Journal of Membrane Computing (2019) 1:145–160
https://doi.org/10.1007/s41965-019-00020-3

REGULAR PAPER

Matrix representation and simulation algorithm of spiking neural P systems with structural plasticity

Zechariah B. Jimenez[1] · Francis George C. Cabarle[1,2] · Ren Tristan A. de la Cruz[1] · Kelvin C. Buño[1] · Henry N. Adorna[1] · Nestine Hope S. Hernandez[1] · Xiangxiang Zeng[3]

Received: 26 January 2019 / Accepted: 25 July 2019 / Published online: 19 August 2019
© Springer Nature Singapore Pte Ltd. 2019

Abstract

In this paper, we create a matrix representation for spiking neural P systems with structural plasticity (SNPSP, for short), taking inspiration from existing algorithms and representations for related variants. Using our matrix representation, we provide a simulation algorithm for SNPSP systems. We prove that the algorithm correctly simulates an SNPSP system: our representation and algorithm are able to capture the syntax and semantics of SNPSP systems, e.g. plasticity rules, dynamism in the synapse set. Analyses of the time and space complexity of our algorithm show that its implementation can benefit using parallel computers. Our representation and simulation algorithm can be useful when implementing SNPSP systems and related variants with a dynamic topology, in software or hardware.

Keywords Spiking neural P systems · Structural plasticity · Matrix representation · Membrane computing

1 Introduction

In the realm of computer science, models are used to describe the workings of various systems and how they are said to be "computing." These models possess various features that depict how inputs are changed and manipulated to achieve the desired outputs. Present-day computers are based on some variants of the Turing machine and other "classical" models and, therefore, carry their characteristic advantages and disadvantages; speed, Turing-completeness, and space capacity are just some of the important properties that describe the abilities of these models. Small changes in any of these could spell the difference between being able to compute or solve a problem, and otherwise. It is thus imperative to come up with new models that overcome the obstacles that impede such classical models and, thus, solve problems in a more efficient way. While many new models still have no commercially available physical realization, simulation on modern computers is enough to highlight the abilities of such models until feasible prototypes are created.

A good example of such new models that are still in the simulation stage is those in the field of natural computing, specifically in membrane computing. These models are based on natural phenomena, like the transfer of chemicals within cells and throughout cell systems. The advantage of these models over classical models is their characteristic parallelism, even over small space constraints. The parallelism could then be used to solve NP-complete and other hard problems in an efficient manner.

The specific model from membrane computing of interest in this work is the spiking neural P system (in short, SNP system) model [11, 20]. Much works exist about the computing power and efficiency of SNP systems: they are known to be Turing universal when computing (sets of) numbers or strings as in Refs. [7, 8, 11, 20]; they are computationally efficient, able to solve hard problems as in Refs. [12–14]. Various ideas from neuroscience and maths provide inspirations to create variants of SNP systems, such as neuron division and budding in Ref. [17], astrocytes in Refs. [18, 21], anti-spikes in Ref. [16], weights in synapses in Refs. [19, 25], rules on synapses in Ref. [23], synapse schedules in Ref. [1], and coloured spikes in Ref. [24]. Some real-world

✉ Francis George C. Cabarle
fccabarle@up.edu.ph

1 Department of Computer Science, University of the Philippines Diliman, 1101 Quezon City, Philippines

2 Shenzhen Research Institute of Xiamen University, Xiamen University, Shenzhen 518000, Guangdong, China

3 School of Information Science and Engineering, Hunan University, 410082 Changsha, China

applications have also been solved using SNP systems and variants, as in Refs. [22, 26, 28].

This paper focuses on a variant of SNP systems known as SNP systems with structural plasticity or SNPSP systems [4]. The matrix representation and simulation algorithm we present here for SNPSP systems draw inspiration from those mentioned in Refs. [4–6, 9, 27]. A preliminary version of the matrix representation in this paper is in Ref. [10].

As elaborated in Refs. [2, 6, 27], the benefits of a matrix representation compared to other representations are due to the increased parallelism when performing linear algebra operations. This increased parallelism when simulating computations can benefit sequential (e.g. CPU) simulators but more so using parallel (e.g. GPU) simulators. More benefits using a matrix representation and other parallel computing techniques are recently given in Refs. [5, 9, 15]. In Ref. [15], variants of SN P systems that have a dynamic topology, i.e. adding or removing neurons, synapses, or both, is compared with respect to the recent technologies of GPUs. It is then noted in Ref. [15] that for such GPUs, the more efficient way to perform dynamism in the topology is the plasticity found in SNPSP systems.

The main contributions of this paper are the matrix representation and simulation algorithm for SNPSP systems. The simulation algorithm is broken into smaller algorithms for clarity. We analyse the time and space complexity of our algorithm, which is useful for future implementations in computers. Our representation and algorithm are able to capture the syntax and semantics of SNPSP systems, and we show this using proofs of correctness. Features specific to SNPSP systems, such as creating or deleting synapses among neurons, are correctly simulated. This paper is structured as follows: in Sect. 2, the preliminaries for this work are introduced; in Sects. 3 and 4, our matrix representation and notations are provided, respectively; the representation and notations are used in our simulation algorithm in Sect. 5; an example of our algorithms is in Sect. 6; lastly, Sect. 7 provides closing remarks and research directions. Detailed proofs of our theorems are given in Appendix 1.

2 Preliminaries

For this work, a specific variant of the SNP system would be in focus, namely the spiking neural P system with structural plasticity (SNPSP). Here, forgetting rules are replaced by plasticity rules, thus marking the characteristic difference between the two models. Plasticity rules allow for the creation, deletion, and rewiring of synapses by their respective source neurons. More formally, it is also given in Ref. [4]:

Definition 1 (*SNPSP system*) A *spiking neural P system with structural plasticity* (SNPSP system, for short) of degree $m \geq 1$ is a construct of the form

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

where

1. $O = \{a\}$ is the singleton alphabet (a is called spike)
2. $\sigma_1, \dots, \sigma_m$ are pairs $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, called neurons, where $n_i \geq 0$ and $n_i \in \mathbb{N} \cup \{0\}$ represents the initial spikes in σ_i and R_i is a finite set of rules of σ_i with the following forms:
   (a) Spiking rule: $E/a^c \rightarrow a$, where E is a regular expression over O, with $c \geq 1$;
   (b) 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 $N_j \subseteq \{1, \dots, m\}$.
3. $syn \subseteq \{1, \dots, m\} \times \{1, \dots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq m$, are synapses between neurons;
4. $out \in \{1, \dots, m\}$ indicates the output neuron.

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 $pos(i) = \{j | (j, i) \in syn\}$.

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 σ_l, $l \in N_j - pres(i)$. If, however, $|N - pres(i)| > k$, then non-deterministically 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 non-deterministically 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

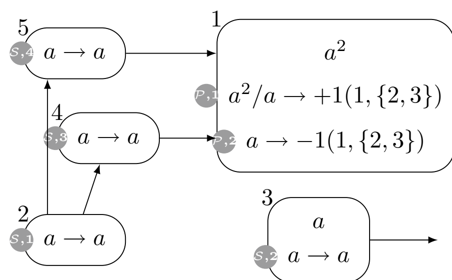

Fig. 1 An SNPSP system Π_{ex}

$t+1$, 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 is 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 $\langle s_1, \dots, s_m \rangle$ where s_i, $1 \le i \le m$, 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 $\langle n_1, \dots, n_m \rangle$, 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 $1 \le t_1 < t_2 < \cdots$ 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 [11]: 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.

As an illustration, consider an SNPSP system Π_{ex} shown in Fig. 1 from [4]. Each rule in Fig. 1 is labeled as $\mathcal{R}, i$ to mean the ith rule of type $\mathcal{R}$. Thus, a rule with label $\mathcal{P}, 3$ is known as rule $r_{\mathcal{P},3}$ (more on this in Sect. 3). Neurons 2, $out = 3$, 4, and 5 contain only the rule $a \to a$, and we omit this from writing. In the initial configuration, at time $t_0 = 0$, 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 we have $syn = \{(2,4), (2,5), (4,1), (5,1)\}$.

As detailed in Ref. [4], we have Π_{ex} computing the set $\{1, 4, 7, 10, \dots\} = \{3m+1 | m \ge 0\}$. In Table 1, the output of Π_{ex} is $t_2 - t_1 = 1$ if neuron σ_1 creates synapse (1, 3), where (!) means that the output neuron σ_3 fires a spike to the environment, and t_2 and t_1 are the second and first time σ_3 fires, respectively. In Table 2, the output of Π_{ex} is 4 if σ_1 creates synapse (1, 2) instead of (1, 3).

Table 1 Computation of Π_{ex} for $\{1\}$

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

Table 2 Computation of Π_{ex} for $\{4\}$

Time	σ_1	σ_2	σ_3	σ_{A_1}	σ_{A_2}	*syn*
0	2	0	1	0	0	*syn*
$t_1 = 1$	1	1	0 (!)	0	0	$syn \cup \{(1,2)\}$
2	0	0	0	1	1	*syn*
3	2	0	0	0	0	*syn*
4	1	0	1	0	0	$syn \cup \{(1,3)\}$
$t_2 = 5$	0	0	0 (!)	0	0	*syn*

3 Matrix representation of SNPSP

To illustrate how SNPSP systems can be represented as specified below, we use Π_{ex} in Fig. 1. Using the formal definition, the system can thus be expressed as $\Pi_{ex} = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_5, syn, 3)$. The neurons are (from σ_1 to σ_5) $(2, \{r_{\mathcal{P},1}, r_{\mathcal{P},2}\})$, $(0, \{r_{\mathcal{S},1}\})$, $(1, \{r_{\mathcal{S},2}\})$, $(0, \{r_{\mathcal{S},3}\})$, and $(0, \{r_{\mathcal{S},4}\})$; the synapses are defined as $syn = \{(2,4), (2,5), (5,1), (4,1)\}, 3)$; and finally the rules are $r_{\mathcal{P},1} = a^2/a \rightarrow +1(1, \{2,3\})$, $r_{\mathcal{P},2} = a \rightarrow -1(1, \{2,3\})$, and $r_{\mathcal{S},i} = a \rightarrow a, \forall i \in \{1,2,3,4\}$.

