WebSnapse v3: Optimization of the Web-based Simulator of Spiking Neural P System using Matrix Representation, WebAssembly and Other Tools

Louie Gallos[1], Jose Lorenzo Sotto[1], Francis George C. Cabarle[1,2]*, and Henry N. Adorna[1]

[1]Dept. of Computer Science, College of Engineering
University of the Philippines Diliman, Diliman, Quezon City, 1101, Philippines;
lmgallos@up.edu.ph, jcsotto1@up.edu.ph, fccabarle@up.edu.ph, hnadorna@up.edu.ph
[2]Research Group on Natural Computing, Dept. of Computer Science and AI, I3US, SCORE lab, Universidad de Sevilla, Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
fcabarle@us.es

Abstract. Spiking Neural P systems (SN P system) are a kind of distributed and parallel computational model under membrane computing inspired by how neurons typically communicate with each other through the sending of spikes between synapses. To further study SN P systems, various dedicated simulators have been developed. WebSnapse, a web-based simulator of SN P systems, has been used to serve as visual aid for creating, modifying, and simulating SN P systems. This was upgraded on WebSnapse v2.0 by improving existing functionalities and adding support for more variants of SN P system. However, WebSnapse suffered performance and stability issues, especially when simulating larger SN P systems. This research aims to optimize the performance and stability of WebSnapse using a new way of computing the next state of the system while achieving feature parity with WebSnapse v2.0 and adding LaTeX support for rule input and visualization.

Keywords: membrane computing, spiking neural p system, matrices, simulation

1 INTRODUCTION

Computational models that are abstracted from the architecture and the functioning of living cells and its possible applications in modern computing are studied in an area of natural computing called membrane computing [17,18]. Under membrane computing, a kind of distributed and parallel-like neural-like computation model called spiking neural P systems (SN P systems) were proposed by [13], inspired by how neurons typically communicate with each other through sending of spikes between the synapses. For more information, some recent surveys on theory and main results are in [14], implementations and applications in [8], with a chapter on SN P systems in the handbook in [19]. Formally, SN P systems is defined as:

J. Caro et al. (eds.), *Proceedings of the Workshop on Computation: Theory and Practice (WCTP 2023)*, Atlantis Highlights in Computer Sciences 20,
https://doi.org/10.2991/978-94-6463-388-7_25

Definition 1 (SN P system). *A spiking neural P system (SN P system) of a finite degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out)$,where:*

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

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m, \text{ where:}$$

   (a) *$n_i \geq 0$ is the **initial number of spikes** contained by the neuron*
   (b) *R_i is a finite set of rules of the following forms:*
      (1) *$E/a^r \rightarrow a; t$, where E is a regular expression over O, $r \geq 1$ and $t \geq 0$*
      (2) *$a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule $E/a^r \rightarrow a; t$ of type (1) from R_i;*

3. *$syn \subseteq \{1, 2, ..., m\} \times \{1, 2, ..., m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (**synapses** among neurons);*
4. *$i_0 \in 1, 2, ..., m$ indicates the **output neuron***

Example 1. Let $\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \sigma_4, sys)$ where $\sigma_1 = (2, R_1)$ with $R_1 = \{a^2/a \rightarrow a, a \rightarrow \lambda\}$; $\sigma_2 = (1, R_2)$ with $R_2 = \{a \rightarrow a; 0, a \rightarrow a; 1\}$; $\sigma_3 = (3, R_3)$ with $R_3 = \{a^3 \rightarrow a; 0, a \rightarrow a; 1, a^2 \rightarrow \lambda\}$; and σ_4 is the output neuron (environment); $sys = \{(1, 2), (1, 3), (2, 1), (2, 3), (3, 4)\}$, visualized on Figure 1. The SN P system in Figure 1 generates the set $\{2n \mid n \geq 1\}$ or the set of all even natural numbers. For further details, including semantics of rule applications of SN P systems, the reader is referred to [13,19,14].

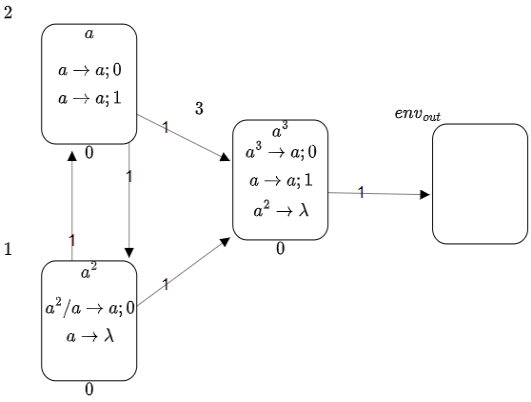

Fig. 1. SN P system that generates all even natural numbers.

To further study SN P systems, web-based simulators are used. WebSnapse, for example, was developed by [7] to serve as visual aid in showing how SN P systems work in a way that is easily understandable to users even without background

knowledge. WebSnapse was extended to improve user experience and include more variants of SN P systems, namely, with weighted synapses and input neurons [6]. However, both versions of the simulator suffered performance and stability issues in simulating more complex SN P systems [7][6]. Both versions also does not exactly follow the syntax in which rules are written in literature.

In this research, the intentions are to improve the performance and stability of WebSnapse v2.0 while achieving at least feature parity using tools such as WebAssembly and implement LaTeX support for rule input and visualization in WebSnapse.

2 MATRIX REPRESENTATION OF SN P SYSTEM

SN P Systems can be represented through matrices and vectors. This representation reduces the computation of configuration transitions to matrix operations. These are the parts of the representation for SN P systems with no delay as defined by [20]:

Definition 2 (Configuration Vectors). *Let Π be an SN P system with m neurons, the vector $C_0 = (n_1, n_2, ..., n_m)$ is called the* ***initial configuration vector*** *of Π, where n_i is the amount of the initial spikes present in neuron $\sigma_i, i = 1, 2, ..., m$ before a computation starts.*

In a computation, for any $k \in \mathbb{N}$, the vector $C_k = (n_1^{(k)}, n_2^{(k)}, ..., n_m^{(k)})$ is called the kth configuration vector of the system, where $n_i^{(k)}$ is the amount of spikes in neuron $\sigma_i, i = 1, 2, ..., m$ after the kth step of the computation.

