Spiking Neural P Systems with Neuron Division and Budding

Linqiang Pan[1,2], Gheorghe Păun[2,3], Mario J. Pérez-Jiménez[2]

[1] Department of Control Science and Engineering
Huazhong University of Science and Technology
Wuhan 430074, Hubei, China
lqpan@mail.hust.edu.cn, lqpan@us.es

[2] Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
marper@us.es

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

Summary. In order to enhance the efficiency of spiking neural P systems, we introduce the features of neuron division and neuron budding, which are processes inspired by neural stem cell division. As expected (as it is the case for P systems with active membranes), in this way we get the possibility to solve computationally hard problems in polynomial time. We illustrate this possibility with SAT problem.

1 Introduction

Spiking neural P systems (in short, SN P systems) were introduced in [6] in the framework of membrane computing [13] as a new class of computing devices which are inspired by the neurophysiological behavior of neurons sending electrical impulses (spikes) along axons to other neurons. Since then, many computational properties of SN P systems have been studied; for example, it has been proved that they are Turing-complete when considered as number computing devices [6], when used as language generators [3, 1] and also when computing functions [12].

Investigations related to the possibility to solve computationally hard problems by using SN P systems were first proposed in [2]. The idea was to encode the instances of decision problems in a number of spikes which are placed in an arbitrarily large pre-computed system at the beginning of the computation. It was shown that the resulting SN P systems are able to solve the **NP**-complete problem SAT (the satisfiability of propositional formulas expressed in conjunctive normal form) in a constant time. Slightly different solutions to SAT and 3-SAT by using SN

P systems with pre-computed resources were considered in [7]; here the encoding of an instance of the given problem is introduced into the pre-computed resources in a polynomial number of steps, while the truth values are assigned to the Boolean variables of the formula and the satisfiability of the clauses is checked. The answer associated with the instance of the problem is thus computed in a polynomial time. Finally, very simple semi-uniform and uniform solutions to the numerical **NP**-complete problem `Subset Sum` – by using SN P systems with exponential size pre-computed resources – have been presented in [8]. All the systems constructed above work in a deterministic way.

A different idea of constructing SN P systems for solving **NP**-complete problems was given in [10, 11], where the `Subset Sum` and `SAT` problems were considered. In these papers, the solutions are obtained in a semi-uniform or uniform way by using non-deterministic devices but without pre-computed resources. However, several ingredients are also added to SN P systems such as extended rules, the possibility to have a choice between spiking rules and forgetting rules, etc. An alternative to the constructions of [10, 11] was given in [9], where only standard SN P systems without delaying rules and having a uniform construction are used. However, it should be noted that the systems described in [9] either have an exponential size, or their computations last an exponential number of steps. Indeed, it has been proved in [11] that a deterministic SN P system of a polynomial size cannot solve an **NP**-complete problem in a polynomial time unless **P=NP**. Hence, under the assumption that $\mathbf{P} \neq \mathbf{NP}$, efficient solutions to **NP**-complete problems cannot be obtained without introducing features which enhance the efficiency of the system.

In this paper, neuron division and budding are introduced into the framework of SN P systems in order to enhance the efficiency of these systems. We exemplify this possibility with a uniform solution to `SAT` problem.

The biological motivation of introducing neuron division and budding into SN P systems comes from the recent discoveries in neurobiology related to neural stem cells – see, e.g., [4]. Neural stem cells persist throughout life within central nervous system in the adult mammalian brain, and this ensures a life-long contribution of new neurons to self-renewing nervous system with about 30000 new neurons being produced every day. Even in vitro, neural stem cells can be grown and extensively expanded for months. New neurons are produced by symmetric or asymmetric division. Two main neuron cell types are found: neuroblasts and astrocytes. The latter form a meshwork and are organized into channels. These observations are incorporated in SN P systems by considering neuron division and budding, and by providing a "synapse dictionary" according to which new synapses are generated, respectively.

The paper is organized as follows. In Section 2 we recall some mathematical preliminaries that will be used in the following. In Section 3 the formal definition of SN P systems with neuron division rules and neuron budding rules is given. In Section 4 we present a uniform family of SN P systems with neuron division and budding rules such that the systems can solve `SAT` problem in a polynomial time.

Section 5 concludes the paper and suggests some possible open problems for future work.

2 Prerequisites

We assume the reader to be familiar with basic elements about membrane computing, e.g., from [13] and [14], and formal language theory, as available in many monographs. We mention here only a few notions and notations which are used through the paper.

For an alphabet V, V^* denotes the set of all finite strings over V, with the empty string denoted by λ. The set of all nonempty strings over V is denoted by V^+. When $V = \{a\}$ is a singleton, then we simply write a^* and a^+ instead of $\{a\}^*$, $\{a\}^+$.

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. With each regular expression E we associate a language $L(E)$, defined in the following way: (i) $L(\lambda) = \{\lambda\}$ and $L(a) = \{a\}$, for all $a \in V$, (ii) $L((E_1) \cup (E_2)) = L(E_1) \cup L(E_2)$, $L((E_1)(E_2)) = L(E_1)L(E_2)$, and $L((E_1)^+) = (L(E_1))^+$, for all regular expressions E_1, E_2 over V. Non-necessary parentheses can be omitted when writing a regular expression, and also $(E)^+ \cup \{\lambda\}$ can be written as E^*.

3 SN P Systems with Neuron Division and Budding

As stated in the Introduction, SN P systems have been introduced in [6], in the framework of membrane computing. They can be considered as an evolution of P systems, corresponding to a shift from *cell-like* to *neural-like* architectures.

In SN P systems the cells (also called *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*. Every cell may also contain a number of *firing* and *forgetting* rules. Firing rules 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 applicability of each rule is determined by checking the contents of the neuron against a regular set associated with the rule. In each time unit, if a neuron can use one of its rules, then one of such rules must be used. If two or more rules could be applied, then only one of them is non-deterministically chosen. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other. Note that, as usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, and hence the functioning of the system is synchronized. When a cell sends out spikes it becomes "closed" (inactive) for a specified period

of time, a fact which 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, then there may be the possibility to apply a *forgetting rule*, that removes from the neuron a predefined number of spikes.

The structure of SN P systems (that is, the synapse graph) introduced in [6] is static. For both biological and mathematical motivations discussed in the Introduction, neuron division and budding are introduced into SN P systems. In this way, an exponential workspace can be generated in polynomial (even linear) time and computationally hard problems can be efficiently solved by means of a space-time tradeoff.

