On the Computational Power of Spiking Neural P Systems

Alberto Leporati, Claudio Zandron,
Claudio Ferretti, Giancarlo Mauri

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano – Bicocca
Via Bicocca degli Arcimboldi 8, 20126 Milano, Italy

{leporati,zandron,ferretti,mauri}@disco.unimib.it

Summary. In this paper we study some computational properties of spiking neural P systems. In particular, we show that by using nondeterminism in a slightly extended version of spiking neural P systems it is possible to solve in constant time both the numerical **NP**–complete problem Subset Sum and the strongly **NP**–complete problem 3-SAT. Then, we show how to simulate a universal *deterministic* spiking neural P system with a deterministic Turing machine, in a time which is polynomial with respect to the execution time of the simulated system. Surprisingly, it turns out that the simulation can be performed in polynomial time with respect to the *size of the description* of the simulated system only if the regular expressions used in such a system are of a very restricted type.

1 Introduction

Membrane systems (also called *P systems*) were introduced in [16] as a new class of distributed and parallel computing devices, inspired by the structure and functioning of living cells. The basic model consists of a hierarchical structure composed by several membranes, embedded into a main membrane called the *skin*. Membranes divide the Euclidean space into *regions*, that contain some *objects* (represented by symbols of an alphabet) and *evolution rules*. Using these rules, the objects may evolve and/or move from a region to a neighboring one. Usually, the rules are applied in a nondeterministic and maximally parallel way; moreover, all the objects that may evolve are forced to evolve. A *computation* starts from an initial configuration of the system and terminates when no evolution rule can be applied. The result of a computation is the multiset of objects contained into an *output membrane*, or emitted from the skin of the system. For a systematic introduction to P systems we refer the reader to [18], whereas the latest information can be found in [23].

In an attempt to pass from cell-like to tissue-like architectures, in [13] *tissue P systems* were defined, in which cells are placed in the nodes of a (directed) graph. Since then, this model has been further elaborated, for example, in [4] and [19], with recent results about both theoretical properties [1] and applications [14]. This evolution has led to explore also *neural-like* architectures, yielding to the introduction of *spiking neural P systems* (SN P systems, for short) [8], based on the neurophysiological behavior of neurons sending electrical impulses (*spikes*) along axons to other neurons. We recall that this biological background has already led to several models in the area of neural computation, e.g., see [11, 12, 6].

Similarly to tissue P systems, in SN P systems the cells (neurons) are placed in the nodes of a directed graph, called the *synapse graph*. The contents of each neuron consist of a number of copies of a single object type, called the *spike*. The *firing rules* assigned to a cell allow a neuron to send information to other neurons in the form of electrical impulses (also called spikes) which are accumulated at the target cell. The application of the rules depends on the contents of the neuron; in the general case, applicability is determined by checking the contents of the neuron against a regular set associated with the rule. As inspired from biology, after a cell sends out spikes it becomes "closed" (inactive) for a specified period of time, that reflects the refractory period of biological neurons. During this period, the neuron does not accept new inputs and cannot "fire" (that is, emit spikes). Another important feature of biological neurons is that the length of the axon may cause a time delay before a spike arrives at the target. In SN P systems this delay is modeled by associating a delay parameter to each rule which occurs in the system. If no firing rule can be applied in a neuron, there may be the possibility to apply a *forgetting rule*, that removes from the neuron a predefined number of spikes.

In the original model of SN P systems defined in [8], computations occur as follows. A *configuration* specifies, for each neuron of the system, the number of spikes it contains and the number of computation steps after which the neuron will become "open" (that is, not closed). Starting from an initial configuration, a positive integer number is given in input to a specified *input neuron*. The number is encoded as the interval of time steps elapsed between the insertion of two spikes into the neuron. To pass from a configuration to another one, for each neuron a rule is chosen among the set of applicable rules, and is executed. The computation proceeds in a sequential way into each neuron, and in parallel among different neurons. Generally, a computation may not halt. However, in any case the output of the system is considered to be the time elapsed between the arrival of two spikes in a designated *output cell*. Defined in this way, SN P systems compute functions of the kind $f : \mathbb{N} \to \mathbb{N}$ (they can also indirectly compute functions of the kind $f : \mathbb{N}^k \to \mathbb{N}$ by using a bijection from $\mathbb{N}^k$ to $\mathbb{N}$). By ignoring the output neuron we can define *accepting* SN P systems, in which the natural number given in input is accepted if the computation halts, and rejected otherwise. On the other hand, by ignoring the input neuron (and thus starting from a predefined input configuration) we can define *generative* SN P systems.

In [8] it was shown that generative SN P systems are universal, that is, can generate any recursively enumerable set of natural numbers. Moreover, a characterization of semilinear sets was obtained by spiking neural P systems with a bounded number of spikes in the neurons. These results can also be obtained with even more restricted forms of spiking P systems; for example, [7] shows that at least one of these features can be avoided while keeping universality: time delay (refractory period) greater than 0, forgetting rules, outdegree of the synapse graph greater than 2, and regular expressions of complex form. Finally, in [20] the behavior of spiking neural P systems on infinite strings and the generation of infinite sequences of 0 and 1 was investigated, whereas in [2] spiking neural P systems were studied as language generators (over the binary alphabet $\{0, 1\}$).

The rest of this paper is organized as follows. In section 2 we give some mathematical preliminaries, and we define the standard version of SN P systems (as found in [9]) as well as a slightly extended version. In section 3 we show how the **NP**–complete problems Subset Sum and 3-SAT can be solved in constant time by exploiting nondeterminism in our extended SN P systems. In section 4 we turn our attention to deterministic systems, and we show how to simulate them by using deterministic Turing machines. Section 5 concludes the paper and gives some directions for future research.