In Example 1, the initial configuration vector $C^{(0)} = \begin{pmatrix} 2 & 1 & 3 & 0 \end{pmatrix}$.

Definition 3 (Spiking Vectors). *Let Π be an SN P system with m neurons and n rules, and $C_k = (n_1^{(k)}, n_2^{(k)}, ..., n_m^{(k)})$ be the kth configuration vector of Π. Assume that a total order $d : 1, ..., n$ is given for all n rules, so the rules can be referred to as $r_1, ..., r_n$. A* ***spiking vector*** *$s^{(k)}$ is defined as follows:*

$$s^{(k)} = (r_1^{(k)}, r_2^{(k)}, ..., r_n^{(k)}), \tag{1}$$

$$r_i^{(k)} = \begin{cases} 1, & \text{if the regular expression } E_i \text{ of rule } r_i \text{ is satisfied by the number of spiked } n_j^{(k)} \text{ (rule } r_i \text{ is in neuron } \sigma_j\text{) and rule } r_i \text{ is chosen and applied;} \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

In Example 1, the spiking vector $Sp^{(0)}$ could either be $\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$ or $\begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 \end{pmatrix}$ due to two rules on σ_2 being applicable to the amount of spikes contained.

Definition 4 (Spiking Transition Matrix). *Let Π be an SN P system with m neurons and n rules, and $d : 1, ..., n$ be a total order given for all the n rules. The* ***spiking transition matrix*** *of the system Π, M_Π, is defined as follows:*

$$M_\Pi = [a_{ij}]_{n \times m} \tag{3}$$

where:

$$a_{ij} = \begin{cases} -c, & \text{if rules } r_i \text{ is in neuron } \sigma_j \text{ and it is applied consuming } c \text{ spikes;} \\ p, & \text{if rule } r_i \text{ is in the neuron } \sigma_s \ (s \neq j \text{ and } (s, j \in syn)) \text{ and is applied producing } p \text{ spikes;} \\ 0, & \text{if rule } r_i \text{ is in neuron } \sigma_s \ ((s \neq j) \text{ and } (s, j) \notin syn) \end{cases}$$

In Example 1,

$$M_\Pi = \begin{pmatrix} -1 & 1 & 1 & 0 \\ -1 & 0 & 0 & 0 \\ 1 & -1 & 1 & 0 \\ 1 & -1 & 1 & 0 \\ 0 & 0 & -3 & 1 \\ 0 & 0 & -1 & 1 \\ 0 & 0 & -2 & 0 \end{pmatrix} \tag{4}$$

This matrix represents the spikes produced and consumed by the rules in the system.

To take into account rules with delay, [4] defined additional vectors such as:

Definition 5 (Delay Vector). *The vector $D = (d_1, d_2, ..., d_m)$ is the delay vector that indicates the amount of delay $d_i \geq 0$ of the rule r_i, for each $i = 1, 2, ..., m$*

In Example 1, D is $\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$.

Definition 6 (Loss Vector). *The vector $Lv^{(k)} = (lv_1, lv_2, ..., lv_m)$ is the loss vector where each lv_i for each neuron $\sigma_i, i = 1, 2, ..., m$, contains the number of spikes consumed, c, if σ_i applies r_i at time step k.*

Definition 7 (Gain Vector). *The vector $Gv^{(k)} = (gv_1, gv_2, ..., gv_m)$ is the gain vector which contains the total number of spikes gained, gv_i for each neuron $\sigma_i, i = 1, 2, ..., m$ at the kth step of computation not considering whether the neuron is open or closed.*

Definition 8 (Status Vector). *The vector $St^{(k)} = (st_1, st_2, ..., st_m)$ is called the status vector at the kth step of computation where each $st_i, i = 1, 2, ..., m$ determines the status of the neuron m*

$$st_i = \begin{cases} 1 & \text{if neuron } m \text{ is open} \\ 0 & \text{if neuron } m \text{ is closed} \end{cases} \tag{5}$$

Note that a neuron is said to be closed when a rule with a delay is activated and is waiting for that delay to become 0. A neuron that is closed may not receive any incoming spike.

Definition 9 (Indicator Vector). *The vector* $Iv^{(k)} = (iv_1, iv_2, ..., iv_n)$ *is called the indicator vector at the kth step of computation where each* $iv_i, i = 1, 2, ..., n$ *determines whether the rule* n *is fired at time step* k.

In Example 1, $Iv^{(0)}$ is $\begin{pmatrix}1&0&1&0&1&0&0\end{pmatrix}$ if the first rule in σ_2 is chosen. If the second rule is chosen, $Iv^{(0)}$ is $\begin{pmatrix}1&0&0&0&1&0&0\end{pmatrix}$, reflecting that the chosen rule is delayed in $t = 0$.

Definition 10 (Net Gain Vector). *Let* $Lv^{(k)}, Gv^{(k)}$ *and* $St^{(k)}$ *be the loss vector, gain vector, and status vector at time step* k*, respectively. The net gain vector* $NG^{(k)}$ *at time step* k *is defined as*

$$NG^{(k)} = St^{(k)} \odot GV^{(k)} + LV^{(k)} \tag{6}$$

Note that on the definition above, $St^{(k)}$ is updated first before computing $NG^{(k)}$. If it is to be defined that $St^{(0)} = \vec{1}$ since all neurons are open initially, then the net gain vector is defined instead as

$$NG^{(k)} = St^{(k+1)} \odot Gv^{(k)} + Lv^{(k)} \tag{7}$$

Referencing [4], [3] defined the following matrices:

Definition 11 (Production Matrix). *For an SN P system* Π *with* n *rules and* m *neurons, a production matrix of* Π *is given by*

$$PM_\Pi = [p_{ij}]_{n \times m}, \tag{8}$$

where for each rule $r_i : E/a^c \rightarrow a^p; d \in \sigma_j$*, we have*

$$p_{ij} = \begin{cases} p, & \text{if } r_i \in \sigma_s, s \neq j \text{ and } (s, j) \in syn \\ 0, & \text{otherwise} \end{cases} \tag{9}$$

