Spiking Neural P Systems with Structural Plasticity

Francis George C. Cabarle[1], Henry N. Adorna[1], Mario J. Pérez-Jiménez[2], Tao Song[3]

[1]Algorithms & Complexity Lab, Department of Computer Science
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines;
[2]Department of Computer Science and AI
University of Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain;
[3]School of Automation
Huazhong University of Science and Technology
Wuhan 430074, Hubei, China
email: fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph, marper@us.es, taosong@hust.edu.cn

Abstract. Spiking neural P (SNP) systems are a class of parallel, distributed, and nondeterministic computing models inspired by the spiking of biological neurons. In this work, the biological feature known as structural plasticity is introduced in the framework of SNP systems. Structural plasticity refers to synapse creation and deletion, thus changing the synapse graph. The "programming" therefore of a brain-like model, the SNP system with structural plasticity (SNPSP system), is based on how neurons connect to each other. SNPSP systems are also a partial answer to an open question on SNP systems with dynamism only for synapses. For both the accepting and generative modes, we prove that SNPSP systems are universal. Modifying SNPSP systems semantics, we introduce the spike saving mode and prove that universality is maintained. In saving mode however, a deadlock state can arise, and we prove that reaching such a state is undecidable. Lastly, we provide one technique in order to use structural plasticity to solve a hard problem: a constant time, nondeterministic, and semi-uniform solution to the **NP**-complete problem `Subset Sum`.

Key words: Membrane computing, Spiking neural P systems, Structural plasticity, Computational universality, Deadlock, Undecidability, Subset sum

1 Introduction

Membrane computing is a branch of natural computing, aiming to abstract and obtain ideas (e.g. data structures, control operations, models) from the structure

and functioning of biological cells [18]. The idea of spiking in biological neurons have been introduced in the framework of membrane computing as spiking neural P systems (SNP systems in short) [9]. In spiking neurons and in SNP systems, the indistinct signals known as *spikes* do not encode the information. Instead, information is derived from the time difference between two spikes, or the number of spikes sent (received) during the computation. Time therefore is an information support in spiking neurons, and not simply a background for performing computations.

An SNP system can be thought of as a network of *spike processors*, processing spike objects and sending them to other neurons. Essentially, SNP systems can be represented by directed graphs where nodes are neurons (often drawn as ovals) and edges between neurons are synapses. Spikes (represented as a multiset of the symbol a) are sent from one neuron to another using their synapses. In SNP system literatures, many biologically inspired features have been introduced for computing use, producing many SNP system variants: see e.g. [5][7][14][16][17][20][22][24][25][27] and references therein.

In this work we are interested in the biological feature known as *neural plasticity* which is concerned with synapse modifications. Related to this feature are works in SNP systems with some forms of neural plasticity: Hebbian SNP (HSNP) systems [7] and SNP systems with neuron division and budding [16][27]. In HSNP systems, given two neurons σ_i and σ_j, and a synapse (i, j) between them, if spikes from neuron σ_i arrive repeatedly and shortly before neuron σ_j sends its own spikes, the synapse weight (strength) of (i, j) increases. If however the spikes from neuron σ_i arrive after the spikes of σ_j are sent, synapse weight of (i, j) decreases. HSNP systems were introduced for possible machine learning use in the framework of SNP systems.

In [16][27], other than spiking rules (rules that allow a neuron to send spike to other neurons), two new rules are introduced: neuron division and neuron budding rules. Both new rules involve creation of novel synapses (*synaptogenesis*) due to creation of novel neurons (*neurogenesis*). The initial synapse graph of the system is thus changed due to the application of the new rules, creating exponential workspace (in terms of neurons) in linear time. The `SAT` problem was then efficiently solved in [16][27] but by using the exponential workspace created.

The particular type of neural plasticity we introduce in the framework of SNP systems in this work is *structural plasticity*. Structural plasticity is concerned with any change in connectivity between neurons, with two mechanisms: (1) synaptogenesis and synapse deletion, (2) synaptic rewiring [2]. Unlike Hebbian (also known as *functional*) plasticity which concerns itself with only two neurons (for synapse strength modification), synaptic rewiring involves at least three neurons: if we have three neurons $\sigma_i, \sigma_j, \sigma_k$ and only one synapse (i, j), in order to create a synapse (i, k) then synapse (i, j) must be deleted first[1]. Synaptogenesis is present during neuron division and budding due to neurogenesis, while synapse

[1] This is inspired by *synaptic homeostasis* in biological neurons, where total synapse number in the system is left unchanged [2].

rewiring is not included, and synapse deletion is implied. In [16][27] a linear number of neurons forms the initial synapse graph of the SNP system. From the initial synapse graph, an exponential number of neurons can be created after some linear amount of time.

Biological neurons in adult human brains can reach more than billions in numbers, and each neuron can wire to thousands of other neurons. This phenomenon is another biological motivation in this work. In this work we can have a collection of (possibly exponential number of) neurons. The initial synapse graph can still be composed of a linear number of neurons wired or connected using synapses. However, we are only concerned with the creation and deletion of synapses over this collection for computing use. The synapse graph will then change: it is possible the system will connect an exponential number of neurons together at certain time steps (due to synapse creation) and make use of additional workspace; at other time steps the system can connect a linear or polynomial number of neurons (due to synapse deletion) and "release" additional workspace (in terms of neurons or synapses) from the system which are no longer needed. The introduction of spiking neural P systems with structural plasticity (SNPSP systems for short) is also a partial response to the open problem **D** in [22] where "dynamism" only for synapses is to be considered.

Furthermore, the standard SNP system originally from [9] included neurons with spiking rules that can have complex regular expressions, delays (in applying rules), and forgetting rules (rules that remove spikes from the system). A series of papers which proved computational universality (or simply universality, if there is no confusion) while simplifying the regular expressions of rules, removing delays or forgetting rules followed, e.g. [6][8], with the most recent being [15]. In [8] for example, SNP systems can be universal with the following being true: rules are without delays; without using forgetting rules; regular expressions are simple. In [6] it was shown that universality is achieved with more restrictions: without delays and forgetting rules; without delays and simple regular expressions. In order to maintain universality while further simplifying the system, we look for using other biological motivations.

In [17], universality in SNP systems was achieved with the following features: rules are without delays, and all neurons only have exactly one and the same simple rule. The way to control or "program" the SNP systems in [17] was using additional structures from neuroscience called astrocytes. Astrocytes introduce nondeterminism in the system, allowing the system to have simple neurons (that is, neurons in [17] contain only the rule $a \to a$).

In this work we use the biological feature of structural plasticity to achieve universality with the following restrictions: (a) only a "small" number of neurons have plasticity rules (details to follow shortly), (b) neurons without plasticity rules are simple (they only contain the rule $a \to a$), and (c) without the use of delays and forgetting rules. We do not include additional neuroscience structures other than neurons and their synapses. The introduction of the structural plasticity feature, in order to "program" the system, allows the system to maintain universality even with restrictions (a) to (c). Note that the idea of programming

how a network of neurons connect in order to perform tasks is an old one in computer science, see for example [26].

We then modify the semantics of SNPSP systems, introducing a spike saving mode. For SNPSP systems operating in such a mode, we prove that their computing power is not diminished. However, an interesting property called a deadlock state, can occur in an SNPSP system in saving mode. We then prove that reaching such a state in saving mode is undecidable for an arbitrary SNPSP system. Lastly, we provide a constant time solution to the numerical **NP**-complete problem `Subset Sum`, using only nondeterminism in synapses (using plasticity rules), and without still using forgetting rules and delays. The solution is semi-uniform, and not surprisingly, can require an exponential number of neurons. Our preliminary solution provides one possibility of using synaptic plasticity for solving hard problems.

This work is organized in the usual way as follows: preliminaries used in the rest of this work are given in Section 2; Section 3 introduces the syntax and semantics of SNPSP systems, and an example is given in Section 4; universality of SNPSP systems is given in Section 5; a modification of the semantics of SNPSP systems, the spike saving mode, is introduced in Section 6, including the deadlock property and its undecidability; a solution to `Subset Sum` is given in Section 7; a brief summary followed by final discussions and further research interests are given in Section 8.