2 Preliminaries

Let us start by recalling the standard definition of a spiking neural P system, taken from [9]. A *spiking neural membrane system* (SN P system, for short), of degree $m \geq 1$, is a construct of the form

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \sigma_2, \ldots, \sigma_m$ are *neurons*, of the form $\sigma_i = (n_i, R_i)$, with $1 \leq i \leq m$, 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 $c \geq 1$, $d \geq 0$ are integer numbers; if $E = a^c$, then it is usually written in the following simplified form: $a^c \rightarrow a; d$;
      (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 $a^s \notin L(E)$ (where $L(E)$ denotes the regular language defined by E);
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, is the directed graph of *synapses* between neurons;
4. $in, out \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neurons of Π.

The rules of type (1) are called *firing* (also *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains $k \geq c$ spikes, and $a^k \in L(E)$, then the rule $E/a^c \to a; d \in R_i$ can be applied. The execution of this rule removes c spikes from σ_i (thus leaving $k - c$ spikes), and prepares one spike to be delivered to all the neurons σ_j such that $(i, j) \in syn$. If $d = 0$, then the spike is immediately emitted, otherwhise it is emitted afted d computation steps of the system. (Observe that, as usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized.) 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*, 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), and cannot fire new rules. 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 $t + d + 1$) and select rules to be fired.

Rules of type (2) are called *forgetting* rules, and 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. Note that, by definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

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 $E_2 : a^{c_1} \to a; d_2$, 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. In such a case, only one of them is chosen nondeterministically. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other.

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 each neuron, which can be expressed as the number of steps to count down until it becomes open (this number is zero if the neuron is already open). A *computation* in a system as above starts in the initial configuration. In order to compute a function $f : \mathbb{N}^k \to \mathbb{N}$, we introduce k natural numbers $n_1, n_2, \ldots, n_k$ in the system by "reading" from the environment a binary sequence $z = 0^b 10^{n_1} 10^{n_2} 1 \ldots 10^{n_k} 10^g$, for some $b, g \geq 0$; this means that the input neuron of Π receives a spike in each step corresponding to a digit 1 from the string z. Note that we input exactly $k + 1$ spikes. The result of the computation is also encoded in the distance between two spikes: we impose to the system to output exactly two spikes and halt (sometimes after the second spike) hence producing a train spike of the form $0^{b'} 10^r 10^{g'}$, for some $b', g' \geq 0$ and with $r = f(n_1, n_2, \ldots, n_k)$.

If we use an SN P system in the *generative* mode, then no input neuron is considered, hence no input is taken from the environment; we start from the initial configuration, and the distance between the first two spikes of the output neuron (or other numbers, see the discussion in [9]) is the result of the computation.

Dually, we can ignore the output neuron, and if the computation halts, then the number is accepted.

We define the *description size* of an SN P system Π as the number of bits which are necessary to describe it. Since the alphabet O is fixed, no bits are necessary to define it. In order to represent *syn* we need at most m^2 bits, whereas we can represent the values of *in* and *out* by using $\log m$ bits each. Every neuron σ_i requires to specify a natural number n_i, and a set R_i of rules. Each rule requires to specify its type (firing or forgetting), which can be done with 1 bit, and in the worst case it requires to specify a regular expression and two natural numbers. If we denote by N the maximum natural number that appears in the definition of Π, R the maximum number of rules which occur in its neurons, and S the maximum size required by the regular expressions that occur in Π (more on this later), then we need a maximum of $\log N + R(1 + S + 2\log N)$ bits to describe every neuron of Π. Hence, to describe Π we need a total of $m^2 + 2\log m + m\big(\log N + R(1+S+2\log N)\big)$ bits. Note that this quantity is polynomial with respect to m, R, S and $\log N$. Since the regular languages determined by the regular expressions that occur in the system are *unary* languages, the strings of such languages can be bijectively identified by their lengths. Hence, when writing the regular expression E, instead of writing unions, concatenations and Kleene closures among strings we can do the same by using the lengths of such strings. In this way we obtain a representation of E which is exponentially more compact than the usual representation of regular expressions. As we will see in section 4, this compact representation will yield some difficulties when we will simulate a deterministic accepting SN P system by a deterministic Turing machine.

In what follows it will be convenient to consider also a slightly extended version of SN P systems. Precisely, we will allow rules of the type $E/a^c \to a^p; d$, where $c \geq 1$, $p \geq 0$ and $d \geq 0$ are integer numbers. The semantics of this kind of rules is as follows: if the contents of the neuron matches the regular expression E, then the rule can be applied. When the rule is applied, c spikes are removed from the contents of the neuron and p spikes are prepared to be delivered to all the neurons which are directly connected (through an arc of *syn*) with the current neuron. If $d = 0$, then these p spikes are immediately sent, otherwise the neuron becomes closed for the next d computation steps, after which the p spikes will be sent. As before, a closed neuron does not receive spikes from other neurons, and does not apply any rule. If $p = 0$, then we obtain a forgetting rule as a particular case of our general rules.

Also in the extended SN P systems it may happen that, given two rules $E_1/a^{c_1} \to a^{p_1}; d_1$ and $E_2/a^{c_2} \to a^{p_2}; d_2$, if $L(E_1) \cap L(E_2) \neq \emptyset$ then for some contents of the neuron both the rules can be applied. In such a case, we nondeterministically choose one of them. Note that we do not require that forgetting rules are applied only when no firing rule can be applied. We say that the system is *deterministic* if, for every neuron that occurs in the system, any two rules $E_1/a^{c_1} \to a^{p_1}; d_1$ and $E_2/a^{c_2} \to a^{p_2}; d_2$ in the neuron are such that $L(E_1) \cap L(E_2) = \emptyset$. This means

that, for any possible contents of the neuron, at most one of the rules that occur in the neuron may be applied.