For SNPSP systems, a neuron is said to be defined by its spike count and the set of rules associated with it. With this, we define the spike count vector and the rule source matrix.

Definition 2 (*Spike count vector*) Let Π be an SNPSP system with m neurons. In a computation, for any $k \in \mathbb{N}$, the vector $C^{(k)} = [c_1^{(k)}, c_2^{(k)}, \dots, c_m^{(k)}]$ is called the *spike count vector* of the system at time k, where $c_i^{(k)}$ is the amount of spikes in neuron $\sigma_i, i = 1, 2, \dots, m$ at time k.

Note that a key feature of the matrix representation being defined is the separate ordering of the spiking and plasticity rules, from 1 to $r_{\mathcal{S}}$ and from 1 to $r_{\mathcal{P}}$, respectively. There are a total of $r = r_{\mathcal{S}} + r_{\mathcal{P}}$ rules.

Definition 3 (*Rule source matrix*) Let Π be an SNPSP system with m neurons. Let $r_{\mathcal{R}}$ be the number of rules of type $\mathcal{R} \in \{\mathcal{P}, \mathcal{S}\}$, where $\mathcal{P}$ and $\mathcal{S}$ correspond to plasticity and spiking rules, respectively. Let $d_{\mathcal{R}} : (\mathcal{R}, 1), \dots, (\mathcal{R}, r_{\mathcal{R}})$ be a total ordering of rules of type $\mathcal{R}$. The *rule source matrices* of the system Π, $Sr_{\mathcal{R}}$, are defined as follows:

$$Sr_{\mathcal{R}} = \begin{bmatrix} sr_{\mathcal{R},1,1} & \cdots & sr_{\mathcal{R},1,m} \\ \vdots & \ddots & \vdots \\ sr_{\mathcal{R},r_{\mathcal{R}},1} & \cdots & sr_{\mathcal{R},r_{\mathcal{R}},m} \end{bmatrix}$$

where

$$sr_{\mathcal{R},i,j} = \begin{cases} 1, & \text{if rule } r_{\mathcal{R},i} \text{ is in neuron } \sigma_j; \\ 0, & \text{otherwise.} \end{cases}$$

We also define an aggregate rule source matrix Sr to denote the combination of $Sr_{\mathcal{P}}$ and $Sr_{\mathcal{S}}$, with the rows (rules) arbitrarily ordered.

For Π_{ex}, the initial spike counts and the rule source matrices are

$$C^{(0)} = \begin{bmatrix} 2 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{1}$$

$$Sr_{\mathcal{P}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{2}$$

$$Sr_{\mathcal{S}} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Next in the definition of SNPSP systems is the set of synapses. Here, since these connections are not constant, the synapse matrix is defined to change with time. In addition to that, we define matrices that record the newly created (and deleted) synapses.

Definition 4 (*Synapse matrix*) In an SNPSP system Π with m neurons, the *synapse matrix*, $Sy^{(k)}$, at time k, is defined as follows:

$$Sy^{(k)} = \left[sy_{i,j}^{(k)} \right]_{m \times m} = \begin{bmatrix} sy_{1,1}^{(k)} & \cdots & sy_{1,m}^{(k)} \\ \vdots & \ddots & \vdots \\ sy_{m,1}^{(k)} & \cdots & sy_{m,m}^{(k)} \end{bmatrix}$$

where

$$sy_{i,j}^{(k)} = \begin{cases} 1, & \text{if there exists a synapse from neuron } \sigma_i \\ & \text{to neuron } \sigma_j \text{ at time } k; \\ 0, & \text{otherwise.} \end{cases}$$

Definition 5 (*Synapse creation [deletion] matrix*) In an SNPSP system Π with m neurons, the *synapse creation* [*deletion*] *matrix*, $Sy_+^{(k)}$ $[Sy_-^{(k)}]$, at time k, is defined as follows ($o \in \{+, -\}$):

$$Sy_o^{(k)} = \left[sy_{o,i,j}^{(k)} \right]_{m \times m} = \begin{bmatrix} sy_{o,1,1}^{(k)} & \cdots & sy_{o,1,m}^{(k)} \\ \vdots & \ddots & \vdots \\ sy_{o,m,1}^{(k)} & \cdots & sy_{o,m,m}^{(k)} \end{bmatrix}$$

where

$$sy_{o,i,j}^{(k)} = \begin{cases} 1, & \text{if a synapse from neuron } \sigma_i \text{ to neuron } \sigma_j \\ & \text{was operated on at time } k; \\ 0, & \text{otherwise.} \end{cases}$$

and the indicated operation is deletion if $o = -$ or creation if $o = +$.

We also define a *synapse change matrix* $Sy_\Delta^{(k)} = Sy_+^{(k)} - Sy_-^{(k)}$ to be the net change in the synapse matrix at time k.

Given Definitions 4 and 5, we can obtain the next synapse matrix with

$$Sy^{(k)} = Sy^{(k-1)} + Sy_{\Delta}^{(k)} = Sy^{(k-1)} + Sy_{+}^{(k)} - Sy_{-}^{(k)} \quad (4)$$

Since time $k = 0$ is the start of the computation, no synapses are supposed to have been created or deleted. We see that $Sy_{+}^{(0)}$, $Sy_{-}^{(0)}$, and $Sy_{\Delta}^{(0)}$ are thus zero. $Sy^{(0)}$, on the other hand, shows the initial set of synapses at the start, and so that of the system Π_{ex} is

$$Sy^{(0)} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (5)$$

The synapse from neuron σ_3 to the environment is not shown here, but is indicated in the declaration of Π_{ex}, since $out = 3$. This connection is constant throughout the computation and, thus, cannot be changed by either Sy_+ or Sy_-.

Definition 2 records the spikes stored in the neurons, but here we would also need to know about the spikes sent out to the environment. For this, we have

Definition 6 (*Output spike count and output spike indicator*) In an SNPSP system Π, the *output spike count* at time k is denoted by $os^{(k)}$, which is the number of spikes already sent out by the output neuron to the environment from time 0 to time k. The *output spike indicator* at time k is defined as

$$sp^{(k)} = \begin{cases} 1, & \text{if a spike was sent out to the environment at time } k; \\ 0, & \text{otherwise.} \end{cases}$$

In the computation as in Table 2, the output neuron spiked to the environment at times $t_1 = 1$ and $t_2 = 5$. Table 3 shows the values of $os^{(k)}$ and $sp^{(k)}$.

Next, we define vectors and matrices that describe the rules associated with the neurons of the system Π. First, we need to describe the regular expressions used by the rules to determine the number of spikes required for firing. For this work, we shall be limiting these regular expressions to be of the forms a^k, a^+, a^*, $a^k(a^j)^*$, and $a^k(a^j)^+$, for some positive integers j and k. In general, we describe these regular expressions to be of *linear* form—i.e. they can be described by the pattern $a^{p+qn} = a^p(a^q)^*$ for integers $p, q, n \geq 0,\ p + q \geq 1$. Table 4 illustrates this.

Thus, we can describe the regular expressions by their corresponding p and q values.

Definition 7 (*Regular expression P and Q vectors*) In an SNPSP system Π, $P_{\mathcal{S}}$ and $Q_{\mathcal{S}}$ [$P_{\mathcal{P}}$ and $Q_{\mathcal{P}}$] are the *(regular expression) P and Q vectors* of the spiking [plasticity] rules, defined as $P_{\mathcal{R}} = [p_{\mathcal{R},1}, \dots, p_{\mathcal{R},r_{\mathcal{R}}}]$ and $Q_{\mathcal{R}} = [q_{\mathcal{R},1}, \dots, p_{\mathcal{R},r_{\mathcal{R}}}]$ for $\mathcal{R} \in \{\mathcal{P}, \mathcal{S}\}$, which describe the p and q values for the regular expressions of each rule, such that

$$E_{\mathcal{R},i} = a^{\bar{p}+\bar{q}n} = a^{\bar{p}}(a^{\bar{q}})^*$$

where $\bar{p} = p_{\mathcal{R},i}$, $\bar{q} = q_{\mathcal{R},i}$, and $E_{\mathcal{R},i}$ is the regular expression of the rule $r_{\mathcal{R},i}$.

We also define aggregate P and Q vectors to denote the combination of $P_{\mathcal{P}}$ with $P_{\mathcal{S}}$, and $Q_{\mathcal{P}}$ with $Q_{\mathcal{S}}$, respectively. The elements are arbitrarily ordered.

Once we can decide if a rule can fire, we can then check which rules would fire and which would not. Note that for this work, if a rule is applicable, it must fire immediately. Rules have also been restricted to determinism per neuron, and to sequentiality. Thus, for rules r_a and r_b both in the same neuron, $r_a \neq r_b$, $L(E_a) \cap L(E_b) = \emptyset$. We then have the following definition:

Definition 8 (*Rule firing vector*) In an SNPSP system Π, the *rule firing vectors* at time k are defined as the vector

$$Fi_{\mathcal{R}}^{(k)} = \left[fi_{\mathcal{R},1}^{(k)}, \dots, fi_{\mathcal{R},r_{\mathcal{R}}}^{(k)}\right]$$

for $\mathcal{R} \in \{\mathcal{P}, \mathcal{S}\}$ ($\mathcal{P}$ for plasticity rules, $\mathcal{S}$ for spiking rules). The vectors describe which rules will be fired, as follows:

$$fi_{\mathcal{R},i}^{(k)} = \begin{cases} 1, & \text{if rule } r_{\mathcal{R},i} \text{ is fired at time } k; \\ 0, & \text{otherwise.} \end{cases}$$

We also define an aggregate rule firing vector Fi to denote the combination of $Fi_{\mathcal{P}}$ and $Fi_{\mathcal{S}}$, with the elements arbitrarily ordered.