2 Preliminaries

It is assumed that the readers are familiar with the basics of membrane computing (a good introduction is [19] with recent results and information in the P systems webpage[2] and a recent handbook [23]) and formal language theory (available in many monographs). We only briefly mention notions and notations which will be useful throughout the paper.

We denote the set of natural (counting) numbers as $\mathbb{N} = \{1, 2, \ldots\}$. Let V be an alphabet, V^* is the set of all finite strings over V with respect to concatenation and the identity element λ (the empty string). The set of all non-empty strings over V is denoted as V^+ so $V^+ = V^* - \{\lambda\}$. We call V a singleton if $V = \{a\}$ and simply write a^* and a^+ instead of $\{a\}^*$ and $\{a\}^+$. If a is a symbol in V, we write $a^0 = \lambda$. A regular expression over an alphabet V is constructed starting from λ and the symbols of V using the operations union, concatenation, and $+$. Specifically, (i) λ and each $a \in V$ are regular expressions, (ii) if E_1 and E_2 are regular expressions over V then $(E_1 \cup E_2)$, E_1E_2, and E_1^+ are regular expressions over V, and (iii) nothing else is a regular expression over V. With each 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) $L(E_1 \cup E_2) = L(E_1) \cup L(E_2)$, $L(E_1E_2) = L(E_1)L(E_2)$, and $L(E_1^+) = L(E_1)^+$, for all regular expressions E_1, E_2 over V. Unnecessary parentheses are omitted when writing regular expressions, and $E^+ \cup \{\lambda\}$ is written as E^*.

[2] `http://ppage.psystems.eu/`

By NRE we denote the family of Turing computable sets of numbers, that is, NRE is the family of length sets of recursively enumerable languages recognized by Turing machines. In proving computational universality, we use the notion of register machines. A register machine is a construct $M = (m, I, l_0, l_h, R)$, where m is the number of registers, I is the set of instruction labels, l_0 is the start label, l_h is the halt label, and R is the set of instructions. Every label $l_i \in I$ uniquely labels only one instruction in R. Register machine instructions have the following forms:

- $l_i : (\texttt{ADD}(r), l_j, l_k)$, where the value in register r is increased by 1 then non-deterministically go to either instruction label l_j or l_k;
- $l_i : (\texttt{SUB}(r), l_j, l_k)$, where if the value in register r is nonzero, then subtract 1 from it and go to instruction label l_j, otherwise go to instruction label l_k;
- $l_h : \texttt{HALT}$, the halt instruction.

Given a register machine M, we say M computes or generates a number n as follows: M starts with all its registers empty. The register machine then applies its instructions starting with the instruction labeled l_0. Without loss of generality, we assume that l_0 labels an `ADD` instruction, and that the content of the output register is never decremented, only added to during computation, i.e. no `SUB` instruction is applied to it. If M reaches the halt instruction l_h, then the number n stored during this time in the first register is said to be computed by M. We denote the set of all numbers computed by M as $N(M)$. It was proven that register machines compute all sets of number computed by a Turing machine, therefore characterizing NRE [13].

Register machines can also work in an accepting mode. A number n is stored in the first register of register machine M, with all other registers being empty. If the computation of M starting with this initial configuration halts, then n is said to be accepted or computed by M. In the accepting mode and even with M being deterministic, i.e. given an `ADD` instruction $l_i : (\texttt{ADD}(r), l_j, l_k)$ with $l_j = l_k$ written simply as $l_i : (\texttt{ADD}(r), l_j)$, register machine M can still characterize NRE.

As a convention in membrane computing, when comparing the power of two number generating or accepting devices D_1 and D_2, the number zero is ignored, i.e. $N(D_1) = N(D_2)$ if and only if $N(D_1) - \{0\} = N(D_2) - \{0\}$. This convention corresponds to the common practice in language and automata theory to ignore the empty string.

3 Spiking Neural P Systems with Structural Plasticity

In this section we introduce the variant of SNP systems with structural plasticity. The reader is invited to consult with the original SNP systems paper in [9] for initial motivations and preliminary results.

Formally, a spiking neural P system with structural plasticity (SNPSP system) of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, in, out)$, where:

- $O = \{a\}$ is the singleton alphabet (a is called spike)
- $\sigma_1, \ldots, \sigma_m$ are neurons of the form $(n_i, R_i), 1 \leq i \leq m$, with $n_i \geq 0$ indicating the initial number of spikes in σ_i, and R_i is a finite rule set of σ_i with the following forms:
  1. Spiking rule: $E/a^c \rightarrow a$, where E is a regular expression over O, with $c \geq 1$;
  2. Plasticity rule: $E/a^c \rightarrow \alpha k(i, N_j)$, where $c \geq 1$, $\alpha \in \{+, -, \pm, \mp\}$, $k \geq 1$, $1 \leq j \leq |R_i|$, and $N_j \subseteq \{1, \ldots, m\}$
- $syn \subseteq \{1, \ldots, m\} \times \{1, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, are synapses between neurons;
- $in, out \in \{1, \ldots, m\}$ indicate the input and output neurons, respectively.

Given neuron σ_i (we can also say neuron i or simply σ_i if there is no confusion) we denote the set of neuron labels which has σ_i as their presynaptic neuron as $pres(i)$, i.e. $pres(i) = \{j|(i, j) \in syn\}$. Similarly, we denote the set of neuron labels which has σ_i as their postsynaptic neuron as $pos(i) = \{j|(j, i) \in syn\}$.

Spiking rule semantics in SNPSP systems are similar with SNP systems in [9]. However, we do not use forgetting rules and spiking rules with delays in this work. Spiking rules are applied as follows: If neuron σ_i contains b spikes and $a^b \in L(E), b \geq c$, then a rule $E/a^c \rightarrow a \in R_i$ can be applied. Applying such a rule means consuming c spikes from σ_i, thus only $b - c$ spikes remain in σ_i. Neuron i sends one spike to every neuron in $pres(i)$. If a rule $E/a^c \rightarrow a$ has $E = a^c$, we simply write this as $a^c \rightarrow a$.

Plasticity rules are applied as follows. If at time t we have that σ_i has $b \geq c$ spikes and $a^b \in L(E)$, a rule $E/a^c \rightarrow \alpha k(i, N) \in R_i$ can be applied. The set N is a collection of neurons to which σ_i can connect to (synapse creation) or disconnect from (synapse deletion) using the applied plasticity rule. The rule consumes c spikes and performs one of the following, depending on α:

If $\alpha = +$ and $N - pres(i) = \emptyset$, or if $\alpha = -$ and $pres(i) = \emptyset$, then there is nothing more to do, i.e. c spikes are consumed but no synapse is created or removed. For $\alpha = +$: If $|N - pres(i)| \leq k$, deterministically create a synapse to every $\sigma_l, l \in N_j - pres(i)$. If however $|N - pres(i)| > k$, then nondeterministically select k neurons in $N - pres(i)$, and create one synapse to each selected neuron.

For $\alpha = -$: If $|pres(i)| \leq k$, deterministically delete all synapses in $pres(i)$. If however $|pres(i)| > k$, then nondeterministically select k neurons in $pres(i)$, and delete each synapse to the selected neurons.

If $\alpha \in \{\pm, \mp\}$: create (respectively, delete) synapses at time t and then delete (respectively, create) synapses at time $t + 1$. Only the priority of application of synapse creation or deletion is changed, but the application is similar to $\alpha \in \{+, -\}$. The neuron is always open from time t until $t + 1$, i.e. the neuron can continue receiving spikes. However, the neuron can only apply another rule at time $t + 2$.

An important note is that for σ_i applying a rule with $\alpha \in \{+, \pm, \mp\}$, creating a synapse always involves an *embedded* sending of one spike when σ_i connects to a neuron. This single spike is sent at the time the synapse creation is applied. Whenever σ_i *attaches* to σ_j using a synapse during synapse creation, we have σ_i immediately transferring one spike to σ_j.