By using an *input neuron* and an *output neuron*, we have SN P systems that compute functions of the kind $f : \mathbb{N} \to \mathbb{N}$, and hence we cover both the generative and the accepting cases. If $out = 0$, then it is understood that the output is sent to the environment (as the number of spikes produced by the system, as the distance between the first two spikes, etc.). As usual, to use an SN P system in the generative mode we do not consider the input neuron, and thus no input is taken from the environment; we start from the initial configuration, and the distance between the first two spikes of the output neuron (or the number of spikes contained into the output neuron at the end of the computation, as discussed above) is the result of the computation. Note that generative SN P systems are inherently nondeterministic, otherwise they would always reproduce the same sequence of computation steps, and hence the same output. Dually, we can ignore the output neuron to obtain an accepting SN P system. We input a number in the system as the distance between two spikes entering the input neuron (or the number of spikes that occur in the input neuron in the initial configuration) and, if the computation halts, then the number is accepted.

The *description size* of an extended SN P system is defined exactly as we did for standard systems, the only difference being that now we require (at most) three natural numbers to describe a rule.

3 Solving NP–complete Problems with Extended Spiking Neural P Systems

In this section we show that *nondeterministic* SN P systems are very powerful computing devices, at least in the extended version defined in the previous section: in fact, they are able to solve **NP**–complete problems in a *constant* number of computation steps.

3.1 Solving the Subset Sum problem

Let us first consider the Subset Sum problem, which can be stated as follows.

Problem 1. Name: Subset Sum.

- Instance: a (multi)set $V = \{v_1, v_2, \ldots, v_n\}$ of positive integer numbers, and a positive integer number S
- Question: is there a subset $B \subseteq V$ such that $\sum_{b \in B} b = S$?

If we allow to nondeterministically choose among the rules which occur in the neurons, then the extended SN P system depicted in Figure 1 solves any given instance of Subset Sum in a constant number of steps. We emphasize the fact

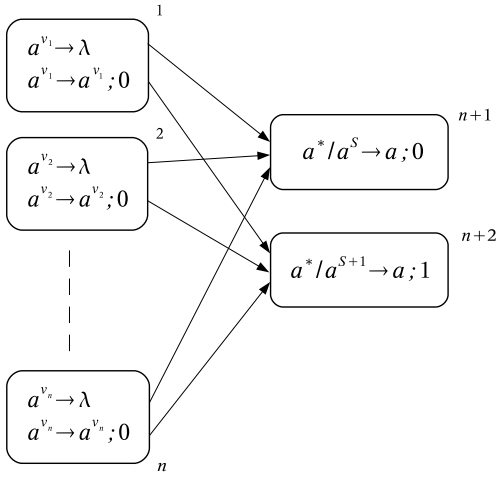

Fig. 1. A nondeterministic extended SN P system that solves the SUBSET SUM problem in constant time

that such a solution occurs in the *semi-uniform* setting, that is, for every instance of SUBSET SUM we build an SN P system that specifically solves that instance.

Let $(V = \{v_1, v_2, \ldots, v_n\}, S)$ be the instance of SUBSET SUM to be solved, and let $B \subseteq V$. In the initial configuration of the system, the leftmost neurons contain (from top to bottom) $v_1, v_2, \ldots, v_n$ spikes, respectively, whereas the rightmost neurons contain zero spikes each. In the first step of computation, in each of the leftmost neurons of the SN P system depicted in Figure 1 it is nondeterministically chosen whether to include or not the element v_i in B; this is accomplished by nondeterministically choosing among one rule that forgets v_i spikes (in such a case, $v_i \notin B$) and one rule that propagates v_i spikes to the rightmost neurons. At the beginning of the second step of computation a certain number N of spikes, that corresponds to the sum of the v_i which have been chosen, occurs in the rightmost neurons. We have three possible cases:

- $N < S$: in this case neither the rule $a^S \to a; 0$ nor the rule $a^{S+1} \to a; 1$ (which occur in the neuron at the top and at the bottom of the second layer, respectively) fire, and thus no spike is emitted to the environment;
- $N = S$: only the rule $a^S \to a; 0$ fires, and emits a single spike to the environmnent. No further spikes are emitted;
- $N > S$: both the rules $a^S \to a; 0$ and $a^{S+1} \to a; 1$ fire. The first rule immediately sends one spike to the environment, whereas the second rule sends another spike at the next computation step (due to the delay associated with the rule).

Hence, by counting the number of spikes emitted to the environment at the second and third computation steps we are able to read the solution of the given instance of Subset Sum: the instance is positive if and only if a single spike is emitted.

The formal definition of the extended (generating) SN P system depicted in Figure 1 is as follows:

$$\Pi = (\{a\}, \sigma_1, \ldots, \sigma_{n+2}, syn, out)\,,$$

where:

- $\sigma_i = (v_i, \{a^{v_i} \to \lambda,\ a^{v_i} \to a^{v_i}; 0\})$ for all $i \in \{1, 2, \ldots, n\}$;
- $\sigma_{n+1} = (0, \{a^S \to a; 0\})$;
- $\sigma_{n+2} = (0, \{a^{S+1} \to a; 1)$;
- $syn = \bigcup_{i=1}^{n} \{(i, n+1), (i, n+2)\}$;
- $out = 0$ indicates that the output is sent to the environment.

However, here we are faced with a problem that we have already encountered in [10], and that we will encounter again in the rest of the paper. In order to clearly expose the problem, let us consider the following algorithm that solves Subset Sum using the well-known Dynamic Programming technique [3]. In particular, the algorithm returns 1 on positive instances, and 0 on negative instances.

```
Subset Sum({v_1, v_2, ..., v_n}, S)
for j ← 0 to S
    do M[1, j] ← 0
M[1, 0] ← M[1, v_1] ← 1
for i ← 2 to n
     do for j ← 0 to S
              do M[i, j] ← M[i − 1, j]
                 if j ≥ v_i and M[i − 1, j − v_i] > M[i, j]
                    then M[i, j] ← M[i − 1, j − v_i]
return M[n, S]
```

In order to look for a subset $B \subseteq V$ such that $\sum_{b \in B} b = S$, the algorithm uses an $n \times (S+1)$ matrix M whose entries are from $\{0, 1\}$. It fills the matrix by rows, starting from the first row. Each row is filled from left to right. The entry $M[i, j]$ is filled with 1 if and only if there exists a subset of $\{v_1, v_2, \ldots, v_i\}$ whose elements sum up to j. The given instance of Subset Sum is thus a positive instance if and only if $M[n, S] = 1$ at the end of the execution.

