WebSnapse Reloaded: The Next-Generation Spiking Neural P System Visual Simulator using Client-Server Architecture

Mutya Gulapa[1], Jarred Sueño Luzada[1], Francis George C. Cabarle[1,2], Henry N. Adorna[1], Kelvin Buño[1], and Daryll Ko[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 Systems) replicate the brain's information spiking mechanism through synapses. These systems are known for trading memory for time, making them useful in different areas both inside and outside computer science. While web-based simulators like WebSnapse offer the potential for centralized GUIs to simulate SN P system variants, their architecture, codebase, and technologies limit their capabilities. This paper presents WebSnapse Reloaded, a recreated version designed for centralization, optimization, and improved user and developer experiences. With a client-server architecture, it enhances storage usage, scalability, and future development possibilities. The modular codebase simplifies extension and maintenance. Optimizations include algorithm refinement through matrix representation and enhanced user experience. WebSnapse Reloaded reduces task steps, improves software implementation, passes correctness benchmarks, and resolves previous issues. Future recommendations include integrating WebGL or alternative graphing libraries, and developing APIs for other SN P variants to streamline simulator extension.

Keywords: membrane computing, spiking neural p systems, visual simulator, client-server architecture

1 Introduction

Membrane computing is a field of computation inspired by the structure and function of biological cells [27]. It represents computation as the behavior of a network of interacting "membranes" that can move, divide, and merge. P Systems, the computing models in membrane computing, are known for trading memory for time [10] to solve computationally hard problems. Under these are

The original version of the chapter has been revised. The symbol "ã" in the first line of the abstract was removed. A correction to this chapter can be found at https://doi.org/10.2991/978-94-6463-388-7_27

© The Author(s) 2024, corrected publication 2024

Spiking Neural P System (SN P System), inspired by the workings of the human brain and imitates the way neurons work [27].

Several simulators were developed to address the issue of manually tracing the system configuration for each time step. Existing GUI simulators for SN P Systems such as [9], [7], [6], laid the foundations for making these concepts more accessible. However, certain gaps were found in WebSnapse v2 concerning its architecture, computational method, code base organization, and user experience. To address these issues, the researchers created WebSnapse Reloaded to improve the performance, extensibility, and both the developer and user experience of the application. We note that despite the performance improvements in this present work, our simulator WebSnapse Reloaded was developed as a GUI simulator as defined in [28]. It is beyond the scope of WebSnapse reloaded to match simulation engines or parallel simulators as in [22, 5].

The following sections further elaborate the premise and process of the research. Section 2 formally defines SN P systems and its matrix representation. Section 3 provides an overview of this paper's predecessor, WebSnapse v2 and discusses the scope and contributions of the study. Section 4 presents WebSnapse Reloaded in detail: its features and functionalities, enhancements and testing procedures. Section 5 further tests the simulator's correctness through the two case studies provided. Lastly, the conclusions and future recommendations can be found in Section 6.

2 SN P Systems and its Matrix Representation

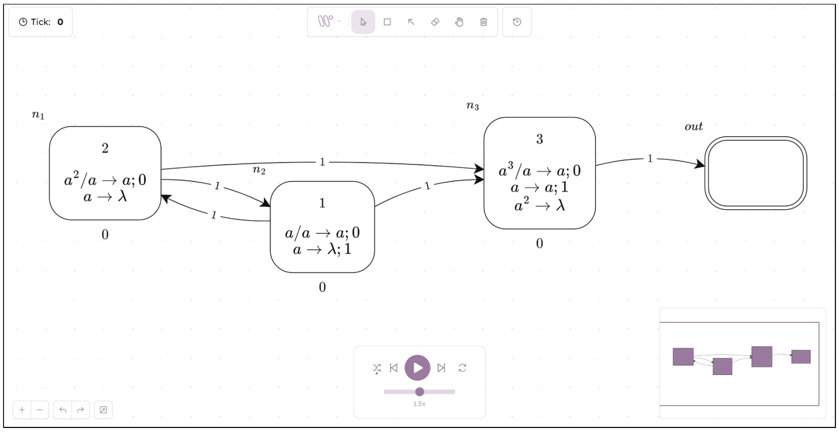

Fig. 1. An SN P System that generates all positive integers greater than 1 represented in WebSnapse Reloaded