Table 3 Output spike counts and indicators for Π_{ex} Computing {4}

Time	$os^{(k)}$	$sp^{(k)}$
0	0	0
$t_1 = 1$	1	1
2	1	0
3	1	0
4	1	0
$t_2 = 5$	2	1

Table 4 Allowed forms of regular expressions

Form	Pattern	p	q
a^*	$a^0(a^1)^*$	0	1
a^+	$a^1(a^1)^*$	1	1
a^k	$a^k(a^0)^*$	k	0
$a^k(a^j)^*$	$a^k(a^j)^*$	k	j
$a^k(a^j)^+$	$a^{k+j}(a^j)^*$	$k+j$	j

Since the regular expressions in Π_{ex} are only of the forms a and a^2, the Q vectors are just zero vectors of lengths 4 and 2 for spiking and plasticity rules, respectively. $P_{\mathcal{S}} = [1, 1, 1, 1]$ since all the spiking rules are the same, and $P_{\mathcal{P}} = [2, 1]$. Nothing fires at the start of the computation, so $Fi_{\mathcal{S}}^{(0)}$ and $Fi_{\mathcal{P}}^{(0)}$ are both zero vectors. Then, since only neurons σ_1 and σ_3 have spikes at the start, and both have applicable rules, the rule firing vectors at time 1 are $Fi_{\mathcal{S}}^{(1)} = [0, 1, 0, 0]$ and $Fi_{\mathcal{P}}^{(1)} = [1, 0]$.

Once a rule is fired, it consumes a specified number of spikes from its source neuron. Thus, we have

Definition 9 (*Spike consumption vector*) In an SNPSP system Π, the *spike consumption vectors* at time k are defined as the vector $Co_{\mathcal{R}}^{(k)} = \left[co_{\mathcal{R},1}^{(k)}, \dots, co_{\mathcal{R},r_{\mathcal{R}}}^{(k)}\right]$, for $\mathcal{R} \in \{\mathcal{P}, \mathcal{S}\}$ ($\mathcal{P}$ for plasticity rules, $\mathcal{S}$ for spiking rules). Here, $co_{\mathcal{R},i}^{(k)} = c$ is the number of spikes consumed by rule $r_{\mathcal{S},i} = E/a^c \to a^p$ if $\mathcal{R} = \mathcal{S}$, or by rule $r_{\mathcal{P},i} = E/a^c \to \alpha k(i, N)$ if $\mathcal{R} = \mathcal{P}$.

We also define an aggregate spike consumption vector Co to denote the combination of $Co_{\mathcal{P}}$ and $Co_{\mathcal{S}}$, with the elements arbitrarily ordered.

Since all the rules in Π_{ex} consume only one spike upon firing, the spike consumption vectors are $Co_{\mathcal{S}} = [1, 1, 1, 1]$ and $Co_{\mathcal{P}} = [1, 1]$.

The plasticity rules have four types of operations, namely $+$ for synapse creation, $-$ for synapse deletion, $\pm$ for successive creation and deletion in two time steps, and $\mp$ for successive deletion and creation. Given that for the latter two operations the creation and deletion occur in two consecutive time steps, we can then decide when to execute which plasticity operation using timers in a similar manner to the delays in Ref. [27]. The timers count down at every time step, execute their respective operation upon reaching 1, and then stop at 0. Just as in Ref. [10], this is further illustrated as timers of the form (*creation*, *deletion*) follows: starting at an idle state, the timer is initialized at (0, 0). A $+$ or a $-$ operation will set it to (1, 0) and (0, 1), respectively. Lastly, the $\pm$ and $\mp$ operations, having their component operations done in two consecutive time steps, set the timer to (1, 2) and (2, 1), respectively. For all of these, the timers count down at every time step up to 0.

Definition 10 (*Timer matrix*) In an SNPSP system Π, the *timer matrix* at time k is defined as the matrix

$$Ti^{(k)} = \begin{bmatrix} ti_{1,1}^{(k)} & ti_{1,2}^{(k)} \\ \vdots & \vdots \\ ti_{r_{\mathcal{P}},1}^{(k)} & ti_{r_{\mathcal{P}},2}^{(k)} \end{bmatrix}$$

where, for $o = [+, -]$:

$$ti_{i,j}^{(k)} = \begin{cases} t, & \text{if rule } r_{\mathcal{P},i} \text{ is to execute } o_j \text{ at time } k + t - 1; \\ 0, & \text{otherwise.} \end{cases}$$

We also define a *primed timer matrix*, $Ti'^{(k)}$, which is the timer after ticking (counting down) at time k, but before rules are fired at time k. Thus, $Ti^{(k)}$ is also called the *unprimed* timer matrix.

Once we find out a rule should fire at time k, we then start the timer using the following matrix:

Definition 11 (*Timer start matrix*) In an SNPSP system Π, the *timer start matrix* is defined as the matrix

$$St = \begin{bmatrix} st_{1,1} & st_{1,2} \\ st_{2,1} & st_{2,2} \\ \vdots & \vdots \\ st_{r_{\mathcal{P}},1} & st_{r_{\mathcal{P}},2} \end{bmatrix}$$

where each $st_{i,j}$ would be the value that $ti_{i,j}^{(k)}$ should be set to once rule $r_{\mathcal{P},i}$ is to fire at time k.

Since rules $r_{\mathcal{P},1}$ and $r_{\mathcal{P},2}$ would, respectively, create and delete a synapse, the timer start matrix would be

$$St = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{6}$$

Given that rule $r_{\mathcal{P},1}$ is to fire at the beginning, the timer matrix at the start would then be

$$Ti^{(0)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{7}$$

For the remaining parts of the plasticity rules as defined, we have the following:

Definition 12 (*Destination candidate matrix*) In an SNPSP system Π, the *destination candidate matrix* is defined as the matrix

$$NM = \begin{bmatrix} nm_{1,1} & \cdots & nm_{1,m} \\ \vdots & \ddots & \vdots \\ nm_{r_{\mathcal{P}},1} & \cdots & nm_{r_{\mathcal{P}},m} \end{bmatrix}$$

where

$$nm_{i,j} = \begin{cases} 1, & \text{if } i \in N_j, \text{for rule } r_{\mathcal{P},j} = E/a^c \to \alpha k(i, N_j); \\ 0, & \text{otherwise.} \end{cases}$$

Definition 13 (*Synapse count vector*) In an SNPSP system Π, the *synapse count vector* is defined as the vector

$$KV = \left[kv_1, \dots, kv_{r_{\mathcal{P}}}\right]$$

where each $kv_i = k$, for rule $r_i = E/a^c \to \alpha k(i, N_j)$.

Rules $r_{\mathcal{P},1}$ and $r_{\mathcal{P},2}$ are to both operate on one synapse to either neuron σ_2 or σ_3, and thus

$$NM = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{8}$$

$$KV = \begin{bmatrix} 1 & 1 \end{bmatrix} \tag{9}$$

Finally, for the simulations, we need to keep track of each computation step and system configuration. Thus, we have the following definitions.

Definition 14 (*Spike gain vector*) In an SNPSP system Π with m neurons, the *spike gain vector* at time k is defined as the vector $G^{(k)} = \left[g_1^{(k)}, \dots, g_m^{(k)}\right]$ where each $g_i^{(k)}$ is the number of spikes gained by the neuron σ_i in time k, from other neurons. These gains can also be segregated according to the type of the rule that caused that gain, as with $G_{\mathcal{R}}^{(k)} = \left[g_{\mathcal{R},1}^{(k)}, \dots, g_{\mathcal{R},m}^{(k)}\right]$ where $\mathcal{R} = \mathcal{P}$ for plasticity rules and $\mathcal{R} = \mathcal{S}$ for spiking rules.

Definition 15 (*Spike loss vector*) In an SNPSP system Π with m neurons, the *spike loss vector* at time k is defined as the vector $L^{(k)} = \left[l_1^{(k)}, \dots, l_m^{(k)}\right]$ where each $l_i^{(k)}$ is the number of spikes lost by the neuron σ_i in time k from spike consumption by rule firing. These losses can also be segregated according to the type of the rule that caused that loss, as with $L_{\mathcal{R}}^{(k)} = \left[l_{\mathcal{R},1}^{(k)}, \dots, l_{\mathcal{R},m}^{(k)}\right]$ where $\mathcal{R} = \mathcal{P}$ for plasticity rules and $\mathcal{R} = \mathcal{S}$ for spiking rules.

Definition 16 (*System state*) In the computations of an SNPSP system Π, the overall *system state* at time k is defined as

$$\begin{aligned} Cf^{(k)} &= \langle Rule^{(k)} | Syn^{(k)} | Conf^{(k)} \rangle \\ &= \langle Fi^{(k)}, Ti^{(k)}, os^{(k)}, sp^{(k)} | Sy_{\Delta}^{(k)} | C^{(k)}, Sy^{(k)}, Ti'^{(k)} \rangle \end{aligned}$$

where $Rule^{(k)}$ is the *rule change node*, $Syn^{(k)}$ is the *synapse change node*, and $Conf^{(k)}$ is the *system configuration node*, all for time k.