Since each entry is considered exactly once to determine its value, the time complexity of the algorithm is proportional to $n(S+1) = \Theta(nS)$. This means that the difficulty of the problem depends on the value of S, as well as on the magnitude of the values in V. In fact, let $K = max\{v_1, v_2, \ldots, v_n, S\}$. If K is polynomially bounded with respect to n, then the above algorithm works in polynomial time. On the other hand, if K is exponential with respect to n, say $K = 2^n$, then the above algorithm works in exponential time and space. This behavior is usually

referred to in the literature by telling that SUBSET SUM is a *pseudo–polynomial* **NP**–complete problem.

The fact that in general the running time of the above algorithm is not polynomial can be immediately understood by comparing its time complexity with the instance size. The usual size for the instances of SUBSET SUM is $\Theta(n \log K)$, since for conciseness every "reasonable" encoding is assumed to represent each element of V (as well as S) using a string whose length is $O(\log K)$. Here all logarithms are taken with base 2. Stated differently, the size of the instance is usually considered to be the number of bits which must be used to represent in binary S and all the integer numbers which occur in V. If we would represent such numbers using the unary notation, then the size of the instance would be $\Theta(nK)$. But in this case we could write a program which first converts the instance in binary form and then uses the above algorithm to solve the problem in polynomial time with respect to the new instance size. We can thus conclude that the difficulty of a numerical **NP**–complete problem depends also on the measure of the instance size we adopt.

The problem we mentioned above about the SN P system depicted in Figure 1 is that the rules $a^{v_i} \to \lambda$ and $a^{v_i} \to a^{v_i}; 0$ which occur in the leftmost neurons, as well as those that occur in the rightmost neurons, check for the existence of a number of spikes which is exponential with respect to the usually agreed instance size of SUBSET SUM. Moreover, to initialize the system the user has to place a number of objects which is also exponential. This is not fair, because it means that the SN P system that solves the **NP**–complete problem has an exponential size with respect to the binary string which is used to describe it; an exponential effort is thus needed to build the system, that easily solves the problem by working in unary notation (hence in polynomial time with respect to the size of the system, but not with respect to its *description size*). This problem is in some aspects similar to what has been described in [10], concerning traditional P systems that solve **NP**–complete problems.

3.2 The 3-SAT problem

In this section we show that SN P systems are also able to solve non-numerical **NP**–complete problems. Such problems are inherently *strongly* **NP**–complete, that is, they remain **NP**–complete even if the numbers eventually contained into the instance are expressed in unary form. Specifically, we first propose a simple extended SN P system that solves 3-SAT, and then we show that also standard SN P systems are able to solve this problem.

We start by recalling some well known definitions, in order to settle the notation. A *boolean* variable is a variable which can assume one of two possible truth values: TRUE and FALSE. As usually done in the literature, we will denote TRUE by 1 and FALSE by 0. A *literal* is either a directed or a negated boolean variable. A *clause* is a disjunction of literals, whereas a 3-*clause* is a disjunction of exactly three literals. Given a set $X = \{x_1, x_2, \ldots, x_n\}$ of boolean variables, an *assignment* is a mapping $a : X \to \{0, 1\}$ that associates to each variable a truth value. The

number of all possible assignments to the variables of X is 2^n. We say that an assignment *satisfies* the clause C if, assigned the truth values to all the variables which occur in C, the evaluation of C (considered as a boolean formula) gives 1 as a result.

The 3-SAT decision problem is defined as follows.

Problem 2. NAME: 3-SAT.

- INSTANCE: a set $C = \{C_1, C_2, \ldots, C_m\}$ of 3-clauses, built on a finite set $\{x_1, x_2, \ldots, x_n\}$ of boolean variables;
- QUESTION: is there an assignment of the variables $x_1, x_2, \ldots, x_n$ that satisfies all the clauses in C?

It is well known [5] that 3-SAT is an **NP**–complete problem. In what follows we will equivalently say that an instance of 3-SAT is a boolean formula ϕ_n, built on n free variables and expressed in conjunctive normal form, with each clause containing exactly three literals. The formula ϕ_n is thus the conjunction of the above clauses. Notice that the number m of possible 3-clauses is polynomially bounded with respect to n: in fact, since each clause contains exactly three literals, we can have at most $(2n)^3 = 8n^3$ clauses.

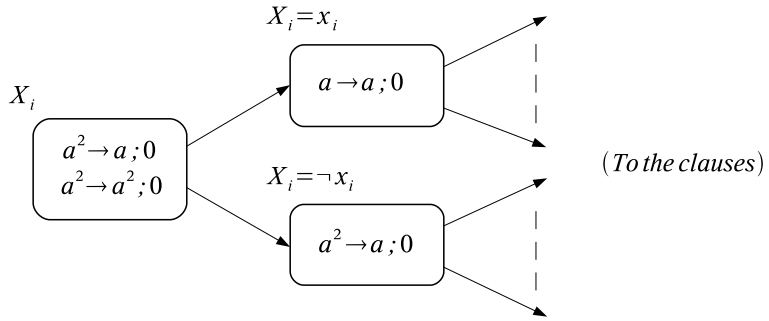

Fig. 2. A nondeterministic extended SN P system that solves the 3-SAT problem in constant time: the module used to build the first two layers (one for each boolean variable)