Formally, a *spiking neural P system with neuron division and budding* of (initial) degree $m \geq 1$ is a construct of the form

$$\Pi = (O, H, syn, n_1, \ldots, n_m, R, in, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. H is a finite set of *labels* for neurons;
3. $syn \subseteq H \times H$ is a *synapse dictionary*, with $(i, i) \notin syn$ for $i \in H$;
4. $n_i \geq 0$ is the *initial number of spikes* contained in neuron i, $i \in \{1, 2, \ldots, m\}$;
5. R is a finite set of *developmental rules*, of the following forms:
   (1) *extended firing* (also called *spiking*) rule $[E/a^c \to a^p; d]_i$, where $i \in H$, E is a regular expression over a, and $c \geq 1$, $p \geq 0$, $d \geq 0$, with the restriction $c \geq p$;
   (2) *neuron division rule* $[E]_i \to [\,]_j \parallel [\,]_k$, where E is a regular expression and $i, j, k \in H$;
   (3) *neuron budding rule* $[E]_i \to [\,]_i / [\,]_j$, where E is a regular expression and $i, j \in H$;
6. $in, out \in H$ indicate the *input* and the *output* neurons of Π.

Note that we have presented here an SN P system in a way slightly different from the usual definition present in the literature, where the neurons present initially in the system are explicitly listed as $\sigma_i = (n_i, R_i)$, where $1 \leq i \leq m$ and R_i are the rules associated with neuron with label i. In what follows we will refer to neuron with label $i \in H$ also denoting it with σ_i.

It is worth to mention that by applying division rules different neurons can appear with the same label. In this context, $(i, j) \in syn$ means the following: there exist synapses from each neuron with label i to each neuron with label j.

If an extended firing rule $[E/a^c \to a^p; d]_i$ has $E = a^c$, then we will write it in the simplified form $[a^c \to a^p; d]_i$; similarly, if a rule $[E/a^c \to a^p; d]_i$ has $d = 0$, then we can simply write it as $[E/a^c \to a^p]_i$; hence, if a rule $[E/a^c \to a^p; d]_i$ has

$E = a^c$ and $d = 0$, then we can write $[a^c \to a^p]_i$. A rule $[E/a^c \to a^p]_i$ with $p = 0$ is written in the form $[E/a^c \to \lambda]_i$ and is called *extended forgetting* rule. Rules of the types $[E/a^c \to a; d]_i$ and $[a^c \to \lambda]_i$ are said to be *standard*.

If a neuron σ_i contains k spikes and $a^k \in L(E), k \geq c$, then the rule $[E/a^c \to a^p; d]_i$ is enabled and it can be applied. This means consuming (removing) c spikes (thus only $k - c$ spikes remain in neuron σ_i); the neuron is fired, and it produces p spikes after d time units. If $d = 0$, then the spikes are emitted immediately; if $d = 1$, then the spikes are 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 open again, so that it can receive spikes (which can be used starting with the step $t + d + 1$, when the neuron can again apply rules). Once emitted from neuron σ_i, the p spikes reach immediately all neurons σ_j such that there is a synapse going from σ_i to σ_j and which are open, that is, the p spikes are replicated and each target neuron receives p spikes; as stated above, spikes sent to a closed neuron are "lost", that is, they are removed from the system. In the case of the output neuron, p spikes are also sent to the environment. Of course, if neuron σ_i has no synapse leaving from it, then the produced spikes are lost. If the rule is a forgetting one of the form $[E/a^c \to \lambda]_i$, then, when it is applied, $c \geq 1$ spikes are removed. When a neuron is closed, none of its rules can be used until it becomes open again.

If (1) a neuron σ_i contains s spikes and $a^s \in L(E)$, and (2) there is no neuron σ_g such that the synapse (g, i) or (i, g) exists in the system, for some $g \in \{j, k\}$, then the division rule $[E]_i \to [\,]_j \parallel [\,]_k$ is enabled and it can be applied. This means that consuming all these s spikes the neuron σ_i is divided into two neurons, σ_j and σ_k. The new neurons contain no spike in the moment when they are created. They can have different labels, but they inherit the synapses that the father neuron already has (if there is a synapse from neuron σ_g to the neuron σ_i, then in the process of division one synapse from neuron σ_g to new neuron σ_j and another one from σ_g to σ_k are established; similarly, if there is a synapse from the neuron σ_i to neuron σ_h, then one synapse from σ_j to σ_h and another one from σ_k to σ_h are established). Note that the restriction provided by condition (2) to the use of the rule ensures that no synapse (j, j) or (k, k) appears. Except for the inheritance of synapses, the new neurons produced by division can have new synapses as provided by the synapse dictionary. Note that during the computation, it is possible that a synapse between neurons involved in the division rule and neurons existing in the system will appear that is not in the synapse dictionary *syn*, because of the inheritance of synapses. Therefore, the synapse dictionary *syn* has two functions: one is to deduce the initial topological structure of the SN P system (a directed graph), for example, if there are neurons $\sigma_1, \ldots, \sigma_k$ at the beginning of computation, then the initial topological structure of the system is $syn \cap (\{1, 2, \ldots, k\} \times \{1, 2, \ldots, k\})$; another function is to guide the synapse establishment associated with the new

neurons generated by neuron division or neuron budding. That is why we call *syn* a synapses dictionary.

If (1) a neuron σ_i contains s spikes, and $a^s \in L(E)$, and (2) there is no neuron σ_j such that the synapse (i, j) exists in the system,then the budding rule $[E]_i \rightarrow [\ ]_i/[\ ]_j$ is enabled and it can be applied. This means that consuming all the s spikes a new neuron is created, σ_j. Both neurons are empty after applying the rule. The neuron σ_i inherits the synapses going to it before using the rule. The neuron σ_j created by budding by neuron σ_i inherits the synapses going out of σ_i before budding, that is, if there is a synapse from neuron σ_i to some neuron σ_h, then a synapse from neuron σ_j to neuron σ_h is established (condition (2) ensures the fact that no synapse (j, j) appears). There is also a synapse (i, j) between neurons σ_i and σ_j. Except for the above synapses associated with neurons σ_i and σ_j, other synapses associated with neuron σ_j can be established according to the synapses dictionary *syn* as in the case of neuron division rule.

In each time unit, if a neuron σ_i can use one of its rules, then a rule from R *must* be used. If several rules are enabled in neuron σ_i, irrespective of their types (spiking, dividing, or budding) then only one of them is chosen non-deterministically. When a spiking rule is used, the state of neuron σ_i (open or closed) depends on the delay d. When a neuron division rule or neuron budding rule is applied, at this step the associated neuron is closed, it cannot receive spikes. In the next step, the neurons obtained by division or budding will be open and can receive spikes. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other.