The *initial state* $Cf^{(0)}$ marks the start of a computation. A computation is only to be terminated by a *halting state* $Cf^{(t)}$, where either (1) $os^{(t)}$ has been set to 2, or (2) t has reached a certain desired maximum time step.

The next few definitions would be for representing and generating computations and would be very important in the simulation algorithms.

Definition 17 (*Computation trace*) Given an SNPSP system Π, a *computation trace* of Π is a sequence of *nodes* $\{Conf^{(0)}, Rule^{(1)}, Syn^{(1)}, Conf^{(1)}, \dots, Conf^{(t)}\}$ starting with an initial configuration node $Conf^{(0)}$ followed by triples of nodes of $(Rule^{(k)}, Syn^{(k)}, Conf^{(k)})$ representing system states. A computation trace is said to be *valid* iff the following conditions are satisfied:

- each system state $Cf^{(k)}$ (after the initial configuration) can be correctly generated or computed from the previous system state $Cf^{(k-1)}$;
- the initial system state is represented by $Rule^{(0)}$ (not in the sequence but defined to be filled with 0-values), $Syn^{(0)}$ (also not in the sequence but defined to be filled with 0-values), and $Conf^{(0)}$;
- the terminating (halting) system step is represented by the last rule change node $Rule^{(t)}$ either holds $os^{(t)} = 2$ or t has reached a maximum time step.

Definition 18 (*Computation tree*) Given an SNPSP system Π, a *computation tree/graph* for Π is a rooted graph where each path from the root (the initial configuration node $Conf^{(0)}$) to a leaf (halting configuration node $Conf^{(t)}$) is a computation trace for Π. A computation tree is said to be *correct* if the set of all paths from the root to the leaves is equal to the set of *valid* computation traces.

Note that we would allow loops in generating a computation tree, thus making it more appropriate to call them computation *graphs*.

4 Notations and conventions

Here, we would describe the conventions and notations in writing matrices. In this work, given a matrix Mat, we would refer to the rth row and the cth column as Mat_r and $Mat_{(c)}$, respectively. Note that these are both row vectors. For a matrix with subscripts and superscripts, as with $Mat_x^{(k)}$, we would then have $Mat_{x,r}^{(k)}$ and $Mat_{x,(c)}^{(k)}$. Since scalars here would usually be written in lowercase, a particular element of the matrix (say, the (i, j)th) would be denoted by $mat_{i,j}$. For example, for a matrix

$$Mat = \begin{bmatrix} 1 & 2 & 3 \\ 1 & 2 & 3 \\ 1 & 2 & 3 \end{bmatrix}$$

$Mat_1 = [1, 2, 3]$ is the first row, $Mat_{(2)} = [2, 2, 2]$ is the second column (as a row vector), and $mat_{2,3} = 3$ is the value at the intersection of the second row and third column.

5 Simulation algorithm

The algorithm is centered on forming the computation tree from a given configuration, by first branching out into *rule nodes* for the rule changes at the current time. Then, the rule nodes propagate into *synapse nodes* for the synapse changes (now for the next time step). Lastly, the synapse nodes branch out into their own *configuration nodes* for the system configurations. Note that since we are only considering nondeterminism in the synapse level, the configuration nodes will only ever branch out into just one single rule node each.

The simulation main algorithm will go as in Algorithm 1, creating a computation "tree" (strictly speaking, since identical configuration nodes will be joined, it is more of a computation graph) up to a specified depth. It creates the graph by forming the configuration nodes in a breadth-first manner using a queue, then the subtree of each configuration node (up to two levels) if created in a depth-first manner. The history of the past configurations (for checking uniqueness) is created using some arbitrary data structure. The specific methods of the graph and queue (connect, dequeue, enqueue, pop, push, empty, tooDeep) would not be specified in detail. Details of the proofs of the following theorems are given in Appendix 1.

Algorithm 1: Main Algorithm

```
1  initializeValues()
   /* gets input, initializes matrices & vectors, and generally initializes
      system                                                              */
2  confs ← [Cf^(0)]
3  hist ← [Cf^(0) → node (Cf^(0))]
   /* mapping of all prev configurations to their nodes in the computation
      tree                                                                */
4  while not empty(confs) :
5      conf ← dequeue(confs)
6      if not tooDeep(conf) :
7          continue
8      rules ← getRules(conf)
9      while not empty(rules) :
10         rule ← pop(rules)
11         connect(conf, rule)
12         syns ← getSyns(conf, rule)
13         while not empty(syns) :
14             syn ← pop(syns)
15             connect(rule, syn)
16             cur ← getConf(conf, rule, syn)
17             if cur in hist :
18                 connect(syn, hist [ cur ] )
19             else:
20                 connect(syn,cur)
21                 if os < 2 : enqueue(confs,conf)
```

Algorithm 2 will then check the applicability of the rules. This is done using the P and Q vectors of the given system. The `for` loop in Line 3 would check if a^{n_i}, where n_i is the number of spikes in neuron σ_i, would satisfy the regular expression for each of the rules. Note that the output spikes are monitored by Line 9. `newFi` Line 5 returns an all-zero firing vector for rules of type $\mathcal{R}$. Line 13 would just check if the rule was already fired and is still applying a plasticity operation, where the timer would be at 0, since it did not just start firing then. The primed timer would just be copied over to the unprimed timer without changes. Otherwise, if the rule would only start to be applied, then Line 16 would start the timer. Lastly, Line 18 would return the appropriate rule node. Details of `newRule`() would not be given, except for it being the constructor of rule nodes.

Algorithm 2: Get Rule Nodes

```
   function getRules( conf ) :
1      k ← k + 1
2      os^(k), sp^(k) ← os^(k-1), 0
3      for each R in {S, P} :
4          Sp ← (C^(k-1) × Sr_R^T) − P_R
5          Fi_R^(k) ← newFi(R)
6          for i from 1 to r_R :
7              if (q_{R,i}, Sp_i > 0 and Sp_i mod q_{R,i} = 0) or q_{R,i}, Sp_i = 0 :
8                  fi_{R,i}^(k) ← 1
9                  if R = S and Sr_{S,out,i} = 1 : os^(k), sp^(k) ← os^(k) + 1, 1
10             else :
11                 fi_{R,i}^(k) ← 0
12     for i from 1 to r_P :
13         if 1 in Ti'_i^(k-1) :
14             fi_{P,i}^(k) ← 0
15             Ti_i^(k) ← Ti'_i^(k-1)
16         else if fi_{P,i}^(k) = 1 :
17             Ti_i^(k) ← St_i
18     return [ newRule(Fi^(k), Ti^(k), os^(k), sp^(k)) ]
```

Theorem 1 *For an SNPSP system* Π, *the* `getRules`() *function* (*as described in Algorithm* 2) *generates a list of all the applicable rule nodes* $Rule^{(k)}$ *given* $Conf^{(k-1)}$.

Algorithm 3 would generate each configuration based on the possible combinations of candidate neurons nondeterministically selected by plasticity rules. `getCandidates`() would generate all permutations for this given these candidates (based on *NM* and *KV* and whichever of the synapses are existent on $Sy^{(k)}$). These candidates are all stored the `syns` stack, as pushed in Line 4. It would return a vector of (Sy_+, Sy_-) pairs.

Algorithm 3: Get Synapse Nodes

```
   function getSyns( conf, rule ) :
1      syns ← [ ]
2      for each (Sy_+, Sy_-) in getCandidates(Sr_P, NM, KV, Sy^(k)) :
           /* gets a list of all possible combinations of candidate synapses
              based on the N of the rule and the previous synapse
              connections                                                 */
3          Sy_Δ^(k) ← Sy_+^(k) − Sy_-^(k)
4          push(syns, newSyn(Sy_Δ^(k)))
5      return syns
```

Theorem 2 *For an SNPSP system* Π, *the* `getSyns`() *function* (*as described in Algorithm* 3) *returns* $Syn^{(k)}$ *given* $Conf^{(k-1)}$, *and* $Rule^{(k)}$.

Lastly, Algorithm 4 is focused on creating the current configuration given the previous one. First, we note that the kth configuration can be calculated from the total gain and the total loss as such

$$C^{(k)} = C^{(k-1)} + G^{(k)} - L^{(k)}$$

We are classifying the gains or losses according to the type of the causing rule, and thus $L^{(k)} = L_{\mathcal{P}}^{(k)} + L_{\mathcal{S}}^{(k)}$ and $G^{(k)} = G_{\mathcal{P}}^{(k)} + G_{\mathcal{S}}^{(k)}$.

Spike gains from spiking rules can be computed by checking the rules that fired and then tracing the source neurons of those rules and the destination of their corresponding out-synapses. Therefore

Theorem 3 *For an SNPSP system Π with m neurons and $r_{\mathcal{S}}$ spiking rules, where $d : 1, \ldots, r_{\mathcal{S}}$ is a total order for the spiking rules, the total spike gain from spiking rules at time k can be computed using*

$$G_{\mathcal{S}}^{(k)} = Fi_{\mathcal{S}}^{(k)} \times Sr_{\mathcal{S}} \times Sy^{(k)}$$

On the other hand, plasticity rules can only cause spike gains during synapse creation. Thus, gains from plasticity rules can be computed by checking the destination of the newly created synapses, if any. In symbol form

Theorem 4 *For an SNPSP system Π with m neurons and $r_{\mathcal{P}}$ plasticity rules, where $d : 1, \ldots, r_{\mathcal{P}}$ is a total order for the plasticity rules, the total spike gain from plasticity rules at time k can be computed by summing all of the rows of $Sy_{+}^{(k)}$ using $G_{\mathcal{P}}^{(k)} = \sum_{i=1}^{r_{\mathcal{P}}} Sy_{+,i}^{(k)}$.*

Spikes are only lost on consumption during rule firing. So for both rule types, this is computed from checking how many spikes are consumed according to the rules and then checking the source neurons of these rules.