If two rules with regular expressions E_1 and E_2 can be applied at the same time, that is, $L(E_1) \cap L(E_2) \neq \emptyset$, then only one of them is nondeterministically chosen and applied. All neurons therefore apply at most one rule in one time step (locally sequential) but all neurons that can apply a rule must do so (globally parallel). Note that the application of rules in neurons are synchronized, that is, a global clock is assumed.

A system state or configuration of an SNPSP system is based on (a) distribution of spikes in neurons, and (b) neuron connections based on the synapse graph syn. We can represent (a) as $\langle s_1, \ldots, s_m \rangle$ where $s_i, 1 \leq i \leq m$, is the number of spikes contained in σ_i. For (b) we can derive $pres(i)$ and $pos(i)$ from syn, for a given σ_i. The initial configuration therefore is represented as $\langle n_1, \ldots, n_m \rangle$, with the possibility of a disconnected graph, i.e. $syn = \emptyset$. A computation is defined as a sequence of configuration transitions from an initial configuration. A computation halts if the system reaches a halting configuration, that is, a configuration where no rules can be applied and all neurons are open. Whether a computation is halting or not, we associate natural numbers $1 \leq t_1 < t_2 < \ldots$ corresponding to the time instances when the neuron out sends a spike out to (or when in receives a spike from) the system.

A result of a computation can be defined in several ways in SNP systems literature, but in this work we use the following as in [9]: we only consider the first two time instances t_1 and t_2 that σ_{out} spikes. Their difference, i.e. the number $t_2 - t_1$, is said to be computed by Π. We denote the set of all numbers computed in this manner by Π as $N_2(\Pi)$. The subscript indicates that we only consider the time difference between the first two spikes of σ_{out}. Also note that 0 cannot be computed as mentioned in section 2.

In $N_2(\Pi)$ the neuron in is ignored, and we refer to this as the generative mode. SNPSP systems in the accepting mode ignore σ_{out} and work as follows: The system begins with an initial configuration, and exactly two spikes are introduced in the system (using σ_{in}) at times t_1 and t_2. The number $t_2 - t_1$ is accepted by Π if the computation halts. The set of numbers accepted by Π is denoted as $N_{acc}(\Pi)$. The families of all sets of $N_\alpha(\Pi)$, with $\alpha \in \{2, acc\}$, is denoted as $N_\alpha SNPSP$.

4 An Example of an SNPSP System

In this section, we provide an example to illustrate the definition and semantics of an SNPSP system. Consider an SNPSP system Π_{ex} shown in figure 1. Neurons $2, out = 3, A_1$, and A_2 contain only the rule $a \to a$ and we omit this from writing. In the initial configuration, at some time t, is where only σ_1 has two spikes and σ_3 has only one spike. Neuron 1 is the only neuron with plasticity rules, where $\alpha \in \{+, -\}$, and we have $syn = \{(2, A_1), (2, A_2), (A_1, 1), (A_2, 1)\}$.

Π_{ex} operates as follows: The application of the two rules in σ_1 are deterministic. Nondeterminism in Π_{ex} exists only in selecting which synapses to create to or delete from.[3] Neuron 3 has $n_3 = 1$, so it sends a spike out to the environ-

[3] We refer to this later as *synapse level nondeterminism*

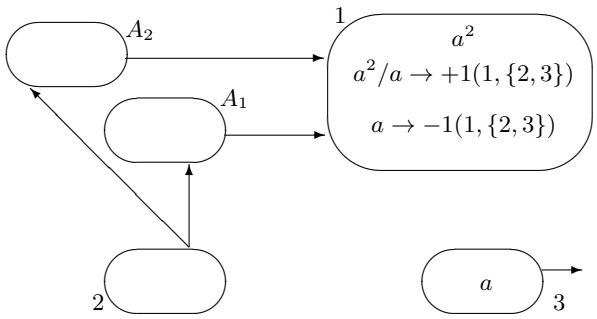

Fig. 1. An SNPSP system Π_{ex}.

ment, and we label this event as time t_1. Since $n_1 = 2$, the rule with $\alpha = +$ is applied, so σ_1 nondeterministically selects one (since $k = 1$) of σ_2 or σ_3 to create a synapse to.

If the synapse $(1, 3)$ is created, then σ_1 sends one spike to σ_3 also at time t_1. The rule with $\alpha = +$ consumes one spike, so σ_1 now has one spike. At time t_2 the rule with $\alpha = -$ is applied, and synapse $(1, 3)$ is deleted. The spike that σ_3 received from σ_1 is sent to the environment at time t_2 also. Two spikes are sent out to the environment by Π_{ex} before it halts. The computed number in this case is then $t_2 - t_1 = 1$. The computation of $\{1\}$ by Π_{ex} is shown in table 1, where (!) means that the output neuron spikes to the environment.

time	σ_1	σ_2	σ_3	σ_{A_1}	σ_{A_2}	syn
t	2	0	1	0	0	syn
t_1	1	0	1(!)	0	0	$syn \cup \{(1,3)\}$
t_2	0	0	0(!)	0	0	syn

Table 1. Computation of Π_{ex} computing $\{1\}$.

If however the synapse $(1, 2)$ is created, σ_1 sends a spike to σ_2 at time t_1 during synapse creation. Neuron 2 then sends one spike each to auxiliary neurons A_1 and A_2 at time t_2. Also at time t_2 is when the rule with $\alpha = -$ is applied, and $(1, 2)$ is deleted. A_1 and A_2 send one spike each to σ_1, so that σ_1 has two spikes again at time t_3, as in the initial configuration. As long as σ_1 creates synapse $(1, 2)$ instead of $(1, 3)$ then Π_{ex} keeps receiving two spikes in a loop.

Notice that if at some time $m > 1$ the synapse $(1, 2)$ is created, it will take σ_1 three time steps for the possibility of creating $(1, 3)$ again. Once $(1, 3)$ is created, one more step is required for σ_3 to spike to the environment for the second and final time. The computation of $\{4\}$ for example, is shown in table 2. Therefore, from the operation of Π_{ex} we then have $N_2(\Pi_{ex}) = \{1, 4, 7, 10, \ldots\} = \{3m + 1 | m \geq 0\}$. Also notice that for Π_{ex}, its two plasticity rules in σ_1 can be replaced by a single plasticity rule: $a^2 \to \pm 1(1, \{2, 3\})$.

time	σ_1	σ_2	σ_3	σ_{A_1}	σ_{A_2}	syn
t	2	0	1	0	0	syn
t_1	1	1	0(!)	0	0	$syn \cup \{(1,2)\}$
t_2	0	0	0	1	1	syn
t_3	2	0	0	0	0	syn
t_4	1	0	1	0	0	$syn \cup \{(1,3)\}$
t_5	0	0	0(!)	0	0	syn

Table 2. Computation of Π_{ex} computing $\{4\}$.

5 Universality of SNPSP Systems

In this section we show that SNPSP systems are computationally universal, i.e. they characterize NRE, both in the accepting and generative modes. In SNPSP systems there are two types of nondeterminism: (1) in selecting which rule to apply in a neuron (rule level nondeterminism), and (2) in selecting which synapses to create to or delete from (synapse level nondeterminism). However, we prove that synapse level nondeterminism is sufficient for universality. Therefore, the SNPSP systems we construct in this work do not involve rule level nondeterminism.

We start with SNPSP systems working in the generative mode, followed by the accepting mode. As in Π_{ex} and unless mentioned otherwise, we omit from writing the rules for simple neurons with $a \to a$ as the only rule. A network of these simple neurons (first referred to in [17]) with rules of a simple form, together with fewer neurons that have plasticity rules, are computationally powerful: they can perform tasks that Turing machines can perform.

SNPSP systems working in the generative mode

Theorem 1 $NRE = N_2SNPSP$.

Proof. To prove theorem 1 we only need to prove $NRE \subseteq N_2SNPSP$ by simulating a register machine M in generative mode with an SNPSP system. The converse, i.e. $N_2SNPSP \subseteq NRE$, is straightforward or the Turing-Church thesis can be invoked. Without loss of generality we may assume for register machine $M = (m, I, l_0, l_h, R)$ to have: (a) all registers except register 1 are empty at halting; (b) the output register is never decremented during any computation; and (c) the initial instruction of M is an `ADD` instruction.