Definition 12 (Consumption Matrix). *For an SN P system* Π *with* n *total rules and* m *neurons, a production matrix of* Π *is given by*

$$CM_\Pi = [c_{ij}]_{n \times m},$$

where for each rule $r_i : E/a^c \rightarrow a^p; d \in \sigma_j$*, we have*

$$c_{ij} = \begin{cases} c, & \text{if } r_i \in \sigma_j \text{ and consumed } c \\ & \text{spikes} \\ 0, & \text{otherwise} \end{cases}$$

With the above definitions, the following lemmas are shown

Lemma 1. *Let Π be an SN P system with delay, $d > 0$*

$$Gv^{(k)} = Iv^{(k)} \cdot PM_{\Pi} \tag{10}$$

Lemma 2. *Let Π be an SN P system with delay, $d > 0$*

$$Lv^{(k)} = -Sp^{(k)} \cdot CM_{\Pi} \tag{11}$$

Thus, the formula for $NG^{(k)}$ can also be defined as

$$NG^{(k)} = St^{(k+1)} \odot (Iv^{(k)} \cdot PM_{\Pi}) - Sp^{(k)} \cdot CM_{\Pi} \tag{12}$$

3 EXTENSION OF MATRIX REPRESENTATION OF SN P SYSTEMS WITH DELAY AND INPUT SPIKES FROM ENVIRONMENT

3.1 Algorithm for computing the next state of the system

In [3], the configuration vector at the next time step is computed by:

$$C^{(k+1)} = C^{(k)} + St^{(k+1)} \odot (Iv^{(k)} \cdot PM_{\Pi}) - Sp^{(k)} \cdot CM_{\Pi} \tag{13}$$

[3] also related the M_{Π} with PM_{Π} and CM_{Π}

Collorary 1. *For any SN P system Π*

$$PM_{\Pi} - CM_{\Pi} = M_{\Pi} \tag{14}$$

where M_{Π} is the spiking transition matrix of Π

What follows is a correction of the results from [3] where another way of computing the next configuration vector $C^{(k+1)}$ is proposed

Theorem 1. *Let Π be an SN P systems with delay, $d > 0$. Then for $k \geq 0$,*

$$C^{(k+1)} = C^{(k)} + St^{(k+1)} \odot (Iv^{(k)} \cdot M_{\Pi}) \tag{15}$$

Proof. Let Π be an SN P System with delay, $d > 0$. Given $C^{(k)}$ for some time step k, an indicator vector $Iv^{(k)}$ is specified to describe which rule is allowed to fire at time step k. Multiplying $Iv^{(k)}$ with PM_{Π} gives us the gain vector $Gv^{(k)}$ that represents the amount of spikes produced by the respective rules at time step k. Assume $St^{(0)} = \hat{1}$. To satisfy the condition that on time steps $q, q+1, ..., q+d-1$ where q is the time step where a rule with delay $d > 0$ is chosen, incoming spikes should be ignored; the gain vector should be multiplied by the next status vector $St^{(k+1)}$.

Assuming that spiking and spike consumption of a rule should be at the same time step, multiplying $Iv^{(k)}$ with CM_{Π} gives us the loss vector $Lv^{(k)}$ that represents

the amount of spikes consumed at time step k. Note that if $St_i^{(k+1)} = 0, Lv_i^{(k)} = 0$ as well. Since spikes on a neuron are only consumed once it opens from a closed state, no spikes are consumed on time steps $q, q+1, ..., q+d-1$ on a closed neuron, where q is the time step where a rule with delay $d > 0$ is chosen. Formally, the elements of $Lv^{(k)}$ are:

$$Lv_i^{(k)} = \begin{cases} 0, & \text{if } St_i^{(k+1)} = 0 \\ Lv_i^{(k)}, & \text{otherwise} \end{cases}, \forall i, 1 \leq i \leq n \tag{16}$$

where n is the number of neurons.

By this definition, we can multiply $St^{(k+1)}$ element-wise with $Lv^{(k)}$ and we should still get $Lv^{(k)}$:

$$St_i^{(k+1)} \times Lv_i^{(k)} = \begin{cases} St_i^{(k+1)} \times 0, & \text{if } St_i^{(k+1)} = 0 \\ St_i^{(k+1)} \times Lv_i^{(k)}, & \text{otherwise} \end{cases} = \tag{17}$$

$$\begin{cases} 0 \times 0 & \text{if } St_i^{(k+1)} = 0 \\ 1 \times Lv_i^{(k)}, & \text{otherwise} \end{cases} = \begin{cases} 0, & \text{if } St_i^{(k+1)} = 0 \\ Lv_i^{(k)}, & \text{otherwise} \end{cases}$$

Thus, we can say that $Lv^{(k)} = St^{(k+1)} \odot (Iv^{(k)} \cdot CM_\Pi)$ and we get the following net gain vector:

$$NG^{(k)} = St^{(k+1)} \odot (Iv^{(k)} \cdot PM_\Pi) - St^{(k+1)} \odot (Iv^{(k)} \cdot CM_\Pi) \tag{18}$$

By the distributive property of Hadamard multiplication and matrix multiplication, both the next status vector and indicator vector can be factored out, resulting in $St^{(k+1)} \odot (Iv^{(k)} \cdot (PM_\Pi - CM_\Pi))$. Substituting $PM_\Pi - CM_\Pi$ with M_Π from Corollary 4, and then adding the net gain vector to the current configuration vector, the following equation is obtained:

$$C^{(k+1)} = C^{(k)} + St^{(k+1)} \odot (Iv^{(k)} \cdot M_\Pi) \tag{19}$$

■

Using this method, the amount of computation needed is reduced since the matrix multiplication needed is reduced to one instead of two. Also note that in the proposed method of [4], the spike consumption of a rule happens first then, after some delay, follows the production of spikes. In contrast, production and consumption of spikes happens at the same time in this method. Ultimately, the results will be the same since the spikes inside a closed neuron cannot change.