It is worth noting here that the two neurons produced by a division rule can have labels different from each other and from the divided neuron, and that they are placed "in parallel", while in the budding case the old neuron (consumes all its spikes and) produces one new neuron which is placed "serially", after the neuron which budded.

The *configuration* of the system is described by the topology structure of the system, the number of spikes associated with each neuron, and the *state* of each neuron (open or closed). Using the rules as described above, one can define *transitions* among configurations. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation *halts* if it reaches a configuration where all neurons are open and no rule can be used.

Traditionally, the input of an SN P system used in the accepting mode is provided in the form of a spike train, a sequence of steps when one spike or no spike enters the input neuron. In what follows we need several spikes at a time to come into the system via the input neuron, that is we consider "generalized spike trains", written in the form $a^{i_1} \cdot a^{i_2} \cdot \ldots \cdot a^{i_r}$, where $r \geq 1, i_j \geq 0$ for each $1 \leq j \leq r$. The meaning is that i_j spikes are introduced in neuron σ_{in} in step j (all these i_j spikes are provided at the same time). Note that we can have $i_j = 0$, which means that no spike is introduced in the input neuron. The period which separate the "packages" a^{i_j} of spikes is necessary in order to make clear that we do not have

here a concatenation of the strings describing these "packages", but a sequence of blocks (more formally, a sequence of multisets over the singleton alphabet O).

Spiking neural P systems can be used to solve decision problems, both in a *semi–uniform* and in a *uniform* way. When solving a problem $\mathcal{Q}$ in the *semi–uniform* setting, for each specified instance $\mathcal{I}$ of $\mathcal{Q}$ an SN P system $\Pi_{\mathcal{Q},\mathcal{I}}$ is (i) built by a Turing machine in a polynomial time (with respect to the size of $\mathcal{I}$), (ii) its structure and initial configuration depend upon $\mathcal{I}$, and (iii) it halts (or emits a specified number of spikes in a given interval of time) if and only if $\mathcal{I}$ is a positive instance of $\mathcal{Q}$. On the other hand, a *uniform* solution of $\mathcal{Q}$ consists of a family $\{\Pi_{\mathcal{Q}}(n)\}_{n\in\mathbb{N}}$ of SN P systems that are built by a Turing machine in a polynomial time (with respect to the size n). When having an instance $\mathcal{I} \in \mathcal{Q}$ of size n, we introduce a polynomial (in n) number of spikes in a designated input neuron of $\Pi_{\mathcal{Q}}(n)$ and the computation halts (or, alternatively, a specified number of spikes is emitted in a given interval of time) if and only if $\mathcal{I}$ is a positive instance. The preference for uniform solutions over semi–uniform ones is given by the fact that they are more strictly related to the structure of the problem, rather than to specific instances. Indeed, in the semi–uniform setting we do not even need any input neuron, as the instance of the problem can be embedded into the initial configuration of the system from the very beginning.

4 A Uniform Solution to SAT Problem

Let us consider the **NP**-complete decision problem SAT [5]. The instances of SAT depend upon two parameters: the number n of variables, and the number m of clauses. We recall that a *clause* is a disjunction of literals, occurrences of x_i or $\neg x_i$, built on a given set $X = \{x_1, x_2, \ldots, x_n\}$ of Boolean variables. Without loss of generality, we can avoid the clauses in which the same literal is repeated or both the literals x_i and $\neg x_i$, for any $1 \leq i \leq n$, occur. In this way, a clause can be seen as a set of at most n literals. An *assignment* of the variables $x_1, x_2, \ldots, x_n$ is a mapping $T : X \to \{0, 1\}$ 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 (*true*) as a result.

We can now formally state the SAT problem as follows.

Problem 1. NAME: SAT.

- INSTANCE: a set $C = \{C_1, C_2, \ldots, C_m\}$ of 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?

Equivalently, we can say that an instance of SAT is a propositional formula $\gamma_{n,m} = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, expressed in the conjunctive normal form as a conjunction

of m clauses, where each clause is a disjunction of literals built using the Boolean variables $x_1, x_2, \ldots, x_n$. With a little abuse of notation, from now on we will denote by $SAT(n,m)$ the set of instances of SAT which have n variables and m clauses.

Let us consider the polynomial time computable function $\langle n,m\rangle = ((m+n)(m+n+1)/2)+m$ (the pair function), which is a primitive recursive and bijective function from $\mathbb{N}^2$ to $\mathbb{N}$. Let us build a uniform family $\{\Pi_{SAT}(\langle n,m\rangle)\}_{n,m\in\mathbb{N}}$ of SN P systems such that for all $n,m\in\mathbb{N}$ the system $\Pi_{SAT}(\langle n,m\rangle)$ solves all the instances of $SAT(n,m)$ in a number of steps which is linear in both n and m. All the systems $\Pi_{SAT}(\langle n,m\rangle)$ will work in a deterministic way.

Because the construction is uniform, we need a way to encode any given instance $\gamma_{n,m}$ of $SAT(n,m)$. As stated above, each clause C_i of $\gamma_{n,m}$ can be seen as a disjunction of at most n literals, and thus for each $j\in\{1,2,\ldots,n\}$ either x_j occurs in C_i, or $\neg x_j$ occurs, or none of them occurs. In order to distinguish these three situations we define the *spike variables* $\alpha_{i,j}$, for $1\leq i\leq m$ and $1\leq j\leq n$, as variables whose values are amounts of spikes; we assign to them the following values:

$$\alpha_{i,j}=\begin{cases} a, & \text{if } x_j \text{ occurs in } C_i;\\ a^2, & \text{if } \neg x_j \text{ occurs in } C_i;\\ a^0, & \text{otherwise.}\end{cases}$$