The SNPSP system Π simulating M has three modules: the ADD, SUB and FIN modules shown in figures 2, 3, and 4 respectively. All three modules consist of simple neurons and some neurons that have plasticity rules. The ADD and SUB modules simulate the `ADD` and `SUB` instructions of M, respectively. The FIN module is used to output the computation of Π in the generative mode mentioned in section 3.

Given a register r of M we have an associated neuron σ_r in Π. In any computation, if r stores the value n, then σ_r will contain $2n$ spikes. For every label

l_i in M we have σ_{l_i} in Π. The initial configuration of Π is where only σ_{l_0} has one spike, indicating that instruction l_0 in M is to be executed. Simulating a rule $l_i : (\texttt{OP}(r) : l_j, l_k)$ in M means σ_{l_i} has one spike and is activated. Then, depending on the operation $\texttt{OP} \in \{\texttt{ADD}, \texttt{SUB}\}$, an operation is performed on σ_r. Either σ_{l_j} or σ_{l_k} receives a spike and becomes activated. When M executes l_h (the label for the halting instruction of M), Π completely simulates the computation of M. Π then activates σ_{l_h}, and σ_{out} sends two spikes to the environment with a time difference equal to the stored value in register 1.

Module ADD shown in figure 2 simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$: The initial instruction of M is labeled l_0, and it is an ADD instruction. Assume that instruction $l_i : (\texttt{ADD}(r) : l_j, l_k)$ is to be simulated at time t. Therefore σ_{l_i} contains one spike while all other neurons are empty except neurons associated with registers. Neuron l_i uses its rule $a \to a$ and sends one spike each to neurons l_i^1 and l_i^2. At time $t+1$ both neurons l_i^1 and l_i^2 send one spike to σ_r so that σ_r receives two spikes corresponding to the value one in register r of M. Also at time $t+1$ is when neuron l_i^2 sends one spike to σ_p, which is the only neuron in module ADD with plasticity rules.

Once σ_p receives a spike, σ_p consumes one spike and at time $t+2$, σ_p nondeterministically creates one synapse (since $\alpha = \pm, k = 1$) to either neuron l_j or l_k. Also at time $t+2$, σ_p sends a spike to either neuron l_j or l_k; If synapse (p, l_j) was created then σ_p removes (p, l_j) afterwards, otherwise σ_p removes (p, l_k), at time $t+3$.

At time $t+3$ either neuron l_j or l_k is activated, i.e. receives one spike from σ_p, and can therefore perform the associated instruction. From firing neuron l_i the module ADD increments the spikes contained in σ_r by two and nondeterministically activates either neuron l_j or l_k. We therefore correctly simulate an ADD instruction.

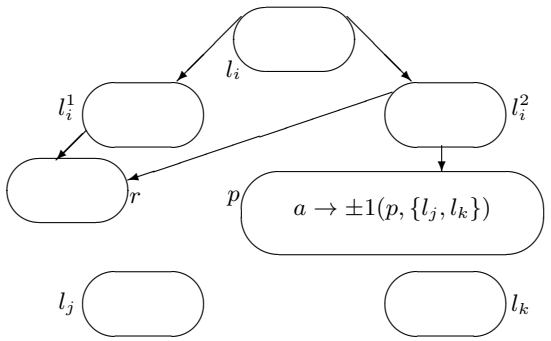

Fig. 2. Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j, l_k)$.

Module SUB shown in figure 3 simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$: We simulate a SUB instruction with the SUB module as follows: Initially only σ_{l_i} has one spike and all other neurons are empty except those associated with registers. Let t be

the time when neuron l_i fires, using its rule $a \to a$ to send one spike to σ_r and $\sigma_{l_i^1}$. As with the SUB instruction in M, the two cases are when register r stores a nonzero value (execute instruction l_j) or when r stores zero (execute l_k). For the moment let us set $N_j = \{l_i^2\}$ and $N_k = \{l_i^3\}$ so that $|N_j| = |N_k| = 1$. The case for a general N_j and N_k will be explained shortly.

If σ_r contains a nonzero number of spikes at time t, then it contains $2n$ (even numbered) spikes corresponding to n stored in register r. At time $t+1$ neuron r has $2n+1$ (odd numbered) spikes, the additional one spike will come from neuron l_i at time t. The rule with regular expression $a(a^2)^+$ is enabled at $t+1$ since σ_r has an odd number of spikes. At time $t+1$ the rule consumes three spikes and deterministically creates (for the moment) $|N_j| = 1$ synapse (r, l_i^2), and sends one spike to $\sigma_{l_i^2}$. At time $t+2$, synapse (r, l_i^2) is deleted (since $\alpha = \pm$). At time $t+2$ therefore, neuron l_i^2 contains two spikes: one each from neurons l_i^1 and r. We then have neuron l_i^2 creating at time $t+2$ a synapse, as well as sending one spike, to neuron l_j. Neuron l_j is therefore activated.

Removing three spikes from $2n+1$ spikes in σ_r makes the number of spikes contained in σ_r even again. In particular, after rule application, σ_r will contain $2(n-1)$ spikes corresponding to $n-1$ in r. Module SUB correctly simulates decreasing the value stored in r by one if r stored a nonzero value, and then executing l_j.

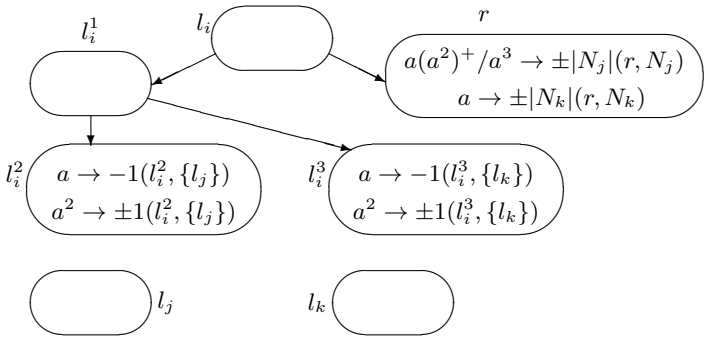

Fig. 3. Module SUB simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$.

If σ_r contains no spikes at time t corresponding to a stored value of zero in r, then σ_r contains one spike at time $t+1$ and applies the rule $a \to \pm 1(r, \{l_i^3\})$. This rule consumes one spike, reducing the spikes contained in σ_r back to zero. The rule also creates synapse (r, l_i^3) and sends one spike to neuron l_i^3 at time $t+1$. At $t+2$, synapse (r, l_i^3) is deleted. Neuron l_i^3 contains 2 spikes at $t+1$, one each from neurons r and l_i^1. At $t+2$, neuron l_i^3 creates synapse (l_i^3, l_k) and sends one spike to σ_{l_k}, activating σ_{l_k}. Module SUB correctly simulated maintaining the zero stored in r if r initially stored zero, and then executing l_k.

For the general case where there are at least two SUB instructions operating on register r, this means we have at least two SUB modules operating

on the same σ_r: to simulate $l_x : (\texttt{SUB}(r) : l_{x_1}, l_{x_2})$ and $l_y : (\texttt{SUB}(r) : l_{y_1}, l_{y_2})$, we have to be certain that simulating l_x (l_y, respectively) only activates either neuron l_{x_1} or l_{x_2} (l_{y_1} or l_{y_2} respectively). The set N_j (N_k, respectively) is defined as $N_j = \{j | j$ is the second element of the triple in a SUB instruction$\}$ ($N_k = \{k | k$ is the third element of the triple in a SUB instruction$\}$, respectively).

The possibility of at least two SUB modules interfering with each other is removed since each l_i^2 and l_i^3, $i \in \{x, y\}$, is only activated when a spike from the corresponding l_i^1 is received. Only the specific SUB instruction to be simulated has its corresponding neuron l_i^2 (otherwise l_i^3, depending on the stored value in r) in a SUB module receive two spikes. The other SUB modules only have their neuron l_i^2 (otherwise, l_i^3) receive only one spike. Either neuron l_i^2 or l_i^3 therefore activate the correct neuron: whenever $\alpha = \pm$ (when containing two spikes), or $\alpha = -$ (when containing one spike). If $\alpha = -$ then only synapse removal is performed. Therefore, the SUB instruction is correctly simulated by the SUB module.