Theorem 5 *For an SNPSP system Π with m neurons, $r_{\mathcal{S}}$ spiking rules, $r_{\mathcal{P}}$ plasticity rules, where $d_{\mathcal{R}} : 1, \ldots, r_{\mathcal{R}}$ is a total order for the spiking [plasticity] rules and $\mathcal{R} = \mathcal{S}$ [$\mathcal{R} = \mathcal{P}$], the total spike loss from spiking [plasticity] rules at time k can be computed using*

$$L_{\mathcal{R}}^{(k)} = (Fi_{\mathcal{R}}^{(k)} \odot Co_{\mathcal{R}}) \times Sr_{\mathcal{R}}$$

where $\mathcal{R} \in \{\mathcal{S}, \mathcal{P}\}$, and $\odot$ is element-wise multiplication.

Algorithm 4: Get Configuration Nodes

```
function getConf( conf, rule, syn ) :
1    Sy^(k) ← Sy^(k−1) + Sy_Δ^(k)
2    G_S^(k) ← Fi_S^(k) × Sr_S × Sy^(k)
3    G_P^(k) ← sumRows(Sy_+^(k))
4    L_S^(k) ← (Fi_S^(k) ⊙ Co_S) × Sr_S
5    L_P^(k) ← (Fi_P^(k) ⊙ Co_P) × Sr_P
6    G^(k) ← G_P^(k) + G_S^(k)
7    L^(k) ← L_P^(k) + L_S^(k)
8    C^(k) ← C^(k−1) + G^(k) − L^(k)
9    for i from 1 to r_P do
10       for j from 1 to 2 do
11           ti'_{i,j}^(k) ← max(ti_{i,j}^(k) − 1,0)
12   return newConf((C^(k),Sy^(k),Ti'^(k))
```

Table 5 Space and time complexities of algorithms presented

Algorithm	Time Complexity	Space Complexity	Notes
Tree (node count)	–	$O((2e)^{m^2t/2})$	
Algorithm 1	$O(F(F+mr))$	$O(F + F^{r/(m^2)})$	$F = (2e)^{m^2t/2}m^2$
Algorithm 2	–	$O(mr)$	
Algorithm 3	$O((2e)^{m^2t/2}m^2)$	–	
Algorithm 4	$O(mr)$	–	

Theorem 6 *For an SNPSP system Π, the* `getConf()` *function (as described in Algorithm 4) returns $Conf^{(k)}$ given $Conf^{(k-1)}$, $Rule^{(k)}$, and $Syn^{(k)}$.*

Theorem 7 *For an SNPSP system Π (that follows the restrictions assumed in this paper), Algorithm 1 can correctly simulate the computation of Π and generate a correct computation tree (graph).*

Further algorithm analysis and proof of correctness are detailed in [10] and Appendix 1, with a summary in Table 5. In the next section, we give an example of our algorithms in this section to simulate a small SNPSP system.

6 Example simulation

In this section, we demonstrate the matrix representation and algorithms from the previous section using Π_{ex} from Fig. 1. Note that for the illustrations to follow, the matrices and vectors that define an aggregate version that combines those for plasticity and for spiking rules (i.e. Sr, P, Q, Fi, Co), the *arbitrary ordering* as specified in their respective definitions would simply be the concatenation of those for the spiking rules and for the plasticity rules. In other words, as with the rule firing vector, the resulting vector would be

$$Fi^{(k)} = \left[fi_{\mathcal{S},0}^{(k)} \cdots fi_{\mathcal{S},r_{\mathcal{S}}}^{(k)} \middle| fi_{\mathcal{P},0}^{(k)} \cdots fi_{\mathcal{P},r_{\mathcal{P}}}^{(k)} \right]$$

Given the initial values as computed above, the initial configuration of the system is

$$Cf^{(0)} = \langle Rule^{(0)} | Syn^{(0)} | Conf^{(0)} \rangle = \left\langle \begin{bmatrix} 0 & 0 & 0 & 0 | 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, 0, 0 \middle| \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} \middle| \begin{bmatrix} 2 & 0 & 1 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&1&1 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \right\rangle \tag{10}$$

We have already computed for the rule firing vectors and the timer matrix at time 1, which are

$$Fi^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 | 1 & 0 \end{bmatrix} \tag{11}$$

$$Ti^{(1)} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \tag{12}$$

Since we have already decided that rules $r_{\mathcal{S},2}$ and $r_{\mathcal{P},1}$ are to fire, we could proceed to selecting which synapses are to be operated on using $r_{\mathcal{P},1}$. Since in the rule source matrix, $sr_{\mathcal{P},1,1} = 1$ (for *plasticity* rule number *one*, for the *first* neuron), then the source neuron of rule $r_{\mathcal{P},1}$ is neuron σ_1. The candidate destination neurons for the same rule are neurons σ_2 and σ_3, since $nm_{1,2} = nm_{1,3} = 1$ (for plasticity rule number *one*, for the *second* and *third* neurons). The operation is $o_1 = +$, given that $ti_{1,1} = 1$ (for plasticity rule number *one*, for the *first* neuron). Thus, we are to select $kv_1 = 1$ neuron from these two candidates to which we would *create* a synapse to (since the chosen operation is $op_1 = +$ for *synapse creation*). In the example where Π_{ex} computed 4, the first selected neuron was σ_2.

Since rule $r_{\mathcal{S},2}$ is in neuron $\sigma_3 = \sigma_{out}$ and has fired, we know that it has caused a spike to be sent to the environment at time 1. Therefore, the output spike count and indicator are $os^{(1)} = 1$ and $sp^{(1)} = 1$.

Afterwards, we could now create the next configuration. We have

$$Sy_{\Delta}^{(1)} = Sy_{+}^{(1)} = \begin{bmatrix} 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} \tag{13}$$

$$\begin{aligned} Sy^{(1)} &= Sy^{(0)} + Sy_{\Delta}^{(1)} = Sy^{(0)} + Sy_{+}^{(1)} - Sy_{-}^{(1)} = Sy^{(0)} + Sy_{+}^{(1)} \\ &= \begin{bmatrix} 0&0&0&0&0 \\ 0&0&0&1&1 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix} + \begin{bmatrix} 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&1&0&0&0 \\ 0&0&0&1&1 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix} \end{aligned} \tag{14}$$

Now we can use Algorithm 4 to create the matrices for the next configuration.

$$\begin{aligned} G_{\mathcal{S}}^{(1)} &= Fi_{\mathcal{S}}^{(0)} \times Sr_{\mathcal{S}} \times Sy^{(1)} \\ &= \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \end{bmatrix} \\ &\times \begin{bmatrix} 0&1&0&0&0 \\ 0&0&0&1&1 \\ 0&0&0&0&0 \\ 1&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 0&0&0&0&0 \end{bmatrix} \end{aligned} \tag{15}$$

$$\begin{aligned} G_{\mathcal{P}}^{(1)} &= sumRows(Sy_{+}^{(1)}) \\ &= \begin{bmatrix} 1&1&1&1&1 \end{bmatrix} \times Sy_{+}^{(1)} \\ &= \begin{bmatrix} 1&1&1&1&1 \end{bmatrix} \times \begin{bmatrix} 0&1&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \\ 0&0&0&0&0 \end{bmatrix} \\ &= \begin{bmatrix} 0&1&0&0&0 \end{bmatrix} \end{aligned} \tag{16}$$

$$\begin{aligned} L_{\mathcal{S}}^{(1)} &= (Fi_{\mathcal{S}}^{(0)} \odot Co_{\mathcal{S}}) \times Sr_{\mathcal{S}} \\ &= \left(\begin{bmatrix} 0&1&0&0 \end{bmatrix} \odot \begin{bmatrix} 1&1&1&1 \end{bmatrix} \right) \\ &\times \begin{bmatrix} 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&1 \end{bmatrix} \\ &= \begin{bmatrix} 0&0&1&0&0 \end{bmatrix} \end{aligned} \tag{17}$$

$$\begin{aligned} L_{\mathcal{P}}^{(1)} &= (Fi_{\mathcal{P}}^{(0)} \odot Co_{\mathcal{P}}) \times Sr_{\mathcal{P}} \\ &= \left(\begin{bmatrix} 1&0 \end{bmatrix} \odot \begin{bmatrix} 1&1 \end{bmatrix} \right) \\ &\times \begin{bmatrix} 1&0&0&0&0 \\ 1&0&0&0&0 \end{bmatrix} \\ &= \begin{bmatrix} 1&0&0&0&0 \end{bmatrix} \end{aligned} \tag{18}$$

$$L^{(1)} = L_{\mathcal{S}}^{(1)} + L_{\mathcal{P}}^{(1)} = \begin{bmatrix} 0&0&1&0&0 \end{bmatrix} + \begin{bmatrix} 1&0&0&0&0 \end{bmatrix} = \begin{bmatrix} 1&0&1&0&0 \end{bmatrix} \tag{19}$$

