neuron σ_r returns to a multiple of 6. The six spikes arrive at the same time in all neurons associated with output labels of SUB instructions which decrement register r. In neuron σ_{l_j} we have in this way 10 spikes (four were waiting here from the previous step), and this neuron can fire. In neuron σ_{l_k} we have 13 spikes (6 coming from σ_r, 4 waiting here, and 3 just arrived from $\sigma_{l'_i}$), and they are removed by the forgetting rule $a^{13} \to \lambda$. In all other neurons associated with output labels of SUB instructions which decrement register r we have 9 spikes, and also they are immediately forgotten by the rule $a^9 \to \lambda$ placed in all neurons associated with labels of M_u.

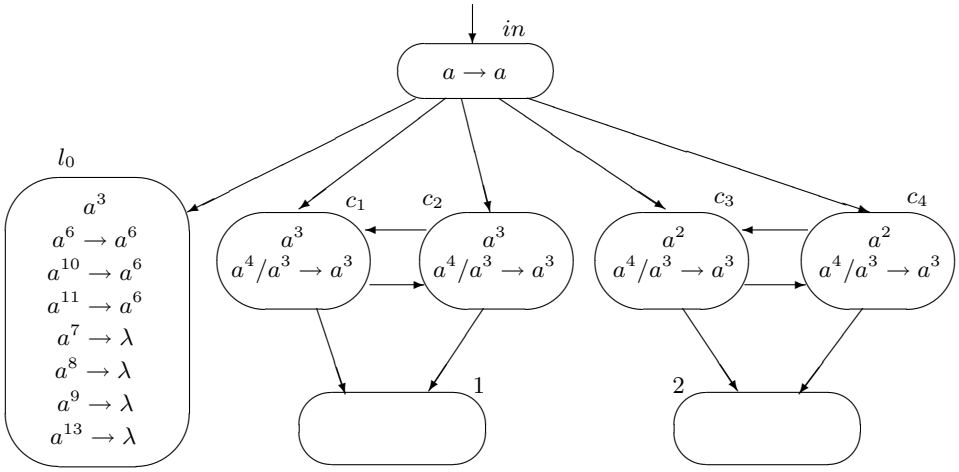

Fig. 11. The INPUT module in the case of extended rules

If register r was empty, this means that after σ_{l_i} fires, we have only four spikes in neuron σ_r, hence the rule $a^4 \to a^4$ is used. This time, neuron σ_{l_j} gets only 8 spikes, and they are removed, but neuron σ_{l_k} gets 11 spikes, and fires. All other neurons associated with output labels of SUB instructions which decrement register r have 7 spikes (4 from σ_r and 3 from $\sigma_{l'_i}$) and these spikes are removed by the rule $a^7 \to \lambda$ present in all these neurons.

In all cases, the simulation of the SUB instruction is correct: we start by activating the neuron σ_{l_i}, we subtract 1 from the contents of register r (i.e., we remove 6 spikes from neuron σ_r) if this is possible, and we continue by activating neuron σ_{l_j}, or, if the register was empty, we just pass to activating neuron σ_{l_k}. No spike remains in any other neuron of the form σ_{l_s} which was involved in this operation.

It is important to note that the neurons σ_{l_i} associated with ADD instructions are different from those associated with SUB instructions: in the first case we have

to produce six spikes, in the latter case four. In Figures 9 and 10 we have indicated this by writing the respective rules in the form $a^{11} \to a^{\delta(l_j)}$, $a^{10} \to a^{\delta(l_j)}$, and $a^6 \to a^{\delta(l_j)}$, where

$$\delta(l) = \begin{cases} 6 & \text{if } l \text{ is the label of an ADD instruction,} \\ 4 & \text{if } l \text{ is the label of a SUB instruction.} \end{cases}$$

The simulation of instructions of M_u is achieved in this way. The computation in our SN P system halts if and only if the computation in M_u halts, which means that the halting instruction is reached.

We have now to see how the INPUT and the OUTPUT modules look like in the case of extended rules. Figures 11 and 12 present these modules. We leave to the reader the task of checking the functioning of the INPUT module (note that l_0 is the label of a SUB instruction of M_u), and we only briefly describe here the work of the OUTPUT module.