Example 2. Using Example 1 and choosing the 2nd rule in σ_2, the computation of the next configuration vector is as follows:

$$C^{(1)} = C^{(0)} + St^{(1)} \odot (Iv^{(0)} \cdot M_{\Pi}) = \begin{pmatrix}2&1&3&0\end{pmatrix} + \begin{pmatrix}1&0&1&1\end{pmatrix} \odot (\begin{pmatrix}1&0&0&0&1&0&0\end{pmatrix} \cdot \begin{pmatrix}-1&1&1&0\\-1&0&0&0\\1&-1&1&0\\1&-1&1&0\\0&0&-3&1\\0&0&-1&1\\0&0&-2&0\end{pmatrix}) = \begin{pmatrix}2&1&3&0\end{pmatrix} + \begin{pmatrix}-1&0&-2&1\end{pmatrix} = \begin{pmatrix}1&1&1&1\end{pmatrix} \quad (20)$$

To compute the indicator vector $Iv^{(k)}$, the following vectors are defined:

Definition 13 (Decision Vector). *The vector $Dcs^{(k)} = (dcs_1, dcs_2, ..., dcs_n)$, where n is the total number of rules in the system, is the decision vector that indicates whether a rule is chosen/applied at time step k.*

Note that this is only a renaming of the spiking vector $Sp^{(k)}$ since it makes more sense in the context of systems with delays

Definition 14 (Delay Indicator Vector). *The delay indicator vector $Div(k) = (div_1, div_2, ..., div_n)$, where n is the total number of rules in the system, is defined as*

$$div_i = \begin{cases} 1 & \text{if } r_i \text{ is scheduled to fire at some time } d \\ 0 & \text{otherwise} \end{cases} \quad (21)$$

Definition 15 (Delay Status Vector). *The delay status vector $Dst^{(k)} = (dst_1, dst_2, ..., dst_m)$, where m is the number of neurons in the system, indicates the amount of delay $dst_i \geq 0$ the neuron σ_i has at time step k*

Note that this is a redefinition of what is defined by [3] from rule-wise to neuron-wise to avoid redundancy.

Using the above definitions, these are the algorithms to obtain the next state of the system. Note that the pseudocode is vectorized, if possible, to take advantage of parallel computing.

Algorithm 1 Compute Indicator Vector

Require: $Dst^{(k+1)}, Dcs^{(k)}, D, Div^{(k)}$
1: **procedure** getIndicatorVector
2: $r \leftarrow 0$
3: **for** $i \leftarrow 0$ to COUNT(σ) in Π **do**
4: $count \leftarrow$COUNT(R of σ_i)
5: $Iv_{(k)}[r : r + count] \leftarrow (Dst_i^{(k+1)} = 0)$ and $((Dcs^{(k)}[r : r + count] = \vec{1}$ and $D[r : r + count] = \vec{0})$ or $Div^{(k)}[r : r + count] = \vec{1})$
6: $r \leftarrow r + count$
7: **end for**
8: **end procedure**

If the delay status vector becomes 0, it means that the neuron becomes open and can fire a rule. If a rule is chosen and it has no delay, it can immediately fire. Also, if there is a delayed rule and it becomes open, that delayed rule is to fire.

Algorithm 2 Compute Next Delay Status Vector

Require: $Dcs^{(k)}, D, Dst^{(k)}$
procedure GETNEXTDELAYSTATUSVECTOR
  $r \leftarrow 0$
  for $i \leftarrow 0$ to COUNT(σ) in Π **do**
    $d \leftarrow 0$
    $count \leftarrow$COUNT(R of σ_i)
    for $j \leftarrow 0$ to $count$ **do**
      if $Dcs_{r+j}^{(k)} = 1$ **then**
        $d \leftarrow D_{r+j}$
        break
      end if
    end for
    $Dst_i^{(k+1)} = Dst_i^{(k)} > 0 \ ? \ Dst^{(k)} - 1 : d$
    $r \leftarrow r + count$
  end for
end procedure

This procedure iterates over $Dcs^{(k)}$ to check which rule is chosen for each neuron, if any, and note the delay of that rule. Then, if the delay status $dst_i > 0$ (meaning that the neuron is closed), decrement dst_i. Otherwise, dst_i becomes the delay of the chosen rule.

Algorithm 3 Compute Next Delay Indicator Vector

Require: $Div^{(k)}, Iv^{(k)}, Dcs^{(k)}, D$
procedure GETNEXTDELAYINDICATORVECTOR
  $Div^{(k+1)} \leftarrow (Div^{(k)}$ and not $Iv^{(k)})$ or $(Dcs^{(k)}$ and $D > 0)$
end procedure

The delay indicator div_i is retained in the next time step if it is not yet fired. div_i also becomes 1 if the corresponding rule r_i is chosen and it has a delay $d > 0$.

These procedures are used in the following order to get the next state.

Algorithm 4 Compute Next State of System

```
1: procedure GETNEXT
2:     call Get Dst^(k+1)
3:     call Get Iv^(k)
4:     call Get C^(k+1)
5:     call Get Div^(k+1)
6: end procedure
```

3.2 Algorithm for input spikes from environment

SN P systems can also function by taking input spikes from the environment as spike trains.

Example 3. Let $\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \sigma_4, sys)$ where σ_1 and σ_2 are input spike trains; $\sigma_3 = (0, R_2)$ with $R_2 = \{a^3 \to a; 0, a \to a; 1; a^2 \to \lambda\}$; and σ_3 is the output neuron (environment); $sys = \{(1,3), (2,3), (3,4)\}$, visualized in Figure 2

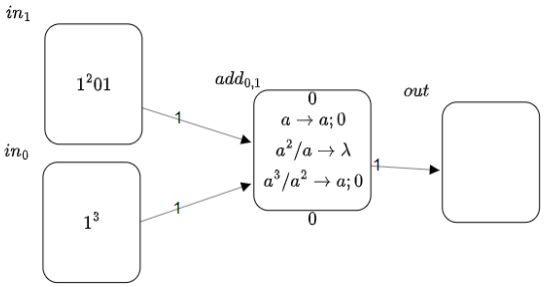

Fig. 2. SN P System that adds two binary numbers

To take these systems into consideration, input nodes are treated as a regular neuron with the following rule:

$$R = (\lambda \to a; 0) \tag{22}$$

Additionally, a new vector is introduced.