Module FIN shown in figure 4 once l_h is reached in M: We assume at time t that the computation in M halts, so that instruction with label l_h is reached. Also at time t we have σ_1 containing $2n$ spikes corresponding to the value n stored in register 1 of M. Neuron l_h sends one spike each to σ_1 and σ_{out}. At time $t+1$ neuron *out* sends the first of two spikes that it will send to the environment before computation halts. Also at time $t+1$ we have σ_1 containing $2n+1$ spikes. Neuron 1 continuously applies its rule $a^3(a^2)^+/a^2 \to +1(1, \{out\})$ if σ_1 initially contained four or more spikes. The rule performs the following every time it is applied: two spikes are consumed and the synapse $(1, out)$ is deleted (since $\alpha = -, k = 1$).

Notice that synapse $(1, out)$ does not exist, so the two spikes consumed are simply removed from the system and no synapse is actually removed. Also notice that applying this rule leaves σ_1 with an odd number of spikes afterwards. The value of k in the rule can actually be any positive integer but in this case it is simply set to one. Once the number of spikes in σ_1 is reduced to three, the rule $a^3 \to \pm 1(1, \{out\})$ is applied. This rule is applied after n applications of the previous rule. At time $t+n$ the rule removes three spikes and leaves no spikes in σ_1, creates a synapse and sends a spike to σ_{out}, and deletes the synapse. Neuron *out* receives one spike from σ_1 and spikes for the second and final time to the environment at time $t+n+1$. The time difference between the first and second spiking of σ_{out} is $(t+n+1)-(t+1) = n$, exactly the number stored in register 1 of M when the computation of M halted.

From the description of the operations of modules ADD, SUB, and FIN, Π clearly simulates the computation of M. Therefore we have $N_2(\Pi) = N(M)$ and this completes the proof. □

SNPSP systems working in the accepting mode

SNPSP systems in the accepting mode ignore the *out* neuron and use an *in* neuron to take in exactly two spikes. The time difference between the two input spikes is the number computed by the system, if the computation halts. Recall

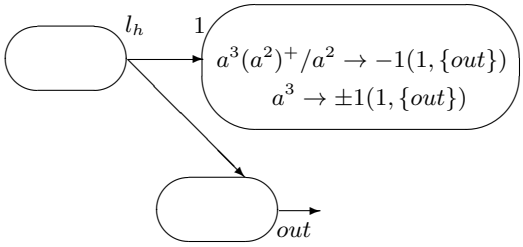

Fig. 4. Module FIN.

that a register machine M is computationally universal even for the deterministic accepting case. The resulting SNPSP system simulating M is therefore simpler compared to the system simulating the generative case as in theorem 1.

Theorem 2 *$NRE = N_{acc}SNPSP$.*

Proof. The proof for theorem 2 is a consequence of the proof of theorem 1. Given a deterministic register machine $M = (m, I, l_0, l_h, R)$ we construct an SNPSP system Π as in the proof of theorem 1. Since M is deterministic in this case, we modify Π so that addition is deterministic and we use an INPUT module instead of a FIN module. As with the proof of theorem 1, the modules will contain simple neurons having only $a \to a$ as their rule, and some neurons with plasticity rules.

The new module INPUT is shown in figure 5 and functions as follows: All neurons are initially empty and we let the time that the first spike enters the module be t. The second and final spike input will arrive at time $t + n$ so that $(t + n) - t = n$ is the value stored in register 1 of M. Neuron in uses its rule $a \to a$ at time $t + 1$ to send one spike each to neurons A_1, A_2, and A_3.

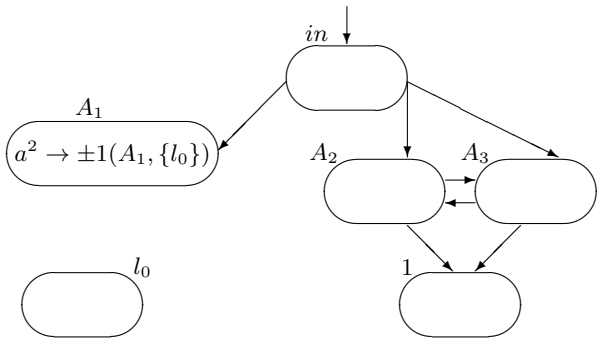

Fig. 5. Module INPUT.

Neurons A_2 and A_3 exchange spikes every step from $t + 2$ until $t + n + 1$. During every exchange, neurons A_2 and A_3 increase the spikes inside σ_1 by two.

Once the second and the final spikes arrive, σ_{in} spikes at $t+n+1$ so that A_2 and A_3 now each have two spikes, and therefore cannot apply their only rule $a \to a$. At time $t+n+1$ neuron A_1 collects two spikes so it can apply its plasticity rule to activate neuron l_0. The activation of neuron l_0 corresponds to the start of the simulation of instruction l_0 in M. Note that from time $t+2$ until $t+n+1$, σ_1 collects a total of $2n$ spikes, corresponding to the value n stored in register 1.

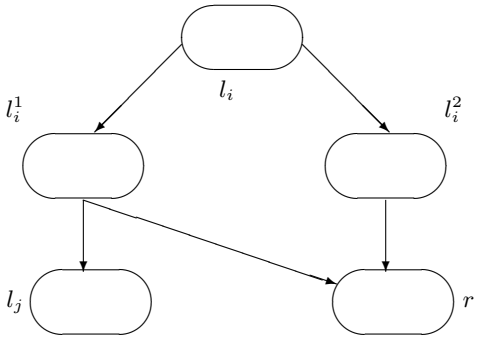

Fig. 6. Module ADD simulating $l_i : (\texttt{ADD}(r) : l_j)$.

Since for the accepting case M can be computationally universal even with a deterministic ADD instruction, we have the deterministic ADD module in figure 6. The functioning of the deterministic ADD module is clear and is simpler than the nondeterministic ADD module. The SUB module remains the same as in the proof of theorem 1. Module FIN is not used, while σ_{l_h} remains in the system except that $pres(l_h) = \emptyset$.

Once σ_{l_h} receives a spike indicating that the computation of M has halted, the neuron applies its rule $a \to a$ but does not send its spike to any neuron. Therefore, the computation in Π halts only if the computation in M halts, and we have $N_{acc}(\Pi) = N(M)$ completing the proof. □

6 Spike Saving Mode Universality and Deadlock

In this section we consider a modification of how plasticity rules are applied: if there is nothing more to do in terms of synapse creation or deletion, we do not consume a spike. We prove that given this modification, SNPSP systems are still universal. Additionally, an interesting property, the deadlock, can occur.

Let us consider again the definition and semantics of plasticity rules in section 3: If σ_i contains b spikes and a rule $E/a^c \to \alpha k : (i, N_j)$, if $a^b \in L(E)$, then the plasticity rule can be applied. Applying the rule means consuming c spikes, and then performing plasticity operations depending on the value of α and k. If however $pres(i) = \emptyset$, there is nothing more to do for $\alpha = -$: This is because there is no synapse to delete, since σ_i is not a presynaptic neuron of any other

neuron. Such a case is seen with the module FIN in figure 4. Note that c spikes are still consumed, even if we later realize that there is nothing more to do. Such a case also exists for $\alpha = +$.

An interesting modification of this plasticity rule mode of operation is as follows: during the "nothing more to do" case, i.e. whenever $\alpha \in \{+, -\}$ and $N_j - pres(i) = \emptyset$ or $pres(i) = \emptyset$, we *save spikes* so that they are not consumed. As an analogy, if we pay a certain money or currency[4] to perform a task, then this saving mode means our money is returned if we discover that the task has been performed already. In contrast, the nonsaving mode in section 3 is analogous to paying money to perform a task, regardless if the task is actually performed or not.

A natural inquiry from modifying SNPSP systems from nonsaving to saving mode is: are SNPSP systems in the saving mode still universal? The answer is affirmative, and requires a modification of the FIN module in figure 4. The modification is the basis for the next result. We denote by $SNPSP_s$ the family of sets computed by an SNPSP system in saving mode.