We start by activating neuron σ_{l_h} associated with the halt label of M_u. (From Figure 1 we know that l_h is present only in a SUB instruction of M_u as an output label, that is why we do not need a rule $a^6 \to a^3$ in this neuron.) When firing, σ_{l_h} produces 3 spikes, which are sent to neuron σ_0. The number of spikes from σ_0 is of the form $6m + 3, m \geq 0$, hence (i) the rules mentioned in Figure 12 for σ_0 can fire, but (ii) no rule used for SUB instructions (like in Figure 10) can fire (they need $6m + 4$ spikes).

This way to avoid the conflict in using the same neuron in two types of operations is the basis for avoiding the two additional instructions of M'_u in comparison with M_u, as well as the additional register 8.

Now, if register 0 is empty, then we use the rule $a^3 \to a^3$ of R_0 immediately after activating neuron σ_{l_h}; the system spikes only once, hence it outputs the number 0. If register 0 is not empty, then for each unit (for each six spikes from σ_0) we use once the rule $a^3(a^6)^+/a^6 \to a^2$. This rule does not change the arity of the number of spikes, hence it is used until only 3 spikes remain in σ_0. When the first two spikes arrive in σ_{out}, this neuron spikes. All subsequent pairs of spikes sent from σ_0 to σ_{out} are forgotten. The last time when σ_0 spikes, it uses the rule $a^3 \to a^3$, hence again σ_{out} spikes – and then the system halts. In this way, we get the spike train $10^{n-1}1$, encoding the number n as the result of the computation.

From Figures 11 and 12 one sees that, in comparison with Figures 4 and 5, we save two neurons in the first case and one in the second.

The same happens with both intermediate neurons in the ADD + ADD module from Figure 7 (using the ADD + ADD module instead of two ADD modules, a label-neuron is saved): we directly send 6 spikes to each neuron $\sigma_{l_{18}}, \sigma_2, \sigma_3$ from Figure 7.

In this way, our system will contain

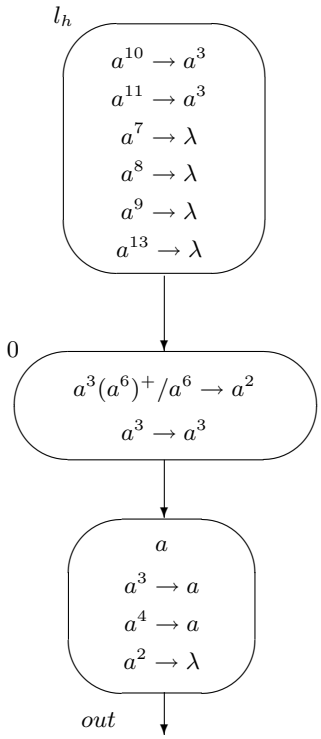

Fig. 12. The OUTPUT module in the case of extended rules.

8 neurons	for the 8 registers,
22 neurons	for the 22 labels (one is saved in the ADD + ADD module),
13 neurons	for the 13 SUB instructions,
5 neurons	in the INPUT module,
1 neuron	in the OUTPUT module,

which means in total 49 neurons.

It is also worth noting that all spiking rules from our system have no delay from firing to spiking.

Theorem 2. *There is a universal computing SN P system with extended rules (without delay) which has 49 neurons.*

5 Universal SN P Systems as Number Generators

Let us now consider the case when with an SN P system Π we associate a set $N(\Pi)$ of numbers as in [3]: the system starts working in its initial configuration;

because of the non-determinism in using the spiking rules, several computations are possible; any halting computation provides a result, in the form of the number of time units between the first two steps when any spike exits the system. Note that this definition covers both the case of restricted rules, with only one spike emitted in a time unit by the output neuron, and the case of extended rules, where it is possible that several spikes are emitted at the same time (they are perceived in the environment of the system as a single spike; however, in order to avoid any complication, in the constructions below we make sure that the output neuron produces always only one spike).