Definition 16 (Spike Train Vector). *The vector* $Spt^{(k)} = (spt_1, spt_2, ..., spt_n)$ *is called the spike train vector, which is defined at the kth step as*

$$spt_i = \begin{cases} ST_k^j & \text{if } r_i \text{ is in } \sigma_j \\ 0 & \text{otherwise} \end{cases} \tag{23}$$

where ST_k^j *is the kth element of the input spike train defined in* σ_j

The spike train vector acts as the indicator vector for rules that function for input spikes from the environment. For illustration, in Example 3, $Spt^{(0)} = \begin{pmatrix}1 & 1 & 0 & 0 & 0\end{pmatrix}$. Thus, Theorem 1 is modified as:

Theorem 2. *Let Π be an SN P system with delay, $d > 0$, and input spike train vector $Spt^{(k)}$ at time k. Then, for $k \geq 0$,*

$$C^{(k+1)} = C^{(k)} + St^{(k+1)} \odot ((Iv^{(k)} + Spt^{(k)}) \cdot M_{\Pi}) \tag{24}$$

Example 4. Using the SN P System from Example 3. The computation of the next configuration vector is as follows:

$$C^{(1)} = C^{(0)} + St^{(1)} \odot ((Iv^{(0)} + Spt^{(0)}) \cdot M_{\Pi}) = \begin{pmatrix}0 & 0 & 0 & 0\end{pmatrix} + \begin{pmatrix}1 & 1 & 1 & 1\end{pmatrix} \odot ((\begin{pmatrix}0 & 0 & 0 & 0 & 0\end{pmatrix} + \begin{pmatrix}1 & 1 & 0 & 0 & 0\end{pmatrix}) \cdot \begin{pmatrix} 0\,0 & 1 & 0 \\ 0\,0 & 1 & 0 \\ 0\,0 & -1 & 1 \\ 0\,0 & -1 & 0 \\ 0\,0 & -2 & 1 \end{pmatrix}) = \begin{pmatrix}0 & 0 & 0 & 0\end{pmatrix} + \begin{pmatrix}0 & 0 & 2 & 0\end{pmatrix} = \begin{pmatrix}0 & 0 & 2 & 0\end{pmatrix} \tag{25}$$

3.3 Algorithm for computing the previous state of the system

By transposing Theorem 2 and moving it one time step back, the following can be derived

Collorary 2. *Let Π be an SN P system with delay $d > 0$. Then for $k \geq 1$*

$$C^{(k-1)} = C^{(k)} - St^{(k)} \odot (Iv^{(k-1)} + Spt^{(k-1)} \cdot M_{\Pi}) \tag{26}$$

To obtain the previous delay status vector $Dst^{(k-1)}$, the following algorithm is used.

Algorithm 5 Compute Previous Delay Status Vector

```
1: procedure getPreviousDelayStatusVector
2:     r ← 0
3:     for i to COUNT(σ in Π) do
4:         count ← COUNT(R of σ_i)
5:         for j to count do
6:             if Div_{r+j}^{(k-1)} = 1 then
7:                 Dst_i^{(k-1)} ← Dst_i^{(k)} + 1
8:                 go to nextNeuron
9:             end if
10:        end for
11:        Dst_i^{(k-1)} ← 0
12:        nextNeuron: r ← r + count
13:    end for
14: end procedure
```

If a rule r_i is previously scheduled to fire, it means that the neuron σ_j that contains it had a delay; thus, the delay status dst_j is incremented. Otherwise, $dst_j = 0$. The use of **goto** is to make it clear that the loop has two exits based on a condition, but it can also be done with having a flag that is set before exiting the loop or having the loop in a function and **return** accordingly.

To get the previous indicator vector $Iv^{(k-1)}$, Algorithm 1 is sufficient except the vectors from the previous state is used.

$$Iv^{(k-1)} = \text{call Get}Iv^{(k)}(Dst^{(k)}, Dcs^{(k-1)}, D, Div^{(k-1)}) \tag{27}$$

Using the above, the stacks needed are for the decision vector $Dcs^{(i)}$, $Div^{(i)}$ for $i \leq k-1$. Stacks can also be implemented for $Iv^{(i)}$ and $Dst^{(i)}$ for $i \leq k-1$ and it would perform better since it would only be $O(1)$. However, it would also need more memory than when these vectors are computed instead.

4 IMPLEMENTATION

Since matrix representation and algorithms are used, it would be advantageous to parallelize the computation of state vectors. One method of parallel computing available in the browser is SIMD.

SIMD (Single Instruction, Multiple Data) is a type of parallel computing that exploits data parallelism by simultaneously performing the same operation on multiple data elements. It works well in this case considering that matrix multiplication involves multiple dot products, which are vector operations. The source code for WebSnapse v3, including test files and previous versions of WebSnapse, can be found at the WebSnapse page [1].

4.1 WebAssembly

To use SIMD in the browser, WebAssembly (WASM) is used, as it defines a portable, performant subset of SIMD operations that are available on most modern architectures [9].

Algorithms 1, 2, 3, 4, 5, Theorem 2 and Corollary 2 are implemented in C and compiled into WASM using Emscripten. Using the O3 and msimd128 compilation flag, operations that could be parallelized such as for-loops are vectorized when compiled.