Theorem 3 $NRE = N_2SNPSP_s$

Proof. The SUB and FIN modules make use of the nonsaving mode in theorem 1. We only need to modify the two modules, while the ADD module remains intact. We denote the modified SUB and FIN modules as SUB′ and FIN′, respectively. Module FIN′ is shown in figure 7. For FIN′, we add one new neuron σ_p and a new synapse $(1, p)$. Neuron p does not have any rules and $pres(p) = \emptyset$. We also remove the plasticity rule $a^3(a^2)^+/a^2 \to -1(1, \{out\})$ and replace it with a spiking rule $a^3(a^2)^+/a^2 \to a$.

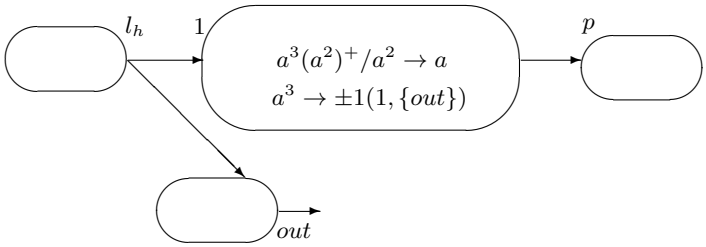

Fig. 7. Module FIN′.

Note that we still do not use forgetting rules. The purpose of the new rule in σ_1 is to reduce the number of spikes each time step by 2 in σ_1 until only 3 remain. Once there are 3 spikes in σ_1, then the remaining rule $a^3 \to \pm 1(1, \{out\})$ is applied. The new σ_p functions as a trap or repository for the spikes removed from σ_1. We then have FIN′ functioning as it should be: delaying the second spike of σ_{out} for n steps, as required, and as was performed by FIN also.

[4] Or in the case of neurons, the quanta of energy which is the spike.

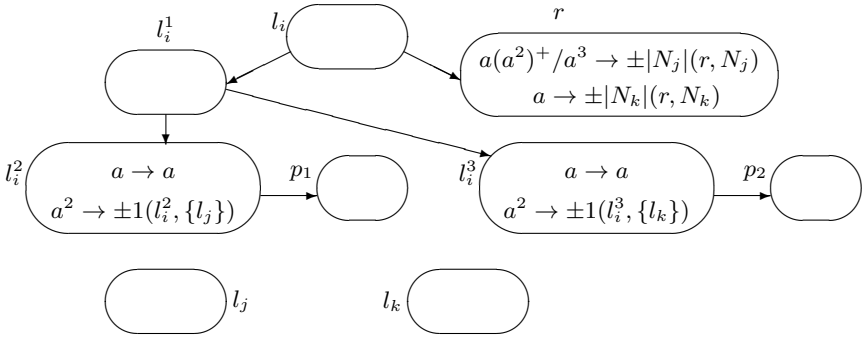

Fig. 8. Module SUB′ simulating $l_i : (\texttt{SUB}(r) : l_j, l_k)$.

Module SUB′ is shown in figure 8. For SUB′, we add trap neurons σ_{p_1} and σ_{p_2} for every neuron l_i^2 and l_i^3 in a SUB module, respectively. We also add for each SUB module, the synapses (l_i^2, p_1) and (l_i^3, p_2). We then replace the rule $a \to -1(l_i^2, \{l_j\})$ in neuron l_i^2 (rule $a \to -1(l_i^3, \{l_k\})$ in neuron l_i^3, respectively) with a spiking rule $a \to a$. Similar to the trap neuron in FIN′, the trap neurons in SUB′ receive the unwanted spikes from neuron l_i^2 (or l_i^3) so that interference does not occur. Therefore, SUB′ functions similarly to SUB, and as required. □

Corollary 1 $NRE = N_{acc}SNPSP_s$

Corollary 1 follows from theorem 3, since in the accepting mode from theorem 2, the saving or nonsaving mode is of no consequence. An interesting property that arises when using the saving mode is the existence of a *deadlock*. During the saving mode, deadlock occurs when a plasticity rule is applied because the regular expression is satisfied, but no actual work is performed by the neuron. The result is a state where no further computation can continue, and the system is stuck or trapped in the current configuration, even if a rule can always be applied. More formally, a deadlock state (or configuration) occurs if a σ_i can apply at least one plasticity rule and we have the following sequence of events during the same time step: a^c spikes are consumed, however the applied plasticity rule either has $\alpha = +$ and $N - pres(i) = \emptyset$, or $\alpha = -$ and $pres(i) = \emptyset$; since there is nothing more to do in such a case, a^c spikes are returned to σ_i.

Lemma 1 *A deadlock can exist in an arbitrary SNPSP system in saving mode having at least two neurons, with* $\alpha \in \{+, -\}$.

Proof. Let us first consider when $\alpha = +$, and we define a $\Pi_{(+)}$ as follows.

$$\Pi_{(+)} = (\{a\}, \sigma_i, \sigma_j, syn_{(+)})$$

where $\sigma_i = (1, R_i)$, $R_i = \{a \to +1(i, \{j\})\}$, $\sigma_j = (0, \emptyset)$, and $syn_{(+)} = \{(i, j)\}$. Figure 9 provides an illustration for $\Pi_{(+)}$. We have that $\Pi_{(+)}$ has exactly one

synapse between σ_i and σ_j, and exactly one rule (a plasticity rule) in σ_i. Once the plasticity rule is applied, the saving mode requires us to consume one spike first. The plasticity rule has $\alpha = +$ and $k = 1 = |pres(i)|$. Therefore we must create a new synapse from σ_i to σ_j. However, the synapse (i, j) already exists and we return the consumed spike, as dictated by the saving mode. Therefore $\Pi_{(+)}$ is stuck in the current and only configuration: the regular expression of the only rule is satisfied, however, we cannot perform further computations.

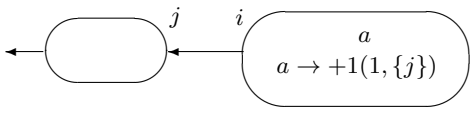

Fig. 9. The SNPSP system $\Pi_{(+)}$ with a deadlock from the proof of lemma 1.

Now let us consider a similar SNPSP system $\Pi_{(-)}$ defined as follows.

$$\Pi_{(-)} = (\{a\}, \sigma_i, \sigma_j, syn_{(-)})$$

where $\sigma_i = (1, R_i)$, $R_i = \{a \to -1(i, \{j\})\}$, $\sigma_j = (0, \emptyset)$, and $syn_{(-)} = \emptyset$. In a similar way, the one and only rule of $\Pi_{(-)}$ is always applied, since the regular expression is satisfied. However, there is no synapse between σ_i and σ_j, since $syn_{(-)} = pres(i) = \emptyset$, so there is nothing to do. The saving mode dictates that the consumed spike must be returned and the system remains in a deadlock. □

An important problem (e.g. see [10]) is whether it is decidable to have an arbitrary SNPSP system in saving mode arrives at a state of deadlock or not. Not surprisingly, the answer to this problem is a negative one, as given by the next result.

Theorem 4 *It is undecidable whether an arbitrary SNPSP system Π' in saving mode, with at least two neurons, reaches a deadlock.*

Proof. We consider an arbitrary recursively enumerable set of natural numbers which we denote D. We can have a register machine, an SNP system, or an SNPSP system generate the elements of D. For simplicity, we then choose to construct an SNPSP system Π similar to section 5 such that $D = N_2(\Pi)$. The system Π produces two spikes (using its output neuron) if and only if D is non-empty, and this task is undecidable. Furthermore, we construct an SNPSP system Π' using Π as a submodule. We refer to figure 10 for a graphical representation for Π' instead of a formal definition.