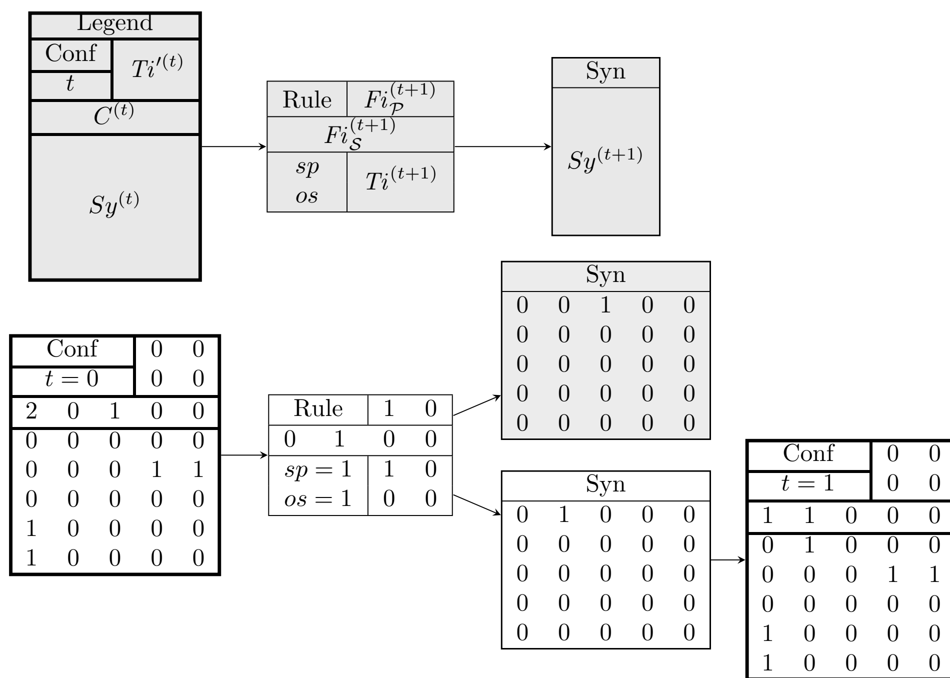

Fig. 2 Incomplete computation tree for Π_{ex} computing $\{4\}$

$$G^{(1)} = G_{\mathcal{S}}^{(1)} + G_{\mathcal{P}}^{(1)} = \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 0 & 0 & 0\end{bmatrix} = \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix} \tag{20}$$

$$C^{(1)} = C^{(0)} + G^{(1)} - L^{(1)} = \begin{bmatrix}2 & 0 & 1 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 1 & 0 & 0 & 0\end{bmatrix} - \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix} = \begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix} \tag{21}$$

Finally, the timer matrix would count down and we would have

$$Ti'^{(1)} = \begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix} \tag{22}$$

Given the computation as illustrated so far, the current state of the computation tree is shown in Fig. 2.

7 Closing remarks

The main purpose of this work is to provide a matrix representation and simulation algorithm for SNPSP systems, similar to what has been done for other variants of SNP systems. The main difference between our representation and matrix representations of other variants is taking care of the semantics of plasticity, e.g. we take into consideration the plasticity of the synapses (thus we have $Sy^{(k)}$ to indicate that the synapses change per time step, as opposed to the *Sy* matrix of [27]), and the timer for $\pm$ and $\mp$ in plasticity rules. Ideas from this work may be employed to represent other SNP system variants, e.g. those with neuron division or budding mentioned in Ref. [15].

In implementing the simulation algorithm, the simulation speed can be accelerated using parallel processors such as GPUs. If the implementation does not check for uniqueness of configurations, then the simulation can be further accelerated by performing parallel configuration generations. Otherwise, the configurations would have to be checked sequentially. If there would be no loops in the computation tree, then the implementation can choose not to perform uniqueness checking.

A future work for this paper includes a software implementation of the algorithm in GPU and CPU. As previously mentioned, for this work we are only dealing with SNPSP systems with determinism on the rule-level. We also note that at present, the matrix representation seems to be applicable to *asynchronous* version of SNPSP systems, see e.g. [3]. In asynchronous mode of rule application, as opposed to the synchronous mode in this work, at each step a neuron can nondeterministically choose not to apply a rule even if a rule can be applied. However, the algorithms given in this work must be modified to include this additional level of nondeterminism.

Acknowledgements R.T.A. de la Cruz is supported by a graduate scholarship from the DOST-ERDT project. F.G.C. Cabarle thanks the support from the DOST-ERDT project; the Dean Ruben A. Garcia PCA AY2018–2019, and an RLC AY2018–2019 grant of the OVCRD, both from UP Diliman. H. Adorna would like to appreciate and thank the support granted by UPD-OVCRD RCL grant, ERDT Research Program of the College of Engineering, UP Diliman and the Semirara Mining Corporation Professorial Chair for Computer Science. N. Hernandez is supported by the Vea Technology for All professorial chair. The work of X. Zeng was supported by the National Natural Science Foundation of China (Grant Nos. 61472333, 61772441, 61472335,

61672033, 61425002, 61872309, 61771331), Project of marine economic innovation and development in Xiamen (No. 16PFW034SF02), Natural Science Foundation of the Higher Education Institutions of Fujian Province (No. JZ160400). K. Buño would like to thank Dr. Olegario G. Villoria Jr. Professorial Chair on Transportation/Logistics since 2018 until present.

Theorem proofs

Proof for Theorem 1 By definition, $Rule^{(k)} = (Fi^{(k)}, Ti^{(k)}, os^{(k)}, sp^{(k)})$. First, given that the input $Conf^{(k-1)}$ is of the previous time step (fed into the function as $Conf^{(k)}$), we first increment k at Line 1 for appropriate usage in the resulting rule node. Thus, we know that the `newRules()` constructor at Line 18 is of the right time step. Line 4 evaluates a formula and assigns it to a temporary variable Sp, for spikes. The formula consists of two parts, the multiplication and the subtraction. It goes as follows:

$$\begin{aligned} Sp &= \left(C^{(k-1)} \times Sr_{\mathcal{R}}^T\right) - P_{\mathcal{R}} \\ &= \left(\left[c_i^{(k-1)}\right]_m \times \left[sr_{\mathcal{R},i,j}\right]_{m\times r_{\mathcal{R}}}\right) - \left[p_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} \\ &= \left(\left[\sum_i c_i^{(k-1)} sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}}}\right) - \left[p_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} \end{aligned}$$

Since $sr_{\mathcal{R},i,j} = 1$ if rule $r_{\mathcal{R},i}$ belongs to neuron σ_j (0 otherwise), and $c_i^{(k)}$ is the number of spikes in neuron σ_i at time k, we have

$$c_i^{(k-1)} sr_{\mathcal{R},i,j} = \begin{cases} c_i^{(k-1)}, & r_{\mathcal{R},j} \in R_i; \\ 0, & \text{otherwise.} \end{cases}$$

Also noting that rules can only be associated with one neuron, we can then conclude that $\sum_i c_i^{(k-1)} sr_{\mathcal{R},i,j}$ is the number of spikes in the source neuron of rule $r_{\mathcal{R},j}$. We let $rsp_{\mathcal{R},i}$ be this number. Now that we know each rule's respective source neuron spike count; we can now use the P and Q vectors to check compatibility with the rule's respective regular expression. Thus

$$\begin{aligned} Sp &= \left(\left[\sum_i c_i^{(k-1)} sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}}}\right) - \left[p_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} \\ &= \left[rsp_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} - \left[p_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} = \left[rsp_{\mathcal{R},i} - p_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} \end{aligned}$$

With s being the current spike count of a certain neuron, we need to match a^s with $a^p(a^q) *= a^{p+qn}$, and thus we need to make sure $s = p + qn$ for some nonnegative integers p,q, and n. So we first subtract p in Line 4, and check for qn in the if clause of Line 7. There are two cases for a^s to match the regular expression. First, if there is a non-zero q for the rule, there is no problem using $Sp_i \mathbf{mod} q_{\mathcal{R},i} = 0$ (so long as Sp_i is not negative, in which case $rsp_{\mathcal{R},i} - p_{\mathcal{R},i} = s - p < 0$). The other case would be if $q = 0$, in which case the regular expression is of the form a^p. Thus, $p + qn = p$, a constant, and so $s = p + qn$ can only be satisfied if $s - p = rsp_{\mathcal{R},i} - p_{\mathcal{R},i} = 0$. If the regular expression is matched, $fi_{\mathcal{R}}, i^{(k)} = 1$; otherwise, $= 0$. Since the loop of Line 6 iterates over all the rules of type $\mathcal{R}$, and that $\mathcal{R}$ goes through both $\mathcal{P}$ and $\mathcal{S}$ (Line 3), these two loops go over all the rules. Thus, $Fi^{(k)}$ now tells us which rules have matched their regular expressions and can fire.

$os^{(k)}$ would by default copy the value from the previous time step, $os^{(k)}$, while $sp^{(k)}$ would stay at 0. The former would only increase and the latter be set to 1 if an output spike was discovered to be sent to the environment at time k. This condition is checked by the `if` clause at Line 9, which would only be reached if rule $r_{\mathcal{R},i}$ were to fire at time k for the given values of $\mathcal{R}$ and i. Thus, we only need to check if this rule sent an output spike. Since only spiking rules can send spikes to the environment, the condition at Line 9 should check if the given rule was a spiking rule ($\mathcal{R} = \mathcal{S}$) and if the current rule belonged to the output neuron ($Sr_{\mathcal{S},out,i} = 1$). Thus, $os^{(k)}$ and $sp^{(k)}$ are now computed correctly.

Lastly, the timer matrix $Ti^{(k)}$ would only be touched in the `for` loop of Line 12. For each plasticity rule, we first check if the rule already fired at the previous time step (Line 13) and is still executing at the current time step (as with the $\pm$ and $\mp$ rules). This could be checked by looking for a 1 in the primed timers of the said rule from the previous time step ($Ti_{i'}^{(k-1)}$), since the timers have already counted down after initial rule firing. $fi_{\mathcal{P},i}^{(k)}$ is simply marked as 0 since the rule is not allowed to fire anew if it is still executing, and just copies the previous primed timers onto the current unprimed timers. Otherwise, if the rule is not to execute a second operation at the current time step, we check if it fired anew at the current time step (Line 9). Since $Fi_{\mathcal{R}}^{(k)}$ now shows which rules are applicable (unless ongoing execution), we can now be sure that the rules *will be* applied at time step k and thus we start the timer (Line 17). Given that, we are now sure that $Fi^{(k)}$ and $Ti^{(k)}$ are now computed correctly.