In this way, clause C_i will be represented by the sequence $\alpha_{i,1}\cdot\alpha_{i,2}\cdot\ldots\cdot\alpha_{i,n}$ of spike variables; in order to represent the entire formula $\gamma_{n,m}$ we just concatenate the representations of the single clauses, thus obtaining the generalized spike train $\alpha_{1,1}\cdot\alpha_{1,2}\cdot\ldots\cdot\alpha_{1,n}.\alpha_{2,1}\cdot\alpha_{2,2}\cdot\ldots\cdot\alpha_{2,n}\cdot\ldots\cdot\alpha_{m,1}\cdot\alpha_{m,1}\cdot\ldots\cdot\alpha_{m,n}$. As an example, the representation of $\gamma_{3,2}=(x_1\vee\neg x_2)(x_1\vee x_3)$ is the sequence $a\cdot a^2\cdot a^0\cdot a\cdot a^0\cdot a$. In order to let the systems have enough time to generate necessary workspace before computing the instances of $SAT(n,m)$, a spiking train $(a^0\cdot)^{2n}$ is added in front of the formula encoding spike train $\alpha_{1,1}\cdot\alpha_{1,2}\cdot\ldots\cdot\alpha_{1,n}.\alpha_{2,1}\cdot\alpha_{2,2}\cdot\ldots\cdot\alpha_{2,n}\cdot\ldots\cdot\alpha_{m,1}\cdot\alpha_{m,1}\cdot\ldots\cdot\alpha_{m,n}$. In general, for any given instance $\gamma_{n,m}$ of $SAT(n,m)$, the encoding sequence is $cod(\gamma_{n,m})=(a^0\cdot)^{2n}\alpha_{1,1}\cdot\alpha_{1,2}\cdot\ldots\cdot\alpha_{1,n}\cdot\alpha_{2,1}\cdot\alpha_{2,2}\cdot\ldots\cdot\alpha_{2,n}\cdot\ldots\cdot\alpha_{m,1}\cdot\alpha_{m,1}\cdot\ldots\cdot\alpha_{m,n}$.

For each $n,m\in\mathbb{N}$, we construct

$$\Pi(\langle n,m\rangle)=(O,H,syn,n_1,\ldots,n_q,R,in,out),$$

with the following components:

The initial degree of the system is $q=4n+7$;

$O=\{a\}$;

$$\begin{aligned}H=\ &\{in,out,cl\}\cup\{d_i\mid i=0,1,\ldots,n\}\\ &\cup\{Cx_i\mid i=1,2,\ldots,n\}\cup\{Cx_i0\mid i=1,2,\ldots,n\}\\ &\cup\{Cx_i1\mid i=1,2,\ldots,n\}\cup\{t_i\mid i=1,2,\ldots,n\}\\ &\cup\{f_i\mid i=1,2,\ldots,n\}\cup\{0,1,2,3\};\end{aligned}$$

$syn = \{(d_i, d_{i+1}) \mid i = 0, 1, \ldots, n-1\} \cup \{(d_n, d_1)\}$
$\cup \{(in, Cx_i) \mid i = 1, 2, \ldots, n\} \cup \{(d_i, Cx_i) \mid i = 1, 2, \ldots, n\}$
$\cup \{(Cx_i, Cx_i 0) \mid i = 1, 2, \ldots, n\} \cup \{(Cx_i, Cx_i 1) \mid i = 1, 2, \ldots, n\}$
$\cup \{(i+1, i) \mid i = 0, 1, 2\} \cup \{(1, 2), (0, out)\}$
$\cup \{(Cx_i 1, t_i) \mid i = 1, 2, \ldots, n\} \cup \{(Cx_i 0, f_i) \mid i = 1, 2, \ldots, n\}$;
$n_{d_0} = n_0 = n_2 = n_3 = 1$, $n_{d_1} = 6$, and there is no spike in the other neurons;

R is the following set of rules:

(1) **spiking rules:**
$[a \to a]_{in}$,
$[a^2 \to a^2]_{in}$,
$[a \to a; 2n + nm]_{d_0}$,
$[a^4 \to a^4]_i$, $i = d_1, \ldots, d_n$,
$[a^5 \to \lambda]_{d_1}$,
$[a^6 \to a^4; 2n + 1]_{d_1}$,
$[a \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^2 \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^4 \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^5 \to a^5; n - i]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^6 \to a^6; n - i]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^5 \to a^4]_{Cx_i 1}$, $i = 1, 2, \ldots, n$,
$[a^6 \to \lambda]_{Cx_i 1}$, $i = 1, 2, \ldots, n$,
$[a^5 \to \lambda]_{Cx_i 0}$, $i = 1, 2, \ldots, n$,
$[a^6 \to a^4]_{Cx_i 0}$, $i = 1, 2, \ldots, n$,
$[(a^4)^+ \to a]_{t_i}$, $i = 1, 2, \ldots, n$,
$[(a^4)^+ \to a]_{f_i}$, $i = 1, 2, \ldots, n$,
$[a^{4k-1} \to \lambda]_{t_i}$, $k = 1, 2, \ldots, n$, $i = 1, 2, \ldots, n$,
$[a^{4k-1} \to \lambda]_{f_i}$, $k = 1, 2, \ldots, n$, $i = 1, 2, \ldots, n$,
$[a^m \to a^2]_{cl}$,
$[(a^2)^+ / a \to a]_{out}$,
$[a \to a]_i$, $i = 1, 2$,
$[a^2 \to \lambda]_2$,
$[a \to a; 2n - 1]_3$;

(2) **neuron division rules:**
$[a]_0 \to [\,]_{t_1} \parallel [\,]_{f_1}$,
$[a]_{t_i} \to [\,]_{t_{i+1}} \parallel [\,]_{f_{i+1}}$, $i = 1, 2, \ldots, n-1$,
$[a]_{f_i} \to [\,]_{t_{i+1}} \parallel [\,]_{f_{i+1}}$, $i = 1, 2, \ldots, n-1$;

(3) **neuron budding rules:**
$[a]_{t_n} \to [\,]_{t_n} / [\,]_{cl}$,
$[a]_{f_n} \to [\,]_{f_n} / [\,]_{cl}$.

The solution of the SAT problem is obtained by means of a brute force algorithm in the framework of SN P systems with neuron division and budding. Our strategy consists in the following phases:

- *Generation Stage*: The neuron division and budding are applied to generate an exponential number of neurons such that each possible assignment of variables $x_1, x_2, \ldots, x_n$ is represented by a neuron (with associated connections with other neurons by synapses).
- *Input Stage*: The system reads the encoding of the given instance of SAT.
- *Satisfiability Checking Stage*: The system checks whether or not there exists an assignment of variables $x_1, x_2, \ldots, x_n$ that satisfies all the clauses in the propositional formula C.
- *Output Stage*: According to the result of the previous stage, the system sends a spike to the environment if the answer is positive; otherwise, the system does not send any spike to the environment.

Let us have an overview of the computation. The initial structure of the system is shown in Figure 1 (in the figures which follow we only present the spiking and the forgetting rules, but not also the division and budding rules). The first three layers of the system constitutes the input module. The neuron σ_0 and its offsprings will be used to generate an exponential workspace by neuron division and budding rules. The auxiliary neurons σ_1, σ_2, and σ_3 supply necessary spikes to the neuron σ_0 and its offsprings for neuron division and budding rules. The neuron σ_{out} is used to output the result.

Generation Stage: By the way of the encoding of instances, it is easy to see that the spike variables $\alpha_{i,j}$ will be introduced into neuron σ_{in} from step $2n+1$ (it takes $2n$ steps to read $(a^0\cdot)^{2n}$ of $cod(\gamma_{n,m})$). In the first $2n$ steps, the system generates an exponential workspace; after that, the system checks the satisfiability, and outputs the result.