If D is nonempty (this means σ_{out} of Π spikes twice), then σ_i receives two spikes and it can apply its rule. However, from lemma 1 we know that the application of the rule in σ_i results in a state of deadlock. The problem of realizing if D is nonempty is undecidable. Therefore, a deadlock is reached if and only if the set D is nonempty, which happens to be undecidable. □

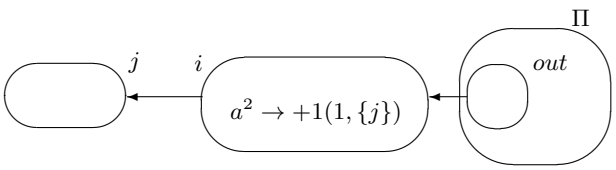

Fig. 10. The SNPSP system Π' for the proof in theorem 4.

7 Solving `Subset Sum`

In this section we provide a solution to the numerical **NP**-complete problem `Subset sum`. The solution provided here is nondeterministic and semi-uniform. By uniform solution we mean that we construct only one solution (in this case, an SNPSP system) for any instance of the problem. The construction of a semi-uniform solution depends on specific instances of a problem, and semi-uniform if the construction is done in polynomial time. We refer to [11][12][16][23][27] for details of uniformity in solutions.

In [11] for example, the `Subset sum` problem was also solved in a semi-uniform way using SNP systems with extended rules. Additionally, their solution used modules that have neurons operating in deterministic or nondeterministic ways, and applying rules in sequential or maximally parallel manner. A follow-up and improved solution was then given in [12]. For the solution we present here, unless otherwise mentioned, the SNPSP systems work in nonsaving mode as in section 3. The problem `Subset sum` can be defined as follows:
Problem: `Subset Sum`.

- Instance: S, and a (multi)set $V = \{v_1, v_2, \ldots, v_n\}$, with $S, v_i \in \mathbb{N}$ and $1 \leq i \leq n$;
- Question: Is there a sub(multi)set $B \subseteq V$ such that $\sum_{b \in B} b = S$?

We can now have the following last result.

Theorem 5 *There exists a semi-uniform SNPSP system solving any instance* $W = (S, V = \{v_1, \ldots, v_n\})$ *of* `Subset Sum`*, with the following parameters:*

- *synapse level nondeterminism only,*
- *without forgetting rules and rules with delays,*
- *computes in constant time.*

Proof. First, we formally describe the SNPSP system $\Pi_{\mathtt{ss}}(W)$ that solves instance W of the problem.

$\Pi_{\mathtt{ss}} = (\{a\}, \sigma_1, \ldots, \sigma_n, \sigma_{1^{(1)}}, \ldots, \sigma_{1^{(2v_1)}}, \ldots, \sigma_{n^{(1)}}, \ldots, \sigma_{n^{(2v_n)}}, \sigma_{out}, syn, out)$ where:

(**1**) For $1 \leq i \leq n, \sigma_i = (v_i, R_i)$, with $R_i = \{a^{v_i} \to +v_i(v_i, \{i^{(1)}, \ldots, i^{(2v_i)}\})\}$;

(**2**) For $1 \leq j \leq v_n, \sigma_{i^{(j)}} = (0, \{a \to a\})$;
(**3**) For $v_n + 1 \leq k \leq 2v_n, \sigma_{i^{(k)}} = (0, \emptyset)$;

and

- $\sigma_{out} = (0, \{a^S \to a\})$;
- $syn = \{(1^{(1)}, out), \ldots, (1^{(v_n)}, out), \ldots, (n^{(1)}, out), \ldots, (n^{(v_n)}, out)\}$;

We refer to figure 11 for a graphical representation of $\Pi_{\texttt{ss}}$. In figure 11, for simple neurons that only have the rule $a \to a$, we omit the writing of the rule as in section 5. Note however that in the definition of $\Pi_{\texttt{ss}}$, some neurons actually do not have rules in them. The distinction between these empty neurons and the simple neurons will be explained shortly. Next, we provide the functioning of $\Pi_{\texttt{ss}}$, which we divide into three stages for clarity:

Stage 1: This stage consists of operations performed by neurons σ_1 to σ_n, where σ_i initially contains v_i spikes. These are the neurons from (**1**). Each σ_i consumes all the initial v_i spikes, to create v_i number of synapses. Notice that each σ_i has $\alpha = +$ and $k = v_i$, but can nondeterministically select among $2v_i$ number of neurons to create a synapse to. Once every σ_i has finished selecting and creating synapses to v_i number of neurons, we move to the next stage.

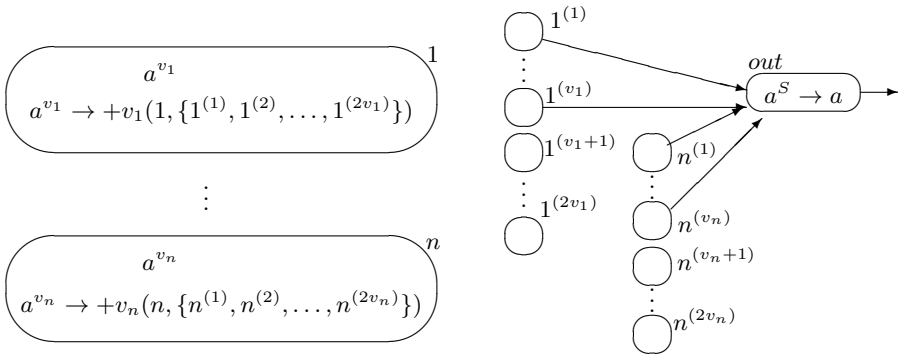

Fig. 11. The SNPSP system $\Pi_{\texttt{ss}}$ solving `Subset Sum`.

Stage 2: This stage consists of operations performed by neurons $\sigma_{1^{(1)}}$ up to $\sigma_{1^{(2v_1)}}$ associated with σ_1 (from stage 1), up to the neurons $\sigma_{n^{(1)}}$ up to $\sigma_{n^{(2v_n)}}$ associated with σ_n (from stage 1). These are the neurons from (**2**) and (**3**). Note in the definition that given the $2v_i$ neurons in this stage associated with σ_i in stage 1, the first half of these neurons (i.e. $\sigma_{i^{(1)}}$ up to $\sigma_{i^{(v_i)}}$) are simple neurons with the rule $a \to a$. The second half (i.e. $\sigma_{i^{(v_i+1)}}$ up to $\sigma_{i^{(2v_i)}}$) are empty neurons, without initial spikes or rules. Also note that neurons in the first half each have a synapse to σ_{out}, while the second half do not have synapses, i.e. they are all traps, where each of their presynaptic sets is empty.

Since the nondeterminism in stage 1 is synapse level, in some computations some (or all) simple neurons will each receive a spike, while in other computations some (or all) empty neurons each receive a spike. For the computations that send a spike to the empty neurons, these neurons function as trap or repository neurons. Consequently, the spikes in these trap neurons can also be removed using forgetting or plasticity rules. The computations that have some (or all) simple neurons in this stage then send one spike each to σ_{out}, leading us to the next stage.

Stage 3: This final stage checks the existence of an S number of spikes from stage 2. If there are S spikes in σ_{out} this means the answer to this instance of the `Subset Sum` is affirmative, and one spike is sent to the environment. This spike, affirming the answer to the problem instance, is sent out to the environment 3 time steps since stage 1. If however the spikes sent to σ_{out} from stage 2 do not add up to S, this means the answer to the problem instance is negative. Therefore, at time step 3 no spike is sent to the environment. □

8 Final Remarks

Before we provide the final discussion, we briefly recall the contributions of this work: Section 3 introduced the structural plasticity feature in the SNP systems framework, providing a partial answer to an open problem in [22] about dynamism only for synapses. The "programming" of the system is dependent on the way neurons (dis)connect to each other using synapses.

In section 5 and 6 we proved that SNPSP systems in the generative and accepting cases are universal, for both the saving and nonsaving modes. The universality results hold even if we impose the following restrictions on SNPSP systems: only 5 among the 17 total neurons for all modules[5] use plasticity rules, only synapse level nondeterminism exist, neurons without plasticity rules are simple (having only the rule $a \to a$), and without using forgetting rules or delays. Additionally, a state known as deadlock can arise during saving mode. Reaching such a state for an arbitrary SNPSP system in saving mode is undecidable. In section 7 we provided a nondeterministic and semi-uniform solution to `Subset Sum`, computing in 3 time steps.