The extended SN P system that solves 3-SAT is depicted in Figures 2 (the module used to build the first and the second layer) and 3 (third and fourth layers). It is composed by four layers of neurons, the first three of which correspond to the variables, the literals and the clauses of ϕ_n, respectively; the fourth layer is composed by a single neuron, that gathers the spikes produced by the neurons of the third layer. Every neuron in the first layer is connected with its corresponding one or two neurons in the second layer, depending upon whether the literal appears only directed/negated in ϕ_n, or both. Similarly, the neurons of the second layer are connected with those of the third layer according to what literals appear in each clause. Since we are dealing with 3-SAT, every neuron in the third layer will

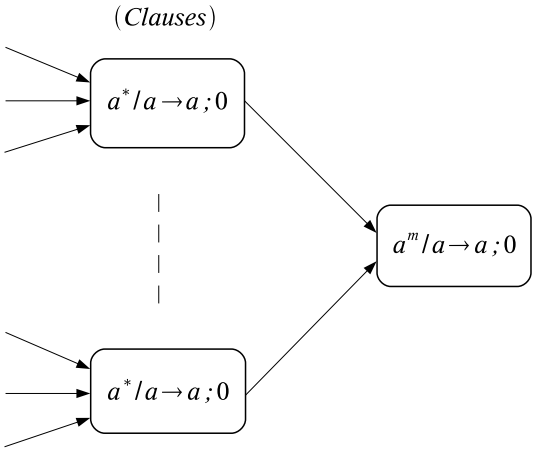

Fig. 3. A nondeterministic extended SN P system that solves the 3-SAT problem in constant time: third and fourth layer (clauses and output neuron)

have exactly three input lines coming from the second layer. Finally, every neuron of the third layer is connected with the output neuron.

During the computation, spikes move from the first to the fourth layer, and then (eventually) are expelled to the environment. In the initial configuration, every neuron in the first layer (which is bijectively associated with one of the n variables of ϕ_n) contains two spikes, whereas all the other neurons are empty. In the first step of the computation, in each neuron of the first layer it is nondeterministically chosen whether to assign 1 or 0 to the corresponding variable, that is, whether to assign 1 to the directed or to the negated literal. This choice is made by nondeterministically choosing between two rules: one that sends two spikes to the next layer, and one that sends a single spike. In the former case, only the neuron that corresponds to the negated literal will fire during the next computation step; in the latter case, only the neuron that corresponds to the directed literal will fire. Since literals are directly connected to the clauses in which they appear, every neuron associated to a clause will receive one spike if and only if at least one of its literals is satisfied. In such a case, one spike is sent to the neuron of the fourth layer, that in this way will contain as many spikes as the number of satisfied clauses. The rule contained in this neuron will fire if and only if there are m spikes, that is, if and only if all the clauses are verified. We can thus conclude that the instance ϕ_n of 3-SAT is positive if and only if (at least) one spike is emitted to the environment.

As mentioned above, it is not necessary to use *extended* SN P systems to solve 3-SAT. In Figure 4 we can see a module which contains only standard rules, whose behavior is almost equivalent to that of the extended module depicted in Figure 2. Neuron number 1 initially contains one spike, which is delivered to all the neurons of the second layer during the first step of computation; note that this neuron is

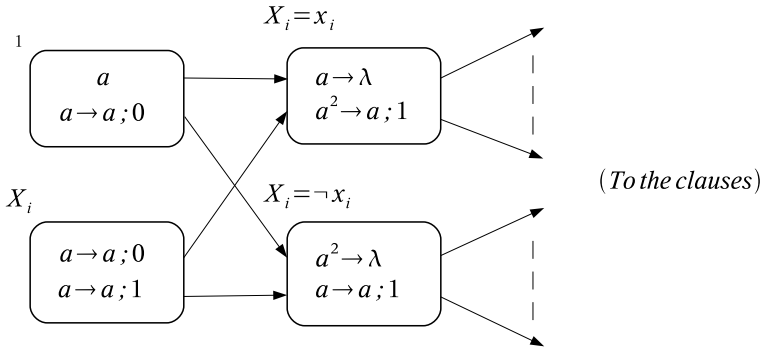

Fig. 4. A nondeterministic standard SN P system that solves the 3-SAT problem in constant time: the module used to build the first two layers (one for each boolean variable). Neuron 1 occurs only one time in the system, and is connected with every neuron of the second layer

not repeated for all modules, but appears only once in the system. All the other neurons of the system are connected exactly in the same way as the corresponding neurons of the extended SN P system described above.

During the first step of computation, every neuron labelled with X_i (in the first layer) nondeterministically chooses whether to immediately emit one spike, or to emit it after a delay of one time step. In the former case, at the end of the first computation step both the neurons of the second layer connected with X_i (corresponding to the directed and to the negated variable, respectively) will contain two spikes. However, these spikes will be removed in the second neuron by applying the rule $a^2 \to \lambda$, whereas in the first neuron they will produce a single spike that will propagate to the third layer after a delay of one time step. Hence, the nondeterministic choice of immediately emitting a spike from the first to the second layer corresponds to assigning 1 to variable x_i. On the other hand, if the spike is emitted from neuron X_i only after a delay of one time step, then at the end of the first computation step both the neurons corresponding to the directed and to the negated variable will contain one spike. This time, the application of the rule $a \to \lambda$ will remove such a spike from the first neuron, whereas the second neuron will emit one spike towards the third layer. Hence, the choice to emit a spike from the first to the second layer after one time step corresponds to assigning 0 to the variable x_i.

Note that the spikes are emitted from the neurons of the second layer with a delay of one computation step; this is done in order to keep such neurons closed during the second computation step, so that the spike (eventually) emitted from the first layer will get lost. However, if desired, we can avoid using delays in the second layer by looking at the answer produced by the system during the fourth computation step, and ignoring what happens afterwards.