4.2 Front-end

The front-end is built using the MVP architecture (Figure 3). The models are the SN P system, which is the model configured for the neurons, rules, and other parts of the system; the Simulator, which keeps track of the state and updates of the simulation; and the System JSON, which imports JSON to System model and export System model to JSON. The views are the Graph, which is graph representation of the SN P System; and Input, which is the view for the input elements of the simulator. The

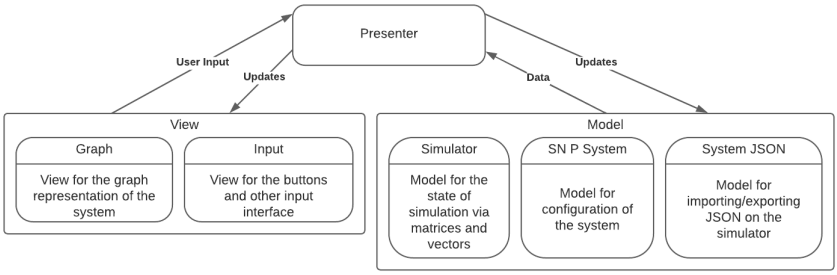

Fig. 3. MVP Architecture for WebSnapse v3

presenter determines how user input is handled and updates the models accordingly. The presenter also updates the view based on the models.

This is implemented with HTML, CSS, and TypeScript. For the libraries, VivagraphJS is used for graph visualization, KaTeX for HTML TeX rendering, and MathJax for SVG TeX rendering. The web application is hosted on Netlify with link available at the WebSnapse page in [1].

Necessary vectors and matrix are generated in JavaScript based on the built or imported system. Since it is necessary for SIMD to have sequential data access, the spiking transition matrix is transposed so that the matrix multiplication becomes row-to-row dot products instead of row-to-column. These vectors are then passed to the compiled WASM module.

4.3 WebSnapse V3.0 Features

Apart from performance and runtime improvements, Websnapse v3.0 also boasts some quality of life improvements. Features from the previous version such as the button sidebar, Import/Export capability, clear All button, simulation controls, and support for SN P system variants are retained. Some user-friendly additions such as right-clickability of neurons and synapses were added for a more intuitive approach. Here are the specific features:

Delete and Edit Synapses. Editing and deleting synapses can now be done by right-clicking on a synapse and editing the weight once the menu appears. In Websnapse v2.0, editing synapses was also possible, but the user had to click the synapse, click the Edit Synapse button, and then edit the synapse weight. The extra step proved to be quite time-consuming when editing a larger SN P system. Also, right-clicking for options is more user-friendly and intuitive.

Delete and Edit Neurons. Similarly, editing and deleting neurons work the same way as for synapses. This was another user-friendly improvement over the previous version.

LaTeX Syntax Support Neuron names, rules, and spike train input all follow the LaTeX format to better align with how these are written in literature.

JSON File Format Websnapse v3.0 uses JSON file format, collaborated with [10], vs the XMP file format of Websnapse v2.0. Any JSON file that follows the format will be properly simulated when imported.

5 TESTING

Testing WebSnapse v3 is divided into two parts: the correctness of the simulations and the performance benchmarks. These tests are run on Microsoft Edge Version 118.0.2088.46 (official build) (64-bit). The machine used runs Windows 10 Build 19045 with an Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz, 1801 Mhz, 4 Core(s), 8 Logical Processor(s) and 8 GB RAM.

5.1 Correctness Testing

The following systems are run on WebSnapse v3 and have been confirmed to be properly simulated.

Multiples of k Generator

These systems generate a number multiple of k by initially firing a spike to the environment, and after a certain delay, firing the following spike. The time difference between these spikes is the output of the system to be interpreted. When to fire the following spike is non-deterministically chosen [18].

Boolean Function

These systems are used as transducer, receiving spike trains from the environment and after 3 time steps delay, the output neuron will fire the output of the Boolean function as a spike train to the environment [18].

Comparator

An increasing comparator represented as an SN P system is proposed by [5]. Two bit strings as spike trains are fired into the system and the smaller of the two bit strings is fired to the *min* output and the larger to the *max* output.

Bit Adder

This SN P system proposed by [12] works as a bit adder that accepts two bit strings as spike trains and, after a one-time-step delay, fires the sum to the environment.

Subset Sum

A uniform SN P system is proposed by [15] that solves the subset sum problem. Numbers in the set are written multiplied by two on the set of neurons labeled with $in_i, 1 \leq i \leq LEN(set)$. The sum is also multiplied by two on neuron $i_{LEN(set)+1}$.

The system will only halt if a solution (a subset that satisfies the problem) is found; otherwise, the system keeps running.

5.2 Benchmark

To gauge Websnapse v3.0 performance and runtime improvements, a baseline test consisting of a single sequential system and a single parallel system was used. These systems are generated in JSON format (for WebSnapse v3) and XML format (for WebSnapse v2) using Python (https://github.com/lmgal/websnapse-stress-test) and imported to each simulator, respectively. The one spike-chain system in Fig. 4 was used for the sequential system benchmark, while the simple complete graph system in Fig. 5 was used for the parallel system benchmark, both figures show systems with only 10 neurons as a sample.

Console timers were added to measure the time it took the simulator to compute the next configuration vector, as well as the time to render it. Neuron count is gradually increased by increments of 20, up to a maximum of 100. The average value of five individual benchmarks for both systems were taken as the final value.

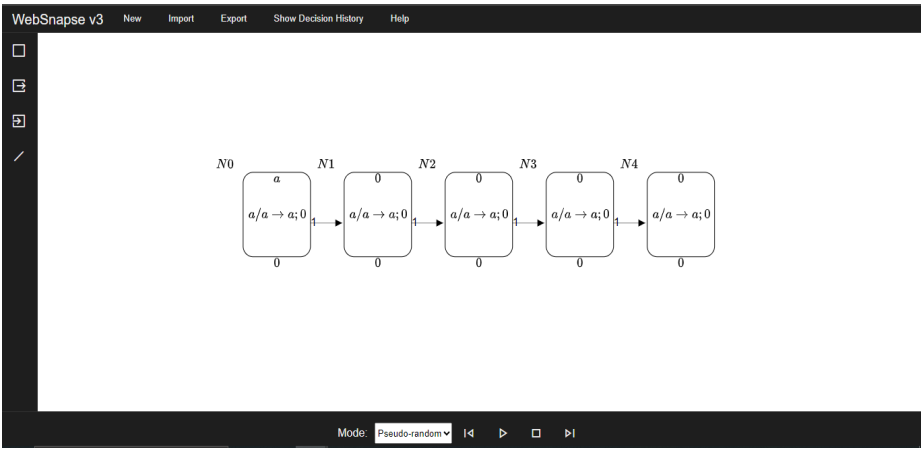

Fig. 4. Sample One Spike-Chain system in WebSnapse v3

The following benchmark results show the observed computation and rendering times for 20 neurons up to 100, for simplicity the results of the 100-node systems will be the point of comparison as this demands the most out of the simulators.