The neuron σ_0 contains one spike, the rule $[a]_0 \to [\,]_{t_1} \parallel [\,]_{f_1}$ is applied, and two neurons σ_{t_1} and σ_{f_1} are generated. They have the associated synapses $(1, f_1)$, $(1, t_1)$, (t_1, out), (f_1, out), $(Cx_1 1, t_1)$ and $(Cx_1 0, f_1)$, where the first 4 synapses are obtained by the heritage of the synapses $(0, out)$ and $(1, 0)$, respectively, and the last 2 synapses are established by the synapse dictionary. The auxiliary neuron σ_2 sends one spike to neuron σ_1, and at step 2 neuron σ_1 sends this spike to neurons σ_{t_1} and σ_{f_1} for the next division. At step 1, the neuron σ_3 contains one spike, and the rule $[a \to a; 2n-1]_3$ is applied. It will send one spike to neuron σ_2 at step $2n$ because of the delay $2n-1$. (As we will see, at step $2n$, neuron σ_1 also sends one spike to neuron σ_2, so neuron σ_2 will have 2 spikes, and the rule $[a^2 \to \lambda]_2$ will be applied. In this way, after step $2n$, the auxiliary neurons stop the work of supplying spikes for division and budding.) The structure of the system after step 1 is shown in Figure 2.

At step 2, neuron σ_1 sends one spike to neurons σ_2, σ_{t_1}, and σ_{f_1}. In the next step, neuron σ_2 sends one spike back to neuron σ_1; in this way, the auxiliary neurons σ_1, σ_2, and σ_3 supply spikes for division and budding every two steps in

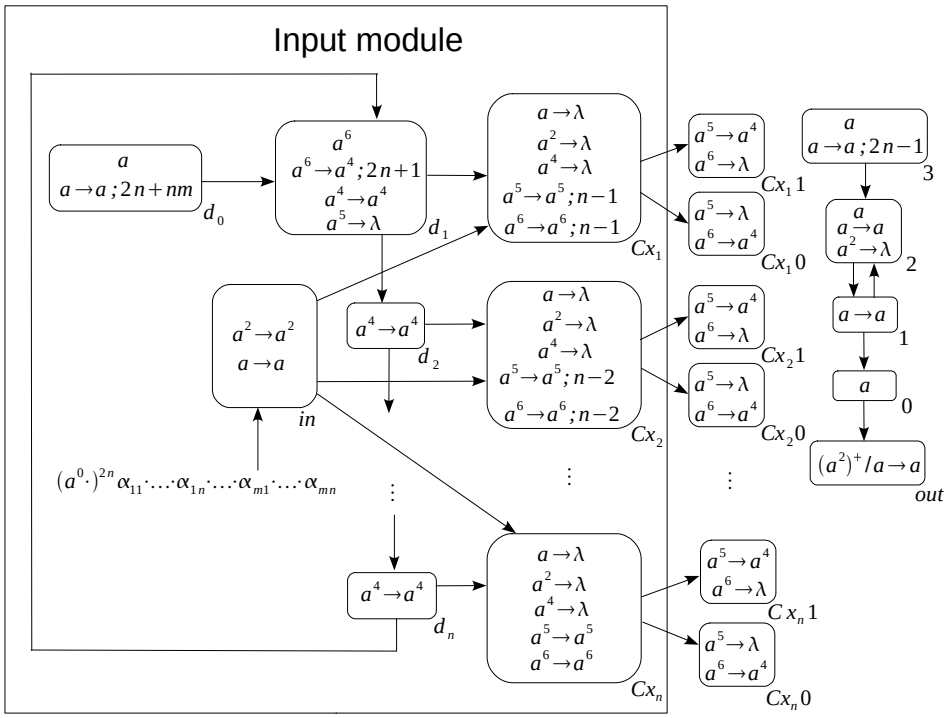

Fig. 1. The initial structure of system $\Pi(\langle n, m\rangle)$

the first $2n$ steps. At step 3, only division rules can be applied in neurons σ_{t_1} and σ_{f_1}, these two neurons are divided, and the associated synapses are obtained by heritage or synapse dictionary. The four neurons with labels t_2 or f_2 correspond to assignments where (1) $x_1 = 1$ and $x_2 = 1$, (2) $x_1 = 1$ and $x_2 = 0$, (3) $x_1 = 0$ and $x_2 = 1$, (4) $x_1 = 0$ and $x_2 = 0$, respectively. The neuron $Cx_1 1$ (encoding that x_1 appears in a clause) has synapses from it to neurons whose corresponding assignments have $x_1 = 1$. That is, assignments with $x_1 = 1$ satisfy clauses where x_1 appears. The structure of the system after step 3 is shown in Figure 3. The neuron division is iterated until 2^n neurons with labels t_n or f_n appear at step $2n - 1$. The corresponding structure after step $2n - 1$ is shown in Figure 4.

At step $2n$, each neuron with label t_n or f_n obtains one spike from neuron σ_1, then in next step the budding rules $[a]_{t_n} \to [\,]_{t_n} / [\,]_{cl}$ and $[a]_{f_n} \to [\,]_{f_n} / [\,]_{cl}$ are applied. Each created neuron σ_{cl} has synapses (t_n, cl) or (f_n, cl) and (cl, out) by heritage. At step $2n$, neuron σ_1 also sends one spike to neuron σ_2, at the time, neuron σ_3 sends one spike to neuron σ_2. So neuron σ_2 has two spikes, and the rule $[a^2 \to \lambda]_2$ is applied at step $2n+1$. After that, the auxiliary neurons cannot supply spikes any more, and the system passes to read the encoding of given instance. The structure of the system after step $2n + 1$ is shown in Figure 5.