The deadlock state does not seem to exist in nonsaving mode. Does this mean that saving mode is "better" than nonsaving mode? Both modes have the same expressiveness, but perhaps the (non)existence of a deadlock is useful for more practical purposes, e.g. modeling or analysis of systems. For example, reaching a deadlock can be interpreted as reaching an unwanted state or fault in a network or system. This is an interesting theoretical and practical question. Deadlocks and other state types, as applied to modeling and analysis, have an extensive body of research in many graphical formalisms such as Petri nets [10]. Several works have related Petri nets, SNP systems, and other P systems. See

[5] ADD, SUB, and FIN module neurons in the saving and generative case, since the accepting case requires a lesser number of neurons with plasticity rules.

for example [3] and references therein. A state of undecidable deadlock can also occur in systems in [20].

The solution provided in section 7 is a preliminary one: it is of course interesting to ask other ways to encode instances of other **NP**-complete problems for (non)deterministic synapse selection. What about (non)deterministic, uniform solutions? In computer science we work to have efficient solutions, seeking minimal parameters. However, and as we mentioned in section 1, biology is usually not space efficient: there are billions of cells in the human body, and billions of neurons each with thousands of synapses in our brains. So the solution in section 7 may not immediately appeal to our search for minimal parameters, but it seems biologically appealing. Depending on the instance to be solved, the number of neurons for example (and possibly the spikes and synapses) in $\Pi_{\mathtt{ss}}$ can be exponential with respect to the instance size.

Other parameter modifications of computational interest abound: what about parallel synapse creation-deletion? The semantics in section 3 for $\alpha \in \{\pm, \mp\}$ is sequential, requiring 2 time steps to perform such rules with such values for α. We only list here a few other complexity parameters that can be used: the size of *syn* during a computation, e.g. the number of synapses created (deleted), or a bound on the size of *syn* (related to synaptic homeostasis); in the universality proofs we had $\alpha \in \{+, -, \pm\}$, so can we have universality with α having at most 2 values only? Normal forms for SNPSP systems are interesting as well.

SNPSP systems and the proofs in section 5 are reminiscent of the more generalized extended SNP systems and their universality proofs in [1]. Investigating the relation of ESNP and SNPSP systems seems interesting. More recently, SNP systems with rules on synapses were shown to be universal [25]. In these systems, neurons are only spike repositories, and the spike processing (including nondeterminism) are done in the synapses. Investigating theoretical or practical usefulness of such systems, together with structural plasticty, is also interesting.

Lastly, SNPSP systems can be equipped with one input and one output neurons, providing us with a transducer. Such transducers can be used to compute (in)finite strings as in [21]. Furthermore, SNPSP systems with multiple inputs or outputs are also interesting, for computing strings or vectors of numbers as in [1] again.

Acknowledgements

F.G.C. Cabarle is supported by a scholarship from the DOST-ERDT Philippines. T. Song is supported by the China Postdoctoral Science Foundation Project (No. 2014M550389). H.N. Adorna is funded by a DOST-ERDT research grant and the Semirara Mining Corporation professorial chair of the College of Engineering, UP Diliman. M.J. Pérez-Jiménez acknowledges the support of the Project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, co-financed by FEDER funds. The authors are also grateful to three anonymous referees for their useful comments.

References

1. Alhazov, A., Freund, R., Oswald, M., Slavkovik, M.: Extended Spiking Neural P Systems. H.J. Hoogeboom et al. (Eds.): WMC 7, LNCS 4361, pp. 123-134 (2006)
2. Butz, M., Wörgötter, F., van Ooyen, A.: Activity-dependent structural plasticity. Brain Research Reviews 60, pp. 287-305 (2009)
3. Cabarle, F.G.C., Adorna, H..: On Structures and Behaviors of Spiking Neural P Systems and Petri Nets. E. Csuhaj-Varjú et al. (Eds.): CMC 2012, LNCS 7762, pp. 145-160 (2013)
4. Cabarle, F.G.C, Adorna, H., Ibo, N.: Spiking neural P systems with structural plasticity. ACMC2013, Chengdu, China, 4-7 November (2013)
5. Cavaliere, M., Ibarra, O., Păun, G., Egecioglu, O., Ionescu, M., Woodworth, S.: Asynchronous spiking neural P systems. Theoretical Computer Science, 410, pp. 2352-2364 (2009)
6. García-Amau, M., Pérez, D., Rodríguez-Patón, A., Sosík, P.: Spiking Neural P Systems: Stronger Normal Forms. International journal of unconventional computing, vol 5(5), pp. 411-425 (2009)
7. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Hebbian Learning from Spiking Neural P Systems View. D. Corne et al. (Eds.) WMC9, LNCS 5391, pp. 217-230 (2009)
8. Ibarra, O., Păun, A., Păun, G., Rodríguez-Patón, A., Sosík, P., Woodworth, S.: Normal forms for spiking neural P systems. Theoretical Computer Science vol 372(2-3) pp. 196-217 (2007)
9. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundamenta Informaticae, vol. 71(2,3), pp. 279-308 (2006)
10. Iordache, M., Antsaklis, P.: Deadlock and Liveness Properties of Petri Nets. Supervisory Control of Concurrent Systems: A Petri Net Structural Approach. pp. 125-151. Birkhäuser Boston (2006)
11. Leporati, A., Zandron, C., Ferretti, C., Mauri, G.: Solving Numerical NP-Complete Problems with Spiking Neural P Systems. Eleftherakis et al. (Eds.): WMC8 2007, LNCS 4860, pp. 336-352 (2007)
12. Leporati, G., Mauri, G., Zandron, C., Păun, G., Pérez-Jiménez, M.: Uniform solutions to SAT and Subset Sum by spiking neural P systems. Natural Computing vol. 8, pp. 681-702 (2009)
13. Minsky, M.: Computation: Finite and infinite machines. Englewood Cliffs, NJ: Prentice Hall, (1967)
14. Pan, L., Păun, G.: Spiking neural P systems with anti-spikes. J. of Computers, Communication, & Control. vol IV(3), pp. 273-282 (2009)
15. Pan, L., Păun, G.: Spiking Neural P Systems: An improved normal form. Theoretical Computer Science vol 411(6), pp. 906-918 (2010)
16. Pan, L., Păun, Gh., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Science China Information Sciences. vol. 54(8) pp. 1596-1607 (2011)
17. Pan, L., Wang, J., Hoogeboom, J.H.: Spiking Neural P Systems with Astrocytes. Neural Computation 24, pp. 805-825 (2012)
18. Păun, Gh.: Computing with membranes. Journal of Computer and System Science, vol. 61(1), pp. 108-143 (1999)
19. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002)
20. Păun, Gh.: Spiking Neural P Systems with Astrocyte-like Control. J. Universal Computer Science. vol. 13(11), pp. 1707-1721 (2007)

21. Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G.: Computing Morphisms by Spiking Neural P Systems. International Journal of Foundations of Computer Science. vol. 8(6) pp. 1371-1382 (2007)
22. Păun, Gh., Pérez-Jiménez, M.J.: Spiking Neural P Systems. Recent Results, Research Topics. A. Condon et al. (eds.), Algorithmic Bioprocesses, Springer (2009)
23. Păun, Gh., Rozenberg, G., Salomaa, A. (Eds) The Oxford Handbook of Membrane Computing, OUP (2010)
24. Song, T., Pan, L., Păun, G.: Asynchronous spiking neural P systems with local synchronization. Information Sciences, 219, pp.197-207 (2013)
25. Song, T., Pan, L., Păun, G.: Spiking neural P systems with rules on synapses. Theoretical Computer Science vol. 529(10) pp. 82-95 (2014)
26. Turing, A. Intelligent Machinery. in Essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artificial Intelligence, and Artificial Life: Plus The Secrets of Enigma. Copeland, B. (Ed.). OUP (2004)
27. Wang, J., Hoogeboom, H.J., Pan, L.: Spiking Neural P Systems with Neuron Division. M. Gheorghe et al. (Eds.): CMC 2010, LNCS 6501, pp. 361-376 (2010)