One Spike-Chain System

Websnapse v3.0 was able to compute the next configuration vector in the one spike-chain system in 0.887 milliseconds while Websnapse v2.0 took 18.403 ms as can be seen in Fig. 6. Fig. 7 shows that rendering took 42.281ms in v3.0, while v2.0 took

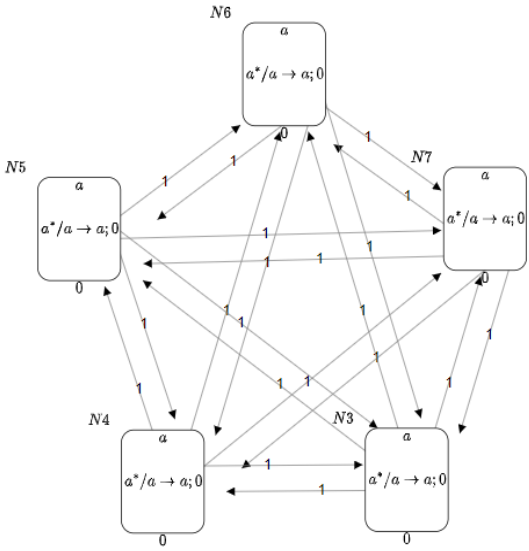

Fig. 5. Sample Simple Complete System

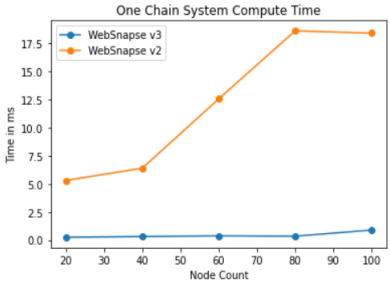

Fig. 6. One Spike-Chain system Computation Time Comparison

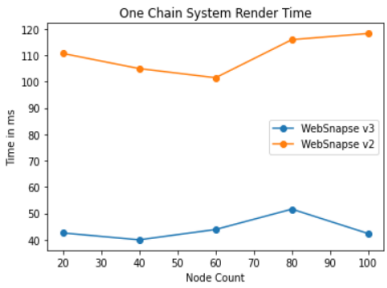

Fig. 7. One Spike-Chain system Rendering Time Comparison

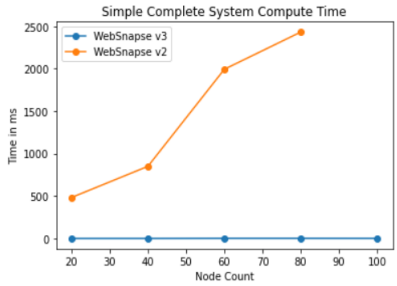

Fig. 8. Simple Complete system Computation Time Comparison

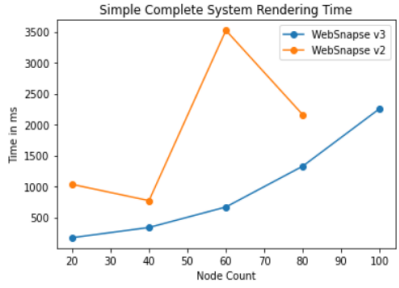

Fig. 9. Simple Complete system Computation Time Comparison

118.321ms. For a one spike-chain system, each neuron is connected to only two other neurons, the previous and the next neuron in the spike chain. This leads to lower computation and rendering time compared to complete simple systems. The upward trend of computation time and rendering time can be seen in both figures, as the node count or total number of neurons in the system increases. Fig. 6 and Fig. 7 show significant improvements in performance and runtime for Websnapse v3.0 versus the previous version when running a sequential SN P system.

Simple Complete System

Websnapse v3.0 was able to compute the next configuration vector in the simple complete system in an average time of 1.376 milliseconds while Websnapse v2.0 ceased computation and rendering at 100 neurons Fig. 8. Fig. 9 shows that the rendering took 2257.227 ms in v3.0. Since a simple complete graph with the same number of neurons as a one spike-chain system contains an exponentially larger number of synapses, it is expected that computation and rendering time will increase. Similarly for the computation time, Fig. 8 shows that as the number of neurons in the system increases, Websnapse v3.0 yields a far lower average computation time than the previous version; Websnapse v2.0 was unable to process the computation for a simple complete system with 100 neurons. For rendering time, an exponential increase is observed in Websnapse v3.0 while Websnapse v2.0 failed to render.

6 CONCLUSION

The main goal of Websnapse v3.0 is to create a SN P system simulator that is performant and intuitive. Websnapse v2.0 already boasted such features but performance and stability-wise could still use some improvements. The implementation of Websnapse v3.0 using WebAssembly (WASM) with single instruction, multiple data (SIMD) allowed it to parallelize the computation of state vectors. Parallelization of this process made use of the matrix representation in 2 specifically for the algorithms used in computation for the states.

WebSnapse v3.0 was successful in creating an updated version that fits the specification for WebSnapse while providing performance and stability improvements.

As can be seen in Figures 6, 7, 8, 9, Websnapse v3.0 shows significant improvement in the computation and rendering time for systems that are either sequential or parallel over the previous version while also allowing simulations of up to 100 neurons or more which Websnapse v2.0 was unable to achieve.

Websnapse v3.0 also improves usability and user-friendliness by including LaTeX syntax support, a new interface with more intuitive buttons that includes right-click capabilities for editing and deleting neurons and synapses.