4 Simulating Deterministic SN P Systems with Deterministic Turing Machines

Now that we have seen how powerful can be nondeterministic SN P systems, let us turn to *deterministic* SN P systems. In this section we consider a slight extension of the universal SN P system Π defined in [2], and we show that any t steps of computation of Π can be simulated by a deterministic Turing machine in a time which is polynomial both with respect to t and with respect to the *description size* of Π. With respect to the universal SN P system defined in [2], our extension allows to send a predefined number q of spikes to adjacent neurons, instead of sending just one spike. Clearly this modification does not affect universality, and thus also our extended SN P systems are universal.

As we will see, it is possible to simulate these systems in polynomial time mainly because the regular expressions used in the simulated SN P system are of a very restricted form. On the other hand, we will show that if the use of unrestricted regular expressions is allowed, then it is possible to solve the SUBSET SUM **NP**–complete problem by exploting the implicit mechanism used by SN P systems to decide whether a rule can be applied or not.

Theorem 1. *Consider a deterministic accepting SN P system Π, of degree $m \geq 1$, in which:*

- *P and Q are the maximum numbers of spikes that appear in the left and in the right side of the rules, respectively;*
- *D is the maximum delay that appears in the rules;*
- *all the regular expressions are of the following forms: a^i, with $i \leq 3$, or $a(aa)^+$.*

Then, t steps of computation of Π can be simulated by a deterministic Turing machine in a polynomial time with respect to t and to the description size of Π.

Proof. Consider a deterministic accepting SN P system

$$\Pi = (\{a\}, \sigma_1, \ldots, \sigma_m, syn, in)$$

having the characteristics mentioned in the statement of the theorem. We build a deterministic Turing machine M with multiple tapes, such that t steps of computation of Π can be simulated by M in a number of steps which is polynomial with respect to t and to the description size of Π.

To simulate Π with M, we basically need to keep track of the state (number of spikes and number of steps after which the neuron will become open) of each neuron. In general, the simulation of a single open neuron proceeds by first looking for the (unique) rule that can fire, among all the rules of the neuron. Once such rule has been found, we remove all the spikes consumed by the rule and we communicate (after a specified number of steps) the produced spikes to all adjacent neurons, provided that they are open.

In order to formalize this simulation we consider a deterministic Turing machine M with $m + 1$ tapes: m tapes are used to simulate the activity of each neuron,

while the remaining tape is used to store the description of Π. In the i-th tape, used to simulate neuron σ_i, we write the triplet $N_i = (n_i, o_i, t_i)$, where:

- n_i indicates the number of spikes contained into neuron σ_i;
- o_i stores the number of spikes produced by the last rule that fired, and that are ready to be sent to adjacent neurons (zero if the neuron is open, but no rule can be applied);
- t_i denotes the number of steps during which the neuron will remain closed (zero if the neuron is open).

With a little abuse of notation, in what follows we will refer to neuron σ_i by the triplet N_i (that contains the dynamical information about the state of σ_i) and by its set of rules R_i.

We simulate each step of computation of Π in two macro substeps: in the first substep we simulate the *firing phase* (i.e., the consumption of spikes) of each open neuron; then, in the second substep we simulate the *spiking phase* (i.e., the receipt of spikes) from adjacent neurons.

The simulation of a step of Π can be illustrated using the following algorithm, given in pseudocode:

```
SIMULATE-COMPUTATION-STEP(N, R)
// Remark: N = (N_1, N_2, ..., N_m), where N_i = (n_i, o_i, t_i)   ∀ i ∈ {1, ..., m}
//         R = (R_1, R_2, ..., R_m), where R_i is the set or rules of neuron σ_i

// Firing phase
for each neuron N_i
    do if t_i = 0                    // if the neuron is open then fire
       then (p; q; t) = SELECT-RULE-TO-APPLY(N_i, R_i)
            n_i ← n_i − p            // remove the spikes used to fire
            o_i ← q                  // prepare the buffer for emitting spikes
            t_i ← d                  // set closing time
       else t_i ← t_i − 1            // else decrease closing time

// Spiking phase
for each neuron N_i
    do if t_i = 0                    // if the neuron is open then spike
       then for each neuron N_j
            if (i, j) ∈ syn and t_j = 0      // if σ_j is connected to
                                             //    σ_i and is open
               then n_j ← n_j + o_i          // send the spikes
                    o_i ← 0                  // once spiked, clear the buffer


SELECT-RULE-TO-APPLY(N_i, R_i)
// Remark: N_i = (n_i, o_i, t_i)
//         R_i = {r_{i,1}, r_{i,2}, ..., r_{i,k}}, where r_{i,h} = E_{i,h}/a^{p_{i,h}} → a^{q_{i,h}}; d_{i,h}
```

```
//                                          for all h ∈ {1, 2, ..., k}
// Select the rule to be applied in neuron N_i
for each rule r_{i,h} ∈ R_i
    if n_i ∈ L(E_{i,h}) and n_i ≥ p_{i,h}        // if the rule is applicable
       then return (p_{i,h}; q_{i,h}; d_{i,h})      // set the parameters to be used
                                                    // in the simulation
return (0; 0; 0)          // if no rule can be applied, set to 0 all parameters
```

The time complexity of this algorithm can be determined as follows.