SN P System Spiking Neural P Systems, or SN P systems for short, is a branch under membrane computing [23, 24] inspired by the human brain and models how neurons work [26]. It is a group of neurons placed in the nodes of a directed graph and communicate by spiking information along the arcs [13]. For further details on the main ideas and recent results on theory the reader may consult [15], for applications in [8], with a dedicated chapter in the handbook in [25]. To visualize SN P systems better, an example by [15] is represented in WebSnapse Reloaded found at Figure 1, which shows an SN P System that generates all positive integers greater than 1. Further, [26] formally defines an SN P System as follows:

Definition 1 (SN P system). *A computing extended Spiking Neural P system of degree $m \geq 1$, is a construct of the form:*

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

where:

1. *$O = \{a\}$ is the singleton alphabet (a is called* ***spike****).*
2. *$\sigma_1, \ldots, \sigma_m$ are neurons of the form $\sigma_i = (n_i, \mathcal{R}_i), 1 \leq i \leq m$, where:*
   - *(a) $n_i \geq 0$ is the* ***initial number of spikes*** *contained in σ_i*
   - *(b) $\mathcal{R}_i$ is a finite set of rules of the following two forms:*
     - *i. $E/a^c \to a^p; d$ where E is a regular expression over a and $c \geq p \geq 1$, $d \geq 0$;*
     - *ii. $a^s \to \lambda$, for $s \geq 1$, with the restriction that for each rule, $E/a^c \to a^p; d$ of type (i) from $\mathcal{R}_i$, we have $a^s \notin L(E)$*
3. *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, $(i, i) \notin syn$ for $1 \leq i \leq m$ (****synapses*** *between neurons);*
4. *$in, out \in \{1, 2, \ldots, m\}$ indicate the* ***input*** *and* ***output*** *spike train neurons.*

An SN P System is a model that runs under a global clock, which synchronizes the functions of all neurons letting them work in parallel. Each neuron contains a finite set of rules that can be either firing or forgetting. Firing rules trigger the sending of spikes to adjacent neurons when the regular expression, E is satisfied, while forgetting rules empty the neuron if it contains exactly s spikes. If the chosen firing rule has a specific delay, the neuron becomes closed, making it unable to send and receive new spikes. If more than one firing rule is applicable in a neuron at a time, only one rule is chosen non-deterministically.

Output spike train neurons receive the spikes sent by the system to the environment, generating a sequence of 1's and 0's called spike train. More advanced

SN P systems can integrate input spike train neurons [14] and weighted synapses to simplify previously larger systems by reducing the number of neurons needed to render the same results [21].

SN P systems serve many functions. Among these are simulating logical gates and circuits and executing a sorting algorithm [14]. They are Turing-complete [13] and are also able to solve NP-complete, or computationally hard, problems [16]. SN P systems are also useful in fields outside of computer science, e.g.: biology, ecology, economics and linguistics [10].

Matrix Representation of SN P Systems The Matrix representation of SN P Systems was first conceptualized by [29] but the model only applies to systems without delay. This work was then revisited by [3] which extended the representation to apply to systems with delay. Their model was used for the implementation of the simulation engine and is described in the following manner:

Let Π be an SN P System with m neurons and n rules. Its representation was covered in [3] which can be broken down into the following vectors and matrices:

Definition 2 (Configuration Vector). *This vector contains the number of spikes that each neuron has. $C^0 = \langle c_1, c_2, \ldots, c_m \rangle$ is called the **initial configuration vector** which contains the number of spikes present in each neuron before the computation starts. The **Configuration Vector** of Π at succeeding time steps k is then defined as:*

$$C^{(k)} = \langle c_1^{(k)}, c_2^{(k)}, \ldots, c_m^{(k)} \rangle$$

where $c_i^{(k)}$ is the amount of spikes that neuron σ_i contains for $i = 1, 2, \ldots, m$.