Overall, the development of WebSnapse v3.0 was a success. However, there are several things that can be improved. To render higher neuron count, implementing an alternative renderer can be explored such as rasterizing SVG to WebGL canvas or simplifying the rendering load such as rendering regular neurons as circles with a number instead to represent the number of spikes. Adding more quality-of-life

features, such as selecting multiple neurons or synapses to move, delete, or duplicate, would also be nice. Implementing other SN P system variants on web simulators with matrix representation can also be explored. Improvements introduced in the present work can also be extended to the massively parallel simulators as in [4,2] and more recently in [11], as well as in hybrid simulators (combining web browsers and GPU access) in [16].

ACKNOWLEDGEMENTS

The authors thank Jarred Luzada, Mutya Gulapa, and Daryll Ko for some technical aspects of our respective simulators and JSON file format. F.G.C. Cabarle was supported by the Dean Ruben A. Garcia PCA, and Project No. 222211 ORG from the Office of the Vice Chancellor for Research and Development, both from the University of the Philippines Diliman. F.G.C. Cabarle is also supported by *QUAL21 008 USE* project (PAIDI 2020 and FEDER 2014-2020 funds). H.N. Adorna would like to thank the support from DOST-ERDT research grants; Semirara Mining Corp. Professorial Chair for Computer Science of College of Engineering, UPDiliman; RLC grant from UPD-OVCRD.

REFERENCES

1. Websnapse page. https://aclab.dcs.upd.edu.ph/productions/software/websnapse.
2. Blaine Corwyn D Aboy, Edward James A Bariring, Jym Paul Carandang, Francis George C Cabarle, Ren Tristan De La Cruz, Henry N Adorna, and Miguel Ángel Martínez-del Amor. Optimizations in cusnp simulator for spiking neural p systems on cuda gpus. In *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 535–542. IEEE, 2019.
3. Henry N Adorna. Matrix representations of spiking neural p systems: Revisited. *arXiv preprint arXiv:2211.15156*, 2022.
4. Jym Paul Carandang, John Matthew B Villaflores, Francis George C Cabarle, Henry N Adorna, and Miguel Ángel Martínez del Amor. Cusnp: Spiking neural p systems simulators in cuda. *Romanian Journal of Information Science and Technology (ROMJIST), 20 (1), 57-70.*, 2017.
5. Rodica Ceterchi and Alexandru Ioan Tomescu. Spiking neural p systems–a natural model for sorting networks. *Proceedings of the Sixth Brainstorming Week on Membrane Computing, 93-105. Sevilla, ETS de Ingeniería Informática, 4-8 de Febrero, 2008*, 2008.
6. Nathan Cruel, Coleen Quirim, and Francis George C Cabarle. Websnapse v2.0: Enhancing and extending the visual and web-based simulator of spiking neural p systems. In *Pre-proceedings of the 11th Asian Conference on Membrane Computing, Quezon City, Philippines*, pages 146–166, September 2022.
7. Annysia Glynis S Dupaya, Anica Clarice Antonella P Galano, Francis George C Cabarle, Ren Tristan De La Cruz, Korsie J Ballesteros, and Prometheus Peter L Lazo. A web-based visual simulator for spiking neural p systems. *Journal of Membrane Computing*, 4(1):21–40, 2022.
8. Songhai Fan, Prithwineel Paul, Tianbao Wu, Haina Rong, and Gexiang Zhang. On applications of spiking neural p systems. *Applied Sciences*, 10(20):7011, 2020.

9. Deepti Gandluri, Thomas Lively, and Ingvar Stepanyan. Fast, parallel applications with webassembly simd. https://v8.dev/features/simd, 2020. Accessed: October 1, 2023.
10. Mutya Gulapa, Jarred Sueo Luzada, Francis George C. Cabarle, Henry Adorna, Kelvin Buño, and Daryll Ko. Websnapse reloaded: The next-generation spiking neural p system visual simulator using client-server architecture. In Shigeki Hagihara, Shin ya Nishizaki, Masayuki Numao, Jaime Caro, and Merlin Teodosia Suarez, editors, *Pre-proc. 12th Workshop on Computation: Theory and Practice (WCTP2023), 4 to 6 December 2023, Chitose-city, Hokkaido, Japan*, pages 511–536, 2023.
11. Rogelio V Gungon, Katreen Kyle M Hernandez, Francis George C Cabarle, Ren Tristan A De la Cruz, Henry N Adorna, Miguel Á Martínez-del Amor, David Orellana-Martín, and Ignacio Pérez-Hurtado. Gpu implementation of evolving spiking neural p systems. *Neurocomputing*, 503:140–161, 2022.
12. Miguel A Gutiérrez-Naranjo and Alberto Leporati. Performing arithmetic operations with spiking neural p systems. *Proc. of the Seventh Brainstorming Week on Membrane Computing*, 1:181–198, 2009.
13. Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural p systems. *Fundamenta informaticae*, 71(2-3):279–308, 2006.
14. Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. Spiking neural p systems: main ideas and results. *Natural Computing*, 21(4):629–649, 2022.
15. Alberto Leporati, Giancarlo Mauri, Claudio Zandron, Gheorghe Păun, and Mario J Pérez-Jiménez. Uniform solutions to sat and subset sum by spiking neural p systems. *Natural computing*, 8(4):681–702, 2009.
16. Ayla Nikki L Odasco, Matthew Lemuel M Rey, and Francis George C Cabarle. Improving gpu web simulations of spiking neural p systems. *Journal of Membrane Computing*, pages 1–16, 2023.
17. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
18. Gheorghe Paun. Membrane computing. *Scholarpedia*, 5(1):9259, 2010.
19. Gheorghe Păun, Gzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing*. Oxford Univeristy Press, 2010.
20. Xiangxiang Zeng, Henry Adorna, Miguel Ángel Martínez del Amor, Linqiang Pan, and Mario J. Pérez-Jiménez. Matrix representation of spiking neural p systems. In *Membrane Computing*, pages 377–391. Springer Berlin Heidelberg, 2010.