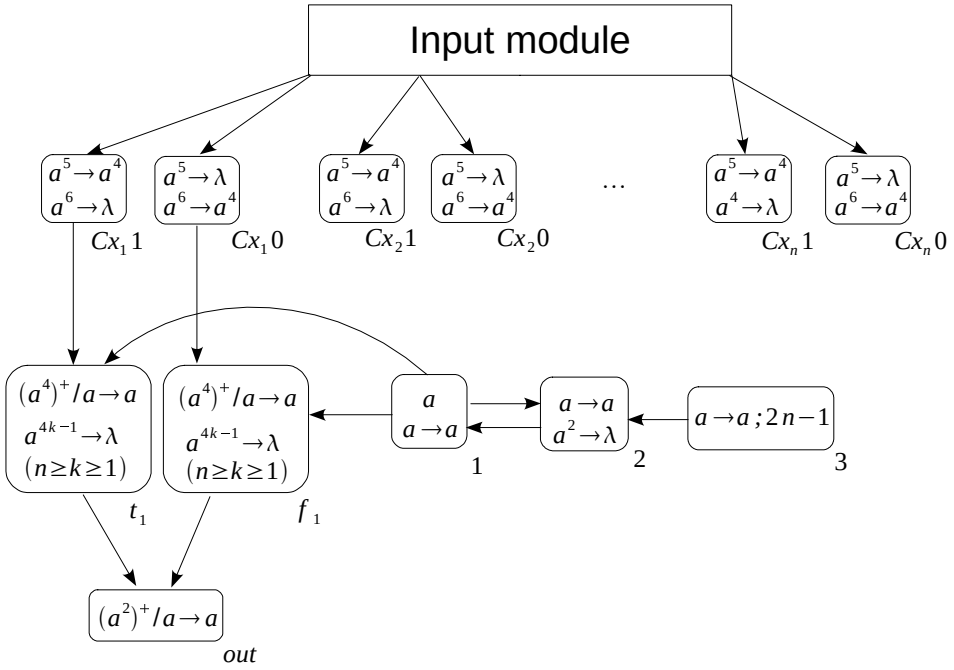

Fig. 2. The structure of system $\Pi(\langle n, m \rangle)$ after step 1

Input Stage: The input module consists of $2n + 2$ neurons, which are in the layers $1 - 3$ as illustrated in Figure 1; σ_{in} is the unique input neuron. The spikes of the encoding sequence $code(\gamma_{n,m})$ are introduced into σ_{in} one "package" by one "package", starting from step 1. It takes $2n$ steps to introduce $(a^0 \cdot)^{2n}$ into neuron σ_{in}. At step $2n + 1$, the value of the first spike variable α_{11}, which is the virtual symbol that represents the occurrence of the first variable in the first clause, enters into neuron σ_{in}. In the next step, the value of the spike variable α_{11} is replicated and sent to neurons σ_{Cx_i}, for all $i \in \{1, 2, \ldots, n\}$; in the meanwhile, neuron σ_{d_1} send four auxiliary spikes to neurons σ_{Cx_1} and σ_{d_2} (the rule $[a^6 \to a^4; 2n+1]_{d_1}$ is applied at step 1). Hence, neuron σ_{Cx_1} will contain 4, 5 or 6 spikes: if x_1 occurs in C_1, then neuron σ_{Cx_1} collects 5 spikes; if $\neg x_1$ occurs in C_1, then it collects 6 spikes; if neither x_1 nor $\neg x_1$ occur in C_1, then it collects 4 spikes. Moreover, if neuron σ_{Cx_1} has received 5 or 6 spikes, then it will be closed for $n - 1$ steps, according to the delay associated with the rules in it; on the other hand, if 4 spikes are received, then they are deleted and the neuron remains open. At step $2n + 3$, the value of the second spike variable α_{12} from neuron σ_{in} is distributed to neurons σ_{Cx_i}, $2 \leq i \leq n$, where the spikes corresponding to α_{11} are deleted by the rules $[a \to \lambda]_{Cx_i}$ and $[a^2 \to \lambda]_{Cx_i}$, $2 \leq i \leq n$. At the same time, the four auxiliary spikes are duplicated and one copy of them enters into neurons σ_{Cx_2} and σ_{d_3}, respectively. The neuron σ_{Cx_2} will be closed for $n - 2$ steps only if it contains