Definition 3 (Spiking Transition Matrix). *This matrix describes the relationship of each neuron with regard to their rules. The **Spiking Transition Matrix** is defined as:*

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

where:

$$a_{ij} = \begin{cases} -c & \text{if neuron } \sigma_j \text{ consumes } c \text{ spikes in rule } r_i; \\ p & \text{if neuron } \sigma_j \text{ produces } p \text{ spikes in rule } r_i; \\ 0 & \text{if neuron } \sigma_j \text{ neither consumes nor produces spikes} \\ & \text{in rule } r_i \end{cases}$$

Definition 4 (Status Vector). *This vector describes which neurons are open or closed at time k. The **Status Vector** is defined as:*

$$St^{(k)} = \langle st_1^{(k)}, st_2^{(k)}, \ldots, st_m^{(k)} \rangle$$

where:

$$st_j^{(k)} = \begin{cases} 1 & \textit{if } \sigma_j \textit{ is open;} \\ 0 & \textit{otherwise} \end{cases}$$

Definition 5 (Indicator Vector). *This vector describes which rules are applicable to produce spikes at time step k. Assume a total order $d : 1, \ldots, n$ is given for all the n rules of the system so they can be referred to as $r_1, r_2, \ldots, r_n$. The* ***Indicator Vector*** *is defined as:*

$$Iv^{(k)} = \langle iv_i^{(k)}, iv_2^{(k)}, \ldots, iv_n^{(k)} \rangle$$

where:

$$iv_i^{(k)} = \begin{cases} 1 & \textit{if the condition in rule } r_i \textit{ is satisfied to produce a} \\ & \textit{spike at time } k\textit{;} \\ 0 & \textit{otherwise} \end{cases}$$

Theorem 1 (Computation of System Configuration). *For $k \geq 0$, the system configuration is given by:*

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

We took the liberty of adapting the Matrix Representation to include input spike train and weights on synapses to achieve the feature parity with WebSnapse v2. The modified computation steps are detailed below. Furthermore, to consider neuron delays, new vectors were defined thereafter to compute for the Indicator Vector.

Definition 6 (Decision Vector). *This vector describes which rules are chosen at time step k. Assume a total order $d : 1, \ldots, n$ is given for all the rules of the system so they can be referred to as $r_1, r_2, \ldots, r_n$. The* ***Decision Vector*** *is defined as:*

$$Dv^{(k)} = \langle dv_1^{(k)}, dv_2^{(k)}, \ldots, dv_n^{(k)} \rangle$$

where:

$$dv_i^{(k)} = \begin{cases} 1 & \textit{if the condition in rule } r_i \textit{ was satisfied and chosen} \\ & \textit{at time step } k\textit{;} \\ 0 & \textit{otherwise} \end{cases}$$

Definition 7 (Delayed Indicator Vector). *This vector describes which rules were chosen yet delayed to produce spikes at time step k. The* ***Delayed Indicator Vector*** *is defined as:*

$$DIv^{(k)} = \langle div_1^{(k)}, div_2^{(k)}, \ldots, div_n^{(k)} \rangle$$

where:

$$div_i^{(k)} = \begin{cases} 1 & \textit{if } r_i \textit{ is scheduled to fire at some time } d > k \\ 0 & \textit{otherwise} \end{cases}$$

The Delayed Indicator Vector retains its value from the current time step to the next if it still has a delay and is already set to 1 (representing an ongoing delay). Additionally, if a rule is selected for firing and it has no delay, the corresponding element in the Delayed Indicator Vector is updated to 1. Logical Bitwise OR operation was performed to ensure that the Delayed Indicator Vector accurately represents the delayed firing status of each rule, taking into account both the existing delays and new delays introduced by rule selection.