Therefore, we are now sure that $os^{(k)}$, $sp^{(k)}$, $Fi^{(k)}$, and $Ti^{(k)}$ are computed correctly. `newRule()` is thus sure to be fed the correct arguments, and will return the correct rule node. □

Proof for Theorem 2 Here, we return a list of all possible synapse nodes $Syn^{(k)}$. `getCandidates()` has not been specified in this paper and is assumed to return a list of all possible combinations of created/deleted synapses based on permutations of destination neurons and synapse counts of applicable plasticity rules. Given this, we are ensured that $Sy_+^{(k)}$ and $Sy_-^{(k)}$ are the appropriate synapse creation and

deletion matrices of each synapse node to be created. Thus, $Sy_{\Delta}^{(k)}$ would then hold the appropriate synapse change matrix for the same synapse node and would thus be appropriately passed onto the constructor for $Syn^{(k)}$ and be included in the return list. Therefore, `getSyns()` returns the correctly computed synapse nodes appropriate for the given rule node. □

Proof for Theorem 3 Given the definitions of $Fi_{\mathcal{S}}$, $Sr_{\mathcal{S}}$, and Sy, we have

$$\begin{aligned} G_{\mathcal{S}}^{(k)} &\stackrel{?}{=} Fi_{\mathcal{S}}^{(k)} \times Sr_{\mathcal{S}} \times Sy^{(k)} \\ &= \left[fi_{\mathcal{S},i}^{(k)}\right]_{r_{\mathcal{R}}} \times \left[sr_{\mathcal{S},i,j}\right]_{r_{\mathcal{R}} \times m} \times \left[sy_{i,j}^{(k)}\right]_{m \times m} \\ &= \left[\sum_i fi_{\mathcal{S},i}^{(k)} sr_{\mathcal{S},i,j}\right]_m \times \left[sy_{i,j}^{(k)}\right]_{m \times m} \end{aligned}$$

Since $fi_{\mathcal{S},i}^{(k)} = 1$ if rule $r_{\mathcal{S},i}$ has spiked at time k (0 otherwise), and $sr_{\mathcal{S},i,j} = 1$ if $r_{\mathcal{S},i}$ belongs to neuron σ_j (0 otherwise), we have

$$fi_{\mathcal{S},i}^{(k)} sr_{\mathcal{S},i,j} = \begin{cases} 1, & r_{\mathcal{S},i} \lightning^{(k)} \in R_j; \\ 0, & \text{otherwise.} \end{cases}$$

Thus, $\sum_i fi_{\mathcal{S},i}^{(k)} sr_{\mathcal{S},i,j}$ is the number of spiking rules that have spiked at time k from neuron σ_j. However, given that we have restricted neurons to only fire a maximum of one rule each, the value of this summation will only ever be 0 or 1, only indicating whether the neuron had a spiking rule fire or not. Continuing further, $sy_{i,j}^{(k)} = 1$ if neuron σ_i is connected to σ_j at time k (0 otherwise), so

$$\begin{aligned} G_{\mathcal{S}}^{(k)} &\stackrel{?}{=} \left[\sum_i fi_{\mathcal{S},i}^{(k)} sr_{\mathcal{S},i,j}\right]_m \times \left[sy_{i,j}^{(k)}\right]_{m \times m} \\ &= \left[\sigma_i \overset{\Sigma s}{\underset{(k)}{\rightarrow}} \sigma\right]_m \times \left[sy_{i,j}^{(k)}\right]_{m \times m} = \left[\sigma_i \overset{s}{\underset{(k)}{\rightarrow}} \sigma\right]_m \times \left[\sigma_i \underset{(k)}{\rightarrow} \sigma_j\right]_{m \times m} \\ &= \left[\sum_i \left(\left(\sigma_i \overset{s}{\underset{(k)}{\rightarrow}} \sigma\right)\left(\sigma_i \underset{(k)}{\rightarrow} \sigma_j\right)\right)\right]_m = \left[\sum_i \sigma_i \overset{s}{\underset{(k)}{\rightarrow}} \sigma_j\right]_m = \left[\sigma \overset{\Sigma s}{\underset{(k)}{\rightarrow}} \sigma_j\right]_m \end{aligned}$$

Spiking rules can only cause spike gains in a destination neuron if some other source neuron fires a spiking rule to the said destination, and so we finally have

$$G_{\mathcal{S}}^{(k)} \stackrel{?}{=} \left[\sigma \overset{\Sigma s}{\underset{(k)}{\rightarrow}} \sigma_j\right]_m = \left[g_{\mathcal{S},j}^{(k)}\right]_m \stackrel{\checkmark}{=} G_{\mathcal{S}}^{(k)}$$

□

Proof for Theorem 4 Since plasticity rules can only cause spike gains by creating synapses (because creating synapses would inherently send one spike to the destination neuron), we only need to check $Sy_{+}^{(k)}$. Given the definition of Sy_{+}, we have

$$\begin{aligned} G_{\mathcal{P}}^{(k)} &\stackrel{?}{=} \sum_{i=1}^{r_{\mathcal{P}}} Sy_{+,i}^{(k)} = \left[\sum_i sy_{+,i,j}^{(k)}\right]_m = \left[\sum_i \sigma_i \overset{+}{\underset{(k)}{\rightarrow}} \sigma_j\right]_m \\ &= \left[\sigma \overset{\Sigma +}{\underset{(k)}{\rightarrow}} \sigma_i\right]_m = \left[g_{\mathcal{P},i}^{(k)}\right]_m \stackrel{\checkmark}{=} G_{\mathcal{P}}^{(k)} \end{aligned}$$

□

Proof for Theorem 5 Both spiking and plasticity rules can only cause spike loss through spike consumption upon rule firing. Thus

$$\begin{aligned} L_{\mathcal{R}}^{(k)} &\stackrel{?}{=} (Fi_{\mathcal{R}}^{(k)} \odot Co_{\mathcal{R}}) \times Sr_{\mathcal{R}} \\ &= \left(\left[fi_{\mathcal{R},i}^{(k)}\right]_{r_{\mathcal{R}}} \odot \left[co_{\mathcal{R},i}\right]_{r_{\mathcal{R}}}\right) \times \left[sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}} \times m} = \left(\left[r_{\mathcal{R},i} \lightning^{(k)}\right]_{r_{\mathcal{R}}} \odot \left[co_{\mathcal{R},i}\right]_{r_{\mathcal{R}}}\right) \times \left[sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}} \times m} \\ &= \left[r_{\mathcal{R},i} \lightning^{(k)} co_{\mathcal{R},i}\right]_{r_{\mathcal{R}}} \times \left[sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}} \times m} = \left[r_{\mathcal{R},i} \lightning_c^{(k)}\right]_{r_{\mathcal{R}}} \times \left[sr_{\mathcal{R},i,j}\right]_{r_{\mathcal{R}} \times m} \\ &= \left[\sum_i r_{\mathcal{R},i} \lightning_c^{(k)} sr_{\mathcal{R},i,j}\right]_m \end{aligned}$$

Since $sr_{\mathcal{R},i,j}$ will only have a nonzero value if rule $r_{\mathcal{R},i}$ is in neuron σ_j, we have

$$r_{\mathcal{R},i} \lightning_c^{(k)} sr_{\mathcal{R},i,j} = \begin{cases} r_{\mathcal{R},i} \lightning_c^{(k)}, & r_{\mathcal{R},i} \in R_j; \\ 0, & \text{otherwise} \end{cases}$$

Spike losses will only ever be caused by spike consumption from rule firing in a given neuron. Thus, also given the definition of $\sigma_j \lightning_{c,\mathcal{R},i}^{(k)}$

$$L_{\mathcal{R}}^{(k)} \stackrel{?}{=} \Big[\sum_i r_{\mathcal{R},i} \lightning_c^{(k)} sr_{\mathcal{R},i,j}\Big]_m = \Big[\sum_i \sigma_j \lightning_{c,R,i}^{(k)}\Big]_m = \Big[\sigma_i \lightning_{c,R}^{(k)}\Big]_m = \Big[l_{\mathcal{R},i}^{(k)}\Big]_m \stackrel{\checkmark}{=} L_{\mathcal{R}}^{(k)}$$

Lines 9–11 would tick the timer to get $Ti'^{(k)}$, by manually decreasing each element of the matrix by 1 unless the value is 0. □

Proof for Theorem 6 By definition, $Conf^{(k)} = (C^{(k)}, Sy^{(k)}, Ti'^{(k)})$. Lines 1 to 8 have been proven to correctly compute for $C^{(k)}$ and $Sy^{(k)}$. The loop in Line 9 iterates over all plasticity rules, while the inner loop of Line 10 goes over the two plasticity operations creation (1) and deletion (2). Line 11 would then either count down the current unprimed timer ($ti_{i,j}^{(k)} - 1$), or keep it at zero (`max`). Thus, the loops correctly compute for $Ti'^{(k)}$. Line 12 thus returns the correct configuration node via the constructor for $Conf^{(k)}$, being passed the correct arguments for $C^{(k)}$, $Sy^{(k)}$, and $Ti'^{(k)}$. □