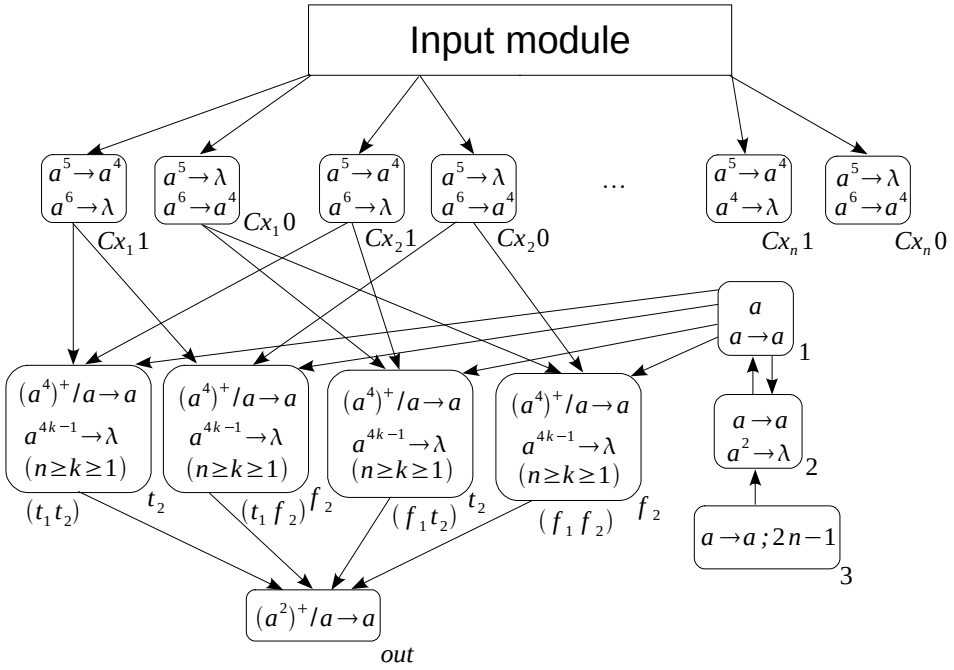

Fig. 3. The structure of system $\Pi(\langle n, m\rangle)$ after step 3

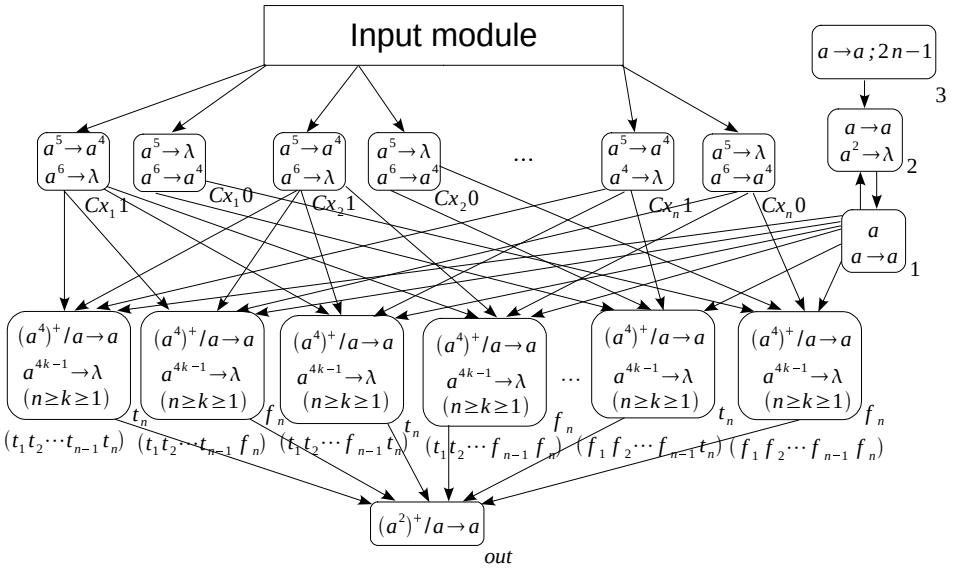

Fig. 4. The structure of system $\Pi(\langle n, m\rangle)$ after step $2n-1$

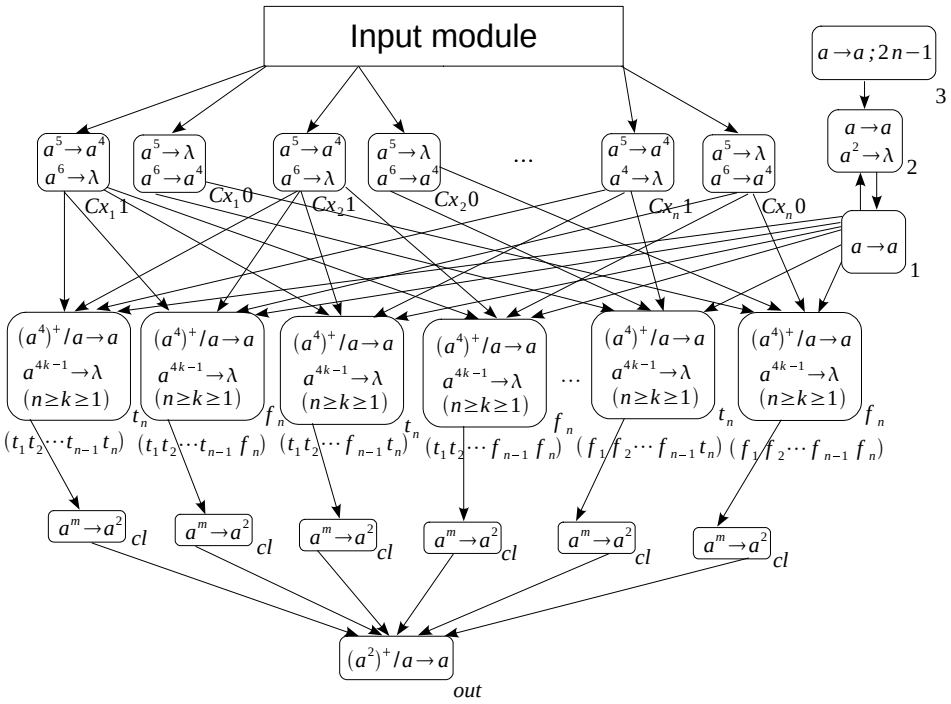

Fig. 5. The structure of system $\Pi(\langle n, m \rangle)$ after step $2n + 1$

5 or 6 spikes, which means that this neuron will not receive any spike during this period. In neurons σ_{Cx_i}, $3 \leq i \leq n$, the spikes represented by α_{12} are forgotten in the next step.

In this way, the values of the spike variables are introduced and delayed in the corresponding neurons until the value of the spike variable α_{1n} of the first clause and the four auxiliary spikes enter together into neuron σ_{Cx_n} at step $3n+1$. At that moment, the representation of the first clause of $\gamma_{n,m}$ has been entirely introduced in the system, and the second clause starts to enter into the input module. In general, it takes $mn + 1$ steps to introduce the whole sequence $code(\gamma_{n,m})$ in the system, and the input process is completed at step $2n + nm + 1$.

At step $2n + nm + 1$, the neuron σ_{d_n} sends four spike to neuron σ_{d_1}. At the same time, the auxiliary neuron σ_{d_0} also sends a spike to the neuron σ_{d_1} (the rule $[a \to a; 2n + nm]_{d_0}$ is used at the first step of the computation). So neuron σ_{d_1} contains 5 spikes, and in the next step these 5 spikes are forgotten by the rule $[a^5 \to \lambda]_{d_1}$. This ensures that the system eventually halts.

Satisfiability Checking Stage: Once all the values of spike variables α_{1i} $(1 \leq i \leq n)$, representing the first clause, have appeared in their corresponding neurons σ_{Cx_i} in layer 3, together with a copy of the four auxiliary spikes, all the spikes contained in σ_{Cx_i} are duplicated and sent simultaneously to the pair of neurons

σ_{Cx_i1} and σ_{Cx_i0}, for $i \in \{1, 2, \ldots, n\}$, at step $3n + 2$. In this way, each neuron σ_{Cx_i1} and σ_{Cx_i0} receives 5 or 6 spikes when x_i or $\neg x_i$ occurs in C_1, respectively, whereas it receives no spikes when neither x_i nor $\neg x_i$ occurs in C_1. In general, if neuron σ_{Cx_i1} receives 5 spikes, then the literal x_i occurs in the current clause (say C_j), and thus the clause is satisfied by all those assignments in which x_i is true. Neuron σ_{Cx_i0} will also receive 5 spikes, but they will be deleted during the next computation step. On the other hand, if neuron σ_{Cx_i1} receives 6 spikes, then the literal $\neg x_i$ occurs in C_j, and the clause is satisfied by those assignments in which x_i is false. Since neuron σ_{Cx_i1} is designed to process the case in which x_i occurs in C_j, it will delete its 6 spikes. However, neuron σ_{Cx_i0} will also have received 6 spikes, and this time it will send four spikes to those neurons which are bijectively associated with the assignments for which x_i is false (refer to the generation stage for the corresponding synapses). In the next step, those neurons with label t_n or f_n that received at least four spikes send one spike to the corresponding neurons σ_{cl} (the remaining spikes will be forgotten; note that the number of spikes received in neurons with label t_n or f_n is not more than $4n$, because, without loss of generality, we assume that the same literal is not repeated and at most one of literals x_i or $\neg x_i$, for any $1 \leq i \leq n$, can occur in a clause; that is, a clause is a disjunction of at most n literals), with the meaning that the clause is satisfied by the assignments in which x_i is false. This occurs in step $3n+4$. Thus, the check for the satisfiability of the first clause has been performed; in a similar way, the check for the remaining clauses can proceed. All the clauses can thus be checked to see whether there exist assignments that satisfy all of them.