In this framework, an SN P system Π_u is universal if, given a fixed admissible enumeration of the unary partial recursive functions, $(\varphi_0, \varphi_1, \ldots)$, there is a recursive function g such that for each natural number x, if we input the number $g(x)$ in Π_u, by "reading" the sequence $10^{g(x)-1}1$ from the environment, the set of numbers generated by the system is equal to $\{n \in \mathbf{N} \mid \varphi_x(n) \text{ is defined}\}$. Otherwise stated, after introducing the "code" $g(x)$ of the partial recursive function φ_x in a specified neuron, the system generates (hence halts sometimes after sending two spikes out) all numbers n for which $\varphi_x(n)$ is defined.

This time, the strategy followed by the universal system is the following:

1. Read the string $10^{g(x)-1}1$ from the environment and load $2g(x)$ spikes in neuron σ_1.
2. Load neuron σ_2 non-deterministically with an arbitrary natural number n (in the case of using restricted rules this means to introduce $2n$ spikes in neuron σ_2 and in the case of extended rules means to introduce $6n$ spikes); at the same time, output the spike train $10^{n-1}1$ (hence the number n).
3. Check whether the function φ_x is defined for n. To this aim, start the register machine M_u from Figure 1, with $g(x)$ in register 1 and n in register 2. If the computation in M_u halts, then also the computation in our SN P system halts, hence n is introduced in the set of generated numbers.

It is worth noting that there is an essential difference between number generating and function computing: we no longer need to output a result after halting the computation, but we have to randomly generate a number at the beginning of the computation. In this way, we have two important consequences for our construction, a simplification and a supplementary task. First, no separate output module is necessary, the additional register 8 can be omitted, and the simulation of the last SUB instruction used in a halting computation, $l_{18} : (\texttt{SUB}(4), l_0, l_h)$, can be simplified, just ignoring l_h and halting. On the other hand, the input module should be combined with the output one, at the same time non-deterministically producing the number n.

The combined INPUT–OUTPUT module for the case of restricted rules is presented in Figure 13.

After loading neuron σ_1 with $2g(x)$ spikes, we start loading neuron σ_2 with an arbitrary number of spikes, by means of neurons σ_{c_4} and σ_{c_5}, at the same time sending out the respective number as the distance between two spikes emitted by