The first (firing) phase requires to check for each neuron if it is open and, in such a case, to select a rule and simulate it. Let Z_i denote the time required to select the rule to be applied in neuron σ_i, and let Z be the maximum of all Z_i. We will return later to this value, due to its importance in the time complexity of the overall simulation. Once the rule to be applied has been selected, we need to update the number of spikes in the neuron. Thus, we first subtract a certain number p of spikes (at most P) from n_i. The time required by this operation depends on the number of binary digits which are needed to represent p and n_i. In general, the largest among the two numbers will be the latter. In fact assume that, at the first step of computation, n_i is initialized with a certain number w of spikes; moreover, assume that the neuron does not consume any spike until the current step, and that all the other neurons send to σ_i the maximum possible amount of spikes (P) per computation step in all the previous steps. Then, after t steps of computation, the neuron σ_i will contain $w + tQm$ spikes. This means that the above subtraction requires a time which is $O(\log(w + tQm))$, where w, m and Q are polynomial with respect to the description size. The time required to prepare the buffer for the emission of spikes is $O(\log Q)$, while the time required to set the closure time is $O(\log D)$. Thus, the time required to simulate the firing of a single neuron is $O(Z + \log(w + tQm) + \log Q + \log D)$, and for m neurons is $O(m\,(Z + \log(w + tQm) + \log Q + \log D))$.

The second phase, the spiking, requires to check for each neuron σ_i if it is open and, in such a case, to check for any other neuron σ_j if it is connected to σ_i. If it is connected, then we need to add to n_j the number of spikes emitted by σ_i. In the worst case, each neuron is connected to all other neurons, and all neurons deliver Q spikes to every other neuron. This means that for each neuron σ_i (that is, m times) we have to check whether it is open; this operation requires $O(\log D)$ steps. If σ_i is open, then for every other neuron σ_j we have to:

- check if neurons σ_i and σ_j are connected: the time needed to execute this operation is proportional to m^2 (here we assume that the Turing machine is able to move only to the next or to the previous cell during a computation step);
- check if neuron σ_j is open: the time needed is $O(\log D)$;
- if both the previous conditions hold, then add the spikes emitted by σ_i to n_j. This sum involves numbers of at most $O(\log(w + tQm))$ bits.

Thus, a step of the spiking phase requires $O\big((m \log D) \cdot m \cdot (m^2 + \log D + \log(w + tQm))\big) = O\big((m^2 \log D)(m^2 + \log D + \log(w + tQm))\big)$ steps.

The total time required to simulate t steps of Π is thus t times the time needed to perform the two phases, that is, $t \cdot \big(O(m\,(Z + \log(w + tQm) + \log Q + \log D)) + O\big((m^2 \log D)(m^2 + \log D + \log(w + tQm))\big)\big)$.

To show that this time is polynomial with respect to the description size of the system Π, we need to explicit the time Z required to select which rule has to be applied in neuron σ_i. We stress the fact that, since the system Π is deterministic, at each computation step there is at most one rule in σ_i which can fire. In order to select such a rule, we need to check whether there are enough spikes in the neuron (and clearly this can be done in polynomial time), as well as to check if $n_i \in L(E_i)$. In general, this last operation cannot be done in polynomial time, as it will be proved in the next proposition. Nonetheless, in [9] it is shown that universality can be obtained by using SN P systems where the regular expressions associated with each rule are of very simple forms: a^i, with $i \leq 3$, or $a(aa)^+$. Considering such systems, it is easy to see that the time required to check if the contents of σ_i is in the regular set defined by the regular expression can be done in polynomial time, since in the former case it suffices to check if the number is equal to i (time proportional to $\log n_i$), whereas in the latter case it suffices to check the last bit of n_i.

Thus, the time required to select the rule to apply depends on the number of rules in each neuron. That is, $Z_i = O(|R_i|) = O(r)$, where $r = \max_{1 \leq i \leq m} |R_i|$ is the maximum number of rules which can be contained in a neuron.

As a consequence, the total time required to simulate t steps of Π is $t \cdot \big(O(m\,(r + \log(w + tQm) + \log Q + \log D)) + O\big((m^2 \log D)(m^2 + \log D + O(\log(w + tQm)))\big)\big)$.

We conclude this section by stressing that the above simulation could be performed in polynomial time because the regular expressions used in the rules are of a very restricted form. Indeed, there is a large amount of computational power hidden into the implicit mechanism that SN P systems use to decide whether a given rule can be applied or not, as proved in the following proposition. The difficulty of checking whether the contents of a neuron is into the regular set determined by the regular expression E of the rule is induced by the fact that we are dealing with unary languages. As told above, in these languages a string is uniquely determined by its length. Hence, a compact representation of the string is obtained by writing (in binary) its length, rather than by writing the string itself. This is an exponentially smaller representation, and consequently all the problems defined upon this representation become harder, as it happens for all "succint" representations (see [15], chapter 20). Concerning the succint version of the membership problem (is a given string into the language generated by E?) we can prove the following proposition.

Proposition 1. *Let Π be an SN P system having a single neuron, that contains the rule $E : a^c \to a; d$, where $c \geq 1$ and $d \geq 0$ are natural numbers, and E is any regular expression (for a unary language defined on the alphabet $\{a\}$). If E*

is described in a succint form, then deciding whether this rule can be applied is at least **NP***–complete.*

Proof. Let us show a polynomial time reduction from Subset Sum to this problem. Let $(V = \{v_1, v_2, \ldots, v_n\}, S)$ be an instance of Subset Sum. If we put $K = \max\{v_1, v_2, \ldots, v_n, S\}$, then the instance size is $\Theta(nK)$, as discussed in section 3.1. Consider the regular expression $E = E_1 \circ E_2 \circ \ldots \circ E_n$, where $E_i = (\lambda \cup \{a^{v_i}\})$, with λ the symbol that represents the empty word. Since we are dealing with unary languages in succint form, every string of $L(E)$ can be uniquely determined by its length, and thus we can write $L(E_i) = \{0, v_i\}$. $L(E)$ is just obtained by performing a language theoretic concatenation among the languages $L(E_i)$: $L(E) = L(E_1) \circ L(E_2) \circ \ldots \circ L(E_n)$. Clearly, the regular expression E can be represented using a string whose length is polynomial with respect to the size of the given instance of Subset Sum. It is immediately verified that $L(E)$ contains 2^n elements, bijectively associated with the subsets of V. Moreover, $a^S \in L(E)$ if and only if there exists a set $B \subseteq V$ such that $\sum_{b \in B} b = S$. Hence, checking whether a^S is in $L(E)$ or not is equivalent to solve the Subset Sum problem on the given instance (V, S).