If there exist some assignments that satisfy all the clauses of $\gamma_{n,m}$, then the corresponding neurons with label cl succeed to accumulate m spikes. Thus, the rule $[a^m \to a^2]_{cl}$ can be applied in these neurons. The satisfiability checking process is completed at step $2n + mn + 4$.

Output Stage: From the above explanation, it is not difficult to see that the output neuron receives spikes if and only if $\gamma_{n,m}$ is *true*. Furthermore, the output neuron sends exactly one spike to the environment at step $2n+mn+6$ if and only if $\gamma_{n,m}$ is *true*.

From the previous explanations, one can see that the system correctly answers the question whether or not $\gamma_{n,m}$ is satisfiable. The duration of the computation is polynomial in term of n and m: if the answer is positive, then the system sends one spike to the environment at step $2n + mn + 6$; if the answer is negative, then the system halts in $2n+mn+6$ steps, but does not send any spike to the environment.

Finally, we show that the family $\Pi = \{\Pi(\langle n, m\rangle) \mid n, m \in \mathbb{N}\}$ is polynomially uniform by deterministic Turing machines. We first note that the sets of rules associated with the system $\Pi(\langle n, m\rangle)$ are recursive. Hence, it is enough to note that the amount of necessary resources for defining each system is linear with respect to n, and this is indeed the case, since those resources are the following:

1. Size of the alphabet: $1 \in O(1)$.
2. Initial number of neurons: $4n + 7 \in O(n)$.
3. Initial number of spikes: $9 \in O(1)$.

4. Number of labels for neurons: $6n + 8 \in O(n)$.
5. Size of synapse dictionary: $7n + 6 \in O(n)$.
6. Number of rules: $2n^2 + 14n + 12 \in O(n^2)$.

5 Conclusions and Remarks

With computer science and biological motivation, neuron division and neuron budding are introduced into the framework of spiking neural P systems. We have proven that spiking neural P systems with neuron division and neuron budding can solve **NP**-complete problems in polynomial time. We exemplify this possibility with `SAT` problem.

Both neuron division rules and neuron budding rules can generate exponential workspace in linear time. In this sense, we used a double "power" to solve `SAT` problem in the systems constructed in this paper. It remains open to design efficient spiking neural P systems with either neuron division rules or neuron budding rules, but not both kinds of rules, for solving **NP**-complete problem.

Actually, we have here a larger set of problems. Besides the budding rules of the form considered above, we can also define "serial division rules", of the form $[E]_i \rightarrow [\ ]_j/[\ ]_k$, where E is a regular expression and $i, j, k \in H$. The meaning is obvious: neuron σ_i produces two neurons, σ_j and σ_k, with possibly new labels j, k, linked by synapses as in the case of budding rules. Note that a budding rule is a particular case, with $i = j$. Thus, we can consider three types of rules: parallel division rules, budding rules (these two types were considered above), and serial division rules. Are rules of a single type sufficient in order to devise SN P systems which solve computationally hard problems in polynomial time?

For cell-like P systems, besides membrane division, there is another operation which was proved to provide ways to generate an exponential workspace in polynomial time, useful for trading space for time in solving **NP**-complete problem, namely membrane creation (a membrane is created from objects present in other membranes). Can this idea be also extended to SN P systems? Note that in the case of SN P systems we do not have neurons inside neurons, neither spikes outside neurons, hence this issue does not look easy to handle; maybe further ingredients should be added, such as glia cells, astrocytes, etc.

Acknowledgements

Valuable comments from Jun Wang are greatly appreciated. The work of L. Pan was supported by National Natural Science Foundation of China (Grant Nos. 60674106, 30870826, 60703047, and 60803113), Program for New Century Excellent Talents in University (NCET-05-0612), Ph.D. Programs Foundation of Ministry of Education of China (20060487014), Chenguang Program of Wuhan (200750731262), HUST-SRF (2007Z015A), and Natural Science Foundation of

Hubei Province (2008CDB113 and 2008CDB180). The work of the last two authors was supported by Project of Excellence with *Investigador de Reconocida Valia*, from Junta de Andalucia, grant P08 – TIC 04200.

References

1. H. Chen, R. Freund, M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez: On string languages generated by spiking neural P systems. *Fundamenta Informaticae*, 75 (2007), 141–162.
2. H. Chen, M. Ionescu, T.-O. Ishdorj: On the efficiency of spiking neural P systems. *Proceedings of the* 8th *International Conference on Electronics, Information, and Communication*, Ulanbator, Mongolia, June 2006, 49–52.
3. H. Chen, M. Ionescu, T.-O. Ishdorj, A. Păun, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with extended rules: universality and languages. *Natural Computing*, 7, 2 (2008), 147–166.
4. R. Galli, A. Gritti, L. Bonfanti, A.L. Vescovi: Neural stem cells: an overview. *Circulation Research*, 92 (2003), 598–608.
5. M.R. Garey, D.S. Johnson: *Computers and Intractability. A Guide to the Theory of* **NP***-completeness*. W.H. Freeman and Company, San Francisco, 1979.
6. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2–3 (2006), 279–308.
7. T.-O. Ishdorj, A. Leporati: Uniform solutions to `SAT` and `3-SAT` by spiking neural P systems with pre-computed resources. *Natural Computing*, 7, 4 (2008), 519–534.
8. A. Leporati, M.A. Gutiérrez-Naranjo: Solving `Subset Sum` by spiking neural P systems with pre-computed resources. *Fundamenta Informaticae*, 87, 1 (2008), 61–77.
9. A. Leporati, G. Mauri, C. Zandron, Gh. Păun, M.J. Pérez-Jiménez: Uniform solutions to `SAT` and `Subset Sum` by spiking neural P systems. *Natural Computing*, online version (DOI: 10.1007/s11047-008-9091-y).
10. A. Leporati, C. Zandron, C. Ferretti, G. Mauri: Solving numerical **NP**-complete problem with spiking neural P systems. *Lecture Notes in Computer Sci.*, 4860 (2007), 336–352.
11. A. Leporati, C. Zandron, C. Ferretti, G. Mauri: On the computational power of spiking neural P systems. *International Journal of Unconventional Computing*, 2007, in press.
12. A. Păun, Gh. Păun: Small universal spiking neural P systems. *BioSystems*, 90, 1 (2007), 48–60.
13. Gh. Păun: *Membrane Computing. An Introduction*. Springer-Verlag, Berlin, 2002.
14. The P systems Website: `http://ppage.psystems.eu`.