Proof for Theorem 7 The first three lines are just for initialization. The loop in Line 4 iterates over the configuration nodes in a breadth-first manner (seen by the use of `dequeue` and `enqueue`). Line 6 would cut off the computation graph once it reaches a given depth. The loop in Line 9 would go through the rule nodes, connecting them to configuration nodes first before heading to the loop in Line 13. This loop would go through the synapse nodes and connect them to the rule nodes, and then generates a new configuration node in Line 16. These two inner loops, from the rule nodes down to the immediate next configuration nodes, would generate these three levels in a depth-first manner (as seen with `pop` and `push`). Essentially, what happens is (1) given a configuration node, generate the subtree of these configuration nodes up to three levels in depth-first manner, (2) go through these configuration nodes in breadth-first manner. □

References

1. Cabarle, F. G. C., Adorna, H. N., Jiang, M., & Zeng, X. (2017). Spiking neural p systems with scheduled synapses. *IEEE Transactions on Nanobioscience*, *16*(8), 792–801.
2. Cabarle, F. G. C., Adorna, H. N., Martínez-del-Amor, M. Á., & Pérez-Jiménez, M. J. (2012). Improving GPU simulations of spiking neural P systems. *ROMJIST*, *15*(1), 5–20.
3. Cabarle, F.G.C., Adorna, H.N., & Pérez-Jiménez, M.J. (2015) Asynchronous spiking neural P systems with structural plasticity. In C. S. Calude, M. J. Dinneen (Eds.), *International conference on unconventional computation and natural computation* (pp. 132–143). Cham: Springer.
4. Cabarle, F. G. C., Adorna, H. N., Pérez-Jiménez, M. J., & Song, T. (2015). Spiking neural p systems with structural plasticity. *Neural Computing and Applications*, *26*(8), 1905–1917.
5. Carandang, J. P., Cabarle, F. G. C., Adorna, H. N., Hernandez, N. H. S., & Martinez-del Amor, M. A. (2019). Handling non-determinism in spiking neural P systems: Algorithms and simulations. *Fundamenta Informaticae*, *164*, 139–155. https://doi.org/10.3233/FI-2019-1759.
6. Carandang, J. P. A., Villaflores, J. M. B., Cabarle, F. G. C., Adorna, H. N., & Martinez-del Amor, M. A. (2017). Cusnp: Spiking neural p systems simulators in cuda. *Romanian Journal for Information Science and Technology (ROMJIST)*, *20*(1), 57–70.
7. Chen, H., Freund, R., Ionescu, M., Păun, G., & Pérez-Jiménez, M. J. (2007). On string languages generated by spiking neural p systems. *Fundamenta Informaticae*, *75*(1–4), 141–162.
8. Chen, H., Ionescu, M., Ishdorj, T. O., Păun, A., Păun, G., & Pérez-Jiménez, M. J. (2008). Spiking neural p systems with extended rules: Universality and languages. *Natural Computing*, *7*(2), 147–166.
9. Dela Cruz, R. T., Cailipan, D., Cabarle, F. G. C., Hernandez, N., Buño, K., Adorna, H., & Carandang, J. (2018) Matrix representation and simulation algorithm for spiking neural p systems with rules on synapses. In *Proceedings of 18th Philippine Computing Science Congress (PCSC2018), 15–17 March, 2018, Cagayan de Oro City, Misamis Oriental, Philippines* (pp. 104–112). https://sites.google.com/site/2018pcsc/proceedings. Accessed 4 Aug 2019.
10. Dela Cruz, R. T., Jimenez, Z. B., Cabarle, F. G. C., Hernandez, N., Buño, K., Adorna, H., & Carandang, J. (2018) Matrix representation of spiking neural p systems with structural plasticity. In *Proceedings of 18th Philippine Computing Science Congress (PCSC2018), 15–17 March, 2018, Cagayan de Oro City, Misamis Oriental, Philippines* (pp. 104–112). https://sites.google.com/site/2018pcsc/proceedings. Accessed 4 Aug 2019
11. Ionescu, M., Păun, G., & Yokomori, T. (2006). Spiking neural p systems. *Fundamenta informaticae*, *71*(2, 3), 279–308.
12. Ishdorj, T. O., & Leporati, A. (2008). Uniform solutions to sat and 3-sat by spiking neural p systems with pre-computed resources. *Natural Computing*, *7*(4), 519–534.
13. Ishdorj, T. O., Leporati, A., Pan, L., Zeng, X., & Zhang, X. (2010). Deterministic solutions to qsat and q3sat by spiking neural p systems with pre-computed resources. *Theoretical Computer Science*, *411*(25), 2345–2358.
14. Leporati, A., Mauri, G., Zandron, C., Păun, G., & Pérez-Jiménez, M. J. (2009). Uniform solutions to sat and subset sum by spiking neural p systems. *Natural Computing*, *8*(4), 681.
15. Martínez-del-Amor, M. Á., Orellana-Martín, D., Cabarle, F. G. C., Pérez-Jiménez, M. J., & Adorna, H. N. (2017) Sparse-matrix representation of spiking neural P systems for GPU. In C. Graciani, G. Păun, A. Riscos-Núñez, L. Valencia-Cabrera (Eds.), *Proceedings of 15th brainstorming week on membrane computing* (pp. 161–170). Seville: Fénix Editora. http://www.gcn.us.es/15bwmc_proceedings.
16. Pan, L., & Păun, G. (2009). Spiking neural p systems with anti-spikes. *International Journal of Computers Communications and Control*, *4*(3), 273–282.
17. Pan, L., Păun, G., & Pérez-Jiménez, M. J. (2011). Spiking neural p systems with neuron division and budding. *Science China Information Sciences*, *54*(8), 1596.
18. Pan, L., Wang, J., & Hoogeboom, H. J. (2012). Spiking neural p systems with astrocytes. *Neural Computation*, *24*(3), 805–825.
19. Pan, L., Zeng, X., Zhang, X., & Jiang, Y. (2012). Spiking neural p systems with weighted synapses. *Neural Processing Letters*, *35*(1), 13–27.
20. Păun, G. (2007). Spiking neural p systems. a tutorial. Bulletin of the European Association for Theoretial Computer Science **91**,

145–159. http://cs.ioc.ee/yik/schools/win2007/paun/snppalmse.pdf. Accessed 4 Aug 2019.
21. Paun, G. (2007). Spiking neural p systems with astrocyte-like control. *Journal of UCS*, *13*(11), 1707–1721.
22. Peng, H., Wang, J., Pérez-Jiménez, M. J., Wang, H., Shao, J., & Wang, T. (2013). Fuzzy reasoning spiking neural p system for fault diagnosis. *Information Sciences*, *235*, 106–116.
23. Song, T., Pan, L., & Păun, G. (2014). Spiking neural p systems with rules on synapses. *Theoretical Computer Science*, *529*, 82–95.
24. Song, T., Rodríguez-Patón, A., Zheng, P., & Zeng, X. (2017). Spiking neural p systems with colored spikes. *IEEE Transactions on Cognitive and Developmental Systems*, *10*(4), 1106–1115.
25. Wang, J., Hoogeboom, H. J., Pan, L., Păun, G., & Pérez-Jiménez, M. J. (2010). Spiking neural p systems with weights. *Neural Computation*, *22*(10), 2615–2646.
26. Wang, T., Zhang, G., Zhao, J., He, Z., Wang, J., & Pérez-Jiménez, M. J. (2014). Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural p systems. *IEEE Transactions on Power Systems*, *30*(3), 1182–1194.
27. Zeng, X., Adorna, H. N., Martínez-del Amor, M. Á., Pan, L., & Pérez-Jiménez, M. J. (2010). Matrix representation of spiking neural p systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, A. Salomaa (Eds.), *International conference on membrane computing* (pp. 377–391). Springer.
28. Zhang, G., Rong, H., Neri, F., & Pérez-Jiménez, M. J. (2014). An optimization spiking neural p system for approximately solving combinatorial optimization problems. *International Journal of Neural Systems*, *24*(05), 1440006.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Zechariah B. Jimenez completed the bachelor's degree in computer science in 2019 from the University of the Philippines Diliman. At present, he is a master's student in the same university. His research interests include membrane computing and parallel computing.

Francis George C. Cabarle received the Ph.D. degree in computer science from the Algorithms and Complexity, at the Department of Computer Science, in the University of the Philippines Diliman, Quezon City, Philippines, in 2015. His current research interests include membrane computing, parallel computing, and automata and formal languages.

Ren Tristan A. de la Cruz is Ph.D. student in computer science from the Algorithms and Complexity Laboratory, Department of Computer Science, University of the Philippines Diliman, Quezon City, Philippines. He received his M.S. degree in computer science from the same university. His current research interests include membrane computing, parallel computing, and automata and formal languages.

Kelvin C. Buño received the masters degree in 2011 from the University of the Philippines Diliman. His research Interests include Automata Theory, Membrane Computing, Communication Complexity.

Henry N. Adorna received the Ph.D. degree in mathematics from the University of the Philippines in 2002. He is currently a Professor with the Algorithms and Complexity, Department of Computer Science, University of the Philippines Diliman, Quezon City, Philippines. His research interests include automata and formal languages, algorithmics for hard problems, discrete mathematics, and membrane computing.

Nestine Hope S. Hernandez received the masters degrees in mathematics (2003) and computer science (2009) both from the University of the Philippines Diliman. Her research interests include natural computing and combinatorial interconnection networks.

Xiangxiang Zeng is an Yuelu distinguished Professor with the College of Information Science and Engineering, Hunan University, Changsha, China. Before joining Hunan University in 2019, he was with Department of Computer Science in Xiamen University. He received the B.S. degree in automation from Hunan University, China, in 2005, the Ph.D. degree in system engineering from Huazhong University of Science and Technology, China, in 2011. His main research interests include membrane computing, neural computing and bioinformatics.