5 Conclusions and Directions for Future Research

In this paper we have started to study the computational power of spiking neural P systems. In particular, by slightly extending the original definition given in [8] and [9] we have shown that by exploiting nondeterminism it is possible to solve **NP**–complete problems such as Subset Sum and 3-SAT.

Concerning deterministic systems, we have shown that the universal deterministic SN P systems defined in [8, 9] can be simulated by deterministic Turing machines with a polynomial slowdown. Surprisingly, this was possible only because the universal P systems described in [8, 9] use regular expressions of a very restricted form. In fact, it was shown that if we allow the use of general regular expressions then we can exploit the mechanism used by SN P systems to decide whether the contents of a neuron matches a regular expression to solve the **NP**–complete problem Subset Sum.

Further research is needed to fully understand the power of accepting SN P systems. In particular, by encoding their inputs as the distance between subsequent spikes we are limiting ourselves to use numbers expressed in unary notation. A more compact way to encode a k-bit natural number n would be to send a sequence of spikes during a prefixed sequence of k computation steps: the presence of a spike indicates a 1, its absence indicates a 0. It is not currently known whether in this way SN P systems can still solve numerical **NP**–complete problems such as Subset Sum, or whether a subsystem that converts integer numbers from binary to unary notation can be designed, as it was made in [10] for traditional P systems.

Acknowledgments

We gratefully thank Gheorghe Păun for introducing the authors to the stimulating subject of spiking neural P systems, and for asking us a "Milano theorem" (in the spirit of [22]) about their computational power, during the Fifth Brainstorming Week on Membrane Computing, held in Seville from January 29$^{\text{th}}$ to February 2$^{\text{nd}}$, 2007. Moreover, we are really indebted with him for suggesting us the standard nondeterministic SN P system that solves 3-SAT, exposed in section 3.

References

1. A. Alhazov, R. Freund, M. Oswald: Cell/symbol complexity of tissue P systems with symport/antiport rules. *Intern. J. Found. Computer Sci.*, 17, 1 (2006), 3–26.
2. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. In: M.A. Gutiérrez-Naranjo, Gh. Păun, A. Riscos-Núñez, F.J. Romero-Campero, eds., *Fourth Brainstorming Week on Membrane Computing*, Vol. I RGCN Report 02/2006, Research Group on Natural Computing, Sevilla University, Fénix Editora, 169–194.
3. T.H. Cormen, C.H. Leiserson, R.L. Rivest: *Introduction to Algorithms*. MIT Press, Boston, 1990.
4. R. Freund, Gh. Păun, M.J. Pérez-Jiménez: Tissue-like P systems with channel states. *Theoretical Computer Science*, 330 (2004), 101–116.
5. M.R. Garey, D.S. Johnson: *Computers and Intractability. A Guide to the Theory on* **NP***–Completeness*. W. H. Freeman and Company, 1979.
6. W. Gerstner, W. Kistler: *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press, 2002.
7. O.H. Ibarra, A. Păun, Gh. Păun, A. Rodríguez-Patón, P. Sosík, S. Woodworth: Normal forms for spiking neural P systems. *Theoretical Computer Science*, 372, 2-3 (2007), 196–217.
8. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
9. M. Ionescu, A. Păun, Gh. Păun, M.J. Pérez-Jiménez: Computing with spiking neural P systems: traces and small universal systems. In C. Mao, T. Yokomori, eds., *DNA Computing,* 12$^{\text{th}}$ *International Meeting on DNA Computing, DNA12*, Seoul, Korea, June 5-9, 2006, Revised Selected Papers. LNCS 4287, Springer, 2006, 1–16.
10. A. Leporati, C. Zandron, M.A. Gutiérrez-Naranjo: P systems with input in binary form. *International Journal of Foundations of Computer Science*, 17, 1 (2006), 127–146.
11. W. Maass: Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.
12. W. Maass, C. Bishop (eds.). *Pulsed Neural Networks*, MIT Press, Cambridge (MA), 1999.
13. C. Martín-Vide, J. Pazos, Gh. Păun, A. Rodríguez-Patón: A new class of symbolic abstract neural nets: Tissue P systems. In *Proceedings of COCOON 2002*, Singapore, LNCS 2387, Springer-Verlag, Berlin, 290–299.
14. M. Oswald: Independent agents in a globalized world modelled by tissue P systems. *Conf. Artificial Life and Robotics*, 2006.

15. C.H. Papadimitriou: *Computational Complexity*, Addison-Wesley, 1994.
16. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61 (2000), 108–143. See also Turku Centre for Computer Science — TUCS Report No. 208, 1998. Available at: *http://www.tucs.fi/Publications/techreports/TR208.php*
17. Gh. Păun: Computing with membranes. An Introduction. *Bulletin of the EATCS*, 67 (2/1999), 139–152.
18. Gh. Păun: *Membrane Computing. An Introduction*. Springer–Verlag, Berlin, 2002.
19. Gh. Păun, Y. Sakakibara, T. Yokomori: P systems on graphs of restricted forms. *Publicationes Mathematicae Debrecen*, 60 (2002), 635–660.
20. Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Infinite spike trains in spiking neural P systems. Submitted for publication.
21. G. Păun, G. Rozenberg: A guide to membrane computing. *Theoretical Computer Science*, 287, 1 (2002), 73–100.
22. C. Zandron, C. Ferretti, G. Mauri: Solving **NP**–complete problems using P systems with active membranes. In I. Antoniou, C.S. Calude, M.J. Dinneen, eds., *Unconventional Models of Computation*, Springer-Verlag, London, 2000, 289–301.
23. The P systems Web page: http://psystems.disco.unimib.it/