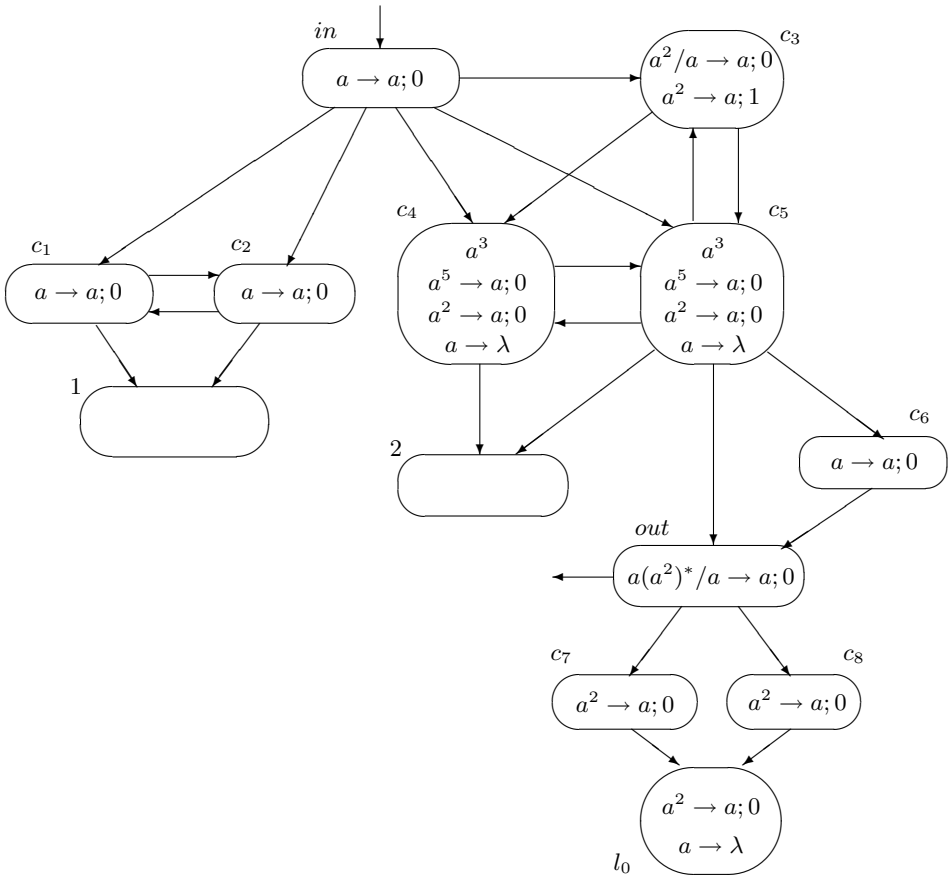

Fig. 13. The INPUT–OUTPUT module for the number generating universal SN P systems

neuron σ_{out}. The work of neurons $\sigma_{c_4}, \sigma_{c_5}$ stops when neuron σ_{c_3} uses the rule $a^2 \to a; 1$, and only after that (after having two spikes emitted by σ_{out}) we load neuron σ_{l_0} with two spikes (via neurons $\sigma_{c_7}, \sigma_{c_8}$) and in this way we start the work of the register machine M_u.

In turn, the simplified module for simulating the instruction $l_{18} : (\texttt{SUB}(4), l_0, l_h)$ is given in Figure 14: when the subtraction is no longer possible, we just halt.

Consequently, the obtained system contains

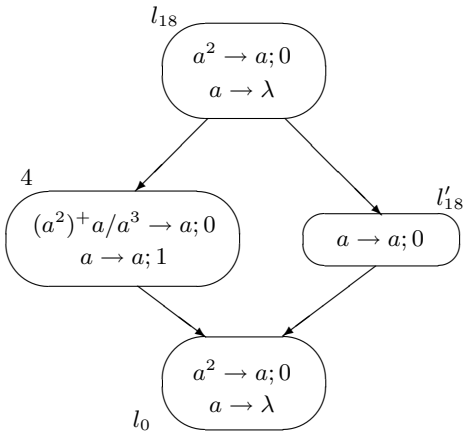

Fig. 14. The simplified module for simulating $l_{18} : (\texttt{SUB}(4), l_0, l_h)$

8 neurons	for the 8 registers,
22 neurons	for the 22 labels (l_h is saved),
42 neurons	for the 21 ADD and SUB instructions,
1 neuron	for the special SUB instruction (Figure 14),
10 neurons	in the INPUT–OUTPUT module,

which means in total 83 neurons. From them, 7 neurons can be saved in the way we have proceeded in the end of Section 3, hence we can conclude with the following result:

Theorem 3. *There is a universal number generating SN P system with standard rules having 76 neurons.*

Passing now to extended rules, we can have the INPUT–OUTPUT module as shown in Figure 15.

This time the system contains 8 neurons for registers, 21 neurons for labels (besides the label saved in the ADD + ADD module, we also save l_h), 13 for SUB instructions, and 8 neurons in the INPUT–OUTPUT module, in total 50 neurons.

Again, the rules use no delay between firing and spiking.

Theorem 4. *There is a universal number generating SN P system with extended rules (without delay) having 50 neurons.*

6 Closing Remarks

It remains as an open problem to decrease the number of neurons from our universal SN P systems. Is this possible in the setups used here? As said in the Introduction, we conjecture that significant improvements are not possible.

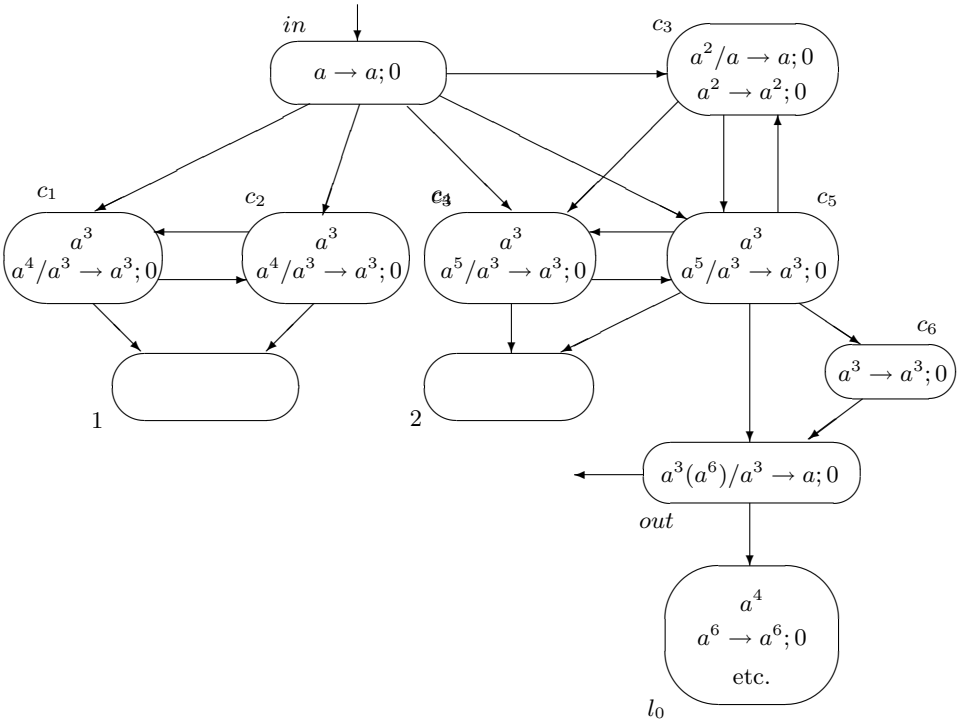

Fig. 15. The INPUT–OUTPUT module in the extended case

Another direction of research is to look for the borderline between universality and non-universality also starting from below: what is the number of neurons for which we can prove that an SN P system is *not* universal? Besides trivial observations (at least three neurons are necessary, one input, one output, and one "working" neuron) we do not have any better estimation at this moment.

Of course, some neurons can be saved if the input to the universal SN P system and its output are not introduced/emitted by means of input or output neurons as above, but simply placed in the necessary neurons. Input–output modules are then saved, but we do not find such a definition "fair", because a "brain" should have interfaces with its environment.

Anyway, the results above look rather optimistic: there exist pretty small universal "brains" (it would be of interest to see where on the evolution scale there are animals with the brain of comparable sizes...).

Note. Useful discussions with Rudi Freund and Takashi Yokomori are gratefully acknowledged.

References

1. E. Csuhaj-Varju, M. Margenstern, G. Vaszil, S. Verlan: Small computationally complete symport/antiport P systems. In volume I of the present proceedings.
2. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosik, S. Woodworth: Normal forms for spiking neural P systems. In the present volume.
3. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
4. I. Korec: Small universal register machines. *Theoretical Computer Science*, 168 (1996), 267–301.
5. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
6. W. Maass, C. Bishop, eds.: *Pulsed Neural Networks*. MIT Press, Cambridge, 1999.
7. M. Minsky: *Computation – Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
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*, to appear (also available at [15]).
10. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted, 2006.
11. Y. Rogozhin: Small universal Turing machines. *Theoretical Computer Science*, 168 (1996), 215–240.
12. J. van Leeuwen, ed.: *Handbook of Theoretical Computer Science*. Elsevier, Amsterdam, 1990.
13. Y. Rogozhin, S. Verlan: On the rule complexity of universal tissue P systems. In *Membrane Computing, International Workshop, WMC6, Vienna, Austria, 2005, Selected and Invited Papers* (R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa, eds.), LNCS 3850, Springer-Verlag, Berlin, 2006, 356–363.
14. D. Wood: *Theory of Computation*. Harper and Row, New York, 1987.
15. The P Systems Web Page: `http://psystems.disco.unimib.it`.