Definition 8 (Delay Status Vector). *This vector describes how many ticks are left before a rule is set to fire. The* ***Delay Status Vector*** *is defined as:*

$$DSv^{(k)} = \langle dsv_1^{(k)}, dsv_2^{(k)}, \ldots, dsv_n^{(k)} \rangle$$

where:

$$dsv_i^{(k)} = \begin{cases} d & \textit{if } r_i \textit{ is scheduled to after } d \textit{ ticks} \\ 0 & \textit{otherwise} \end{cases}$$

The delay status vector was utilized to include rules set to fire at a time step d. When an element dst_i reaches 0, it indicates that the rule r_i is ready to fire a rule. Furthermore, the decision vector denotes which rules were chosen at the current tick. If the selected rule has no delay, it will fire. Finally, if there is a delayed rule that becomes ready to fire, it will be executed. The delay status vector works by setting all the rules of a neuron set to fire a rule with the same value. This makes it easier for checking for the applicability of rules on a closed neuron. To explain algorithm 4, it first loops over each delayed rule and decrements the rules mapped under its parent neuron. Then, it loops to all the chosen rules at the current time step and sets the rules mapped under its parent to that rule's delay initial value.

Definition 9 (Spike Train Vector). *This vector contains the number of spikes from the spike train to be sent to each neuron at time step k. The* ***Spike Train Vector*** *is defined as:*

$$STv^{(k)} = \langle spt_1^{(k)}, spt_2^{(k)}, \ldots, spt_m^{(k)} \rangle$$

where:

$$spt_i^{(k)} = \begin{cases} sw & \textit{if neuron } n_i \textit{ has an incoming synapse weighted } w \\ & \textit{from an input spike train set to fire } s \textit{ spikes at} \\ & \textit{time step } k; \\ 0 & \textit{otherwise} \end{cases}$$

One change for acquiring the next configuration is the addition of the Spike Train vector. For the algorithm 3, it was set as the initial value of the Net Gain before adding the product of the Indicator Vector and the Spiking Transition Matrix. This algorithm also assumes that consumption rules do not yet apply when a rule with delay is chosen, rather, the consumption of spikes will only be done when the neuron is set to fire them at their specified time step.

Definition 10 (Configuration Vector). *This vector was redefined to consider the delay, weights, and spikes from the input spike train. Furthermore, it was modified with the assumption that the consumption of spikes by closed neurons does not apply until they're fired in their firing step. The* ***Configuration Vector*** *is redefined as:*

$$C^{(k+1)} = C^{(k)} + St^{(k+1)} \odot \left(Iv^{(k)} \cdot M_{\Pi} + STv^{(k)} \right)$$

3 WebSnapse v2

Simulators provide a great advantage in testing artificial imitations of real-world systems. Through simulators, large systems that may be rare or impossible to occur in reality can be tested to reach relevant conclusions. Simulators simplify and speed up the testing of large systems that would rather take a long time if done manually. One abstraction of P System Simulators is the *Graphical User Interface (GUI)*, which is designed with graphics to visualize complex systems and interact with them. GUIs focus on providing simulation technology for users who are not familiar with P systems.

WebSnapse v2 is the extended version of the GUI simulator WebSnapse [7] developed by [6] in 2022. It enhanced the user experience of WebSnapse and added support for specific variants e.g. SN P Systems with Weights on Synapses and SN P Systems with Input Spike Train Neurons [2]. The addition of input spike train neurons and weights was able to reduce the number of neurons needed to construct certain SN P systems, paving the way for designing simpler SN P systems [6].

However, it still has multiple points for growth in terms of user experience. Additionally, like the previously discussed simulators, its computational method still follows an iterative algorithm as in the previous version [2], instead of the existing matrix representation for computing SN P systems provided at [3].

Note that existing simulators which aim to enhance simulation engine performance using CUDA [20, 19, 22, 5] and [11] have been in development by our colleageus from the ACLab and thus were left out of the study's scope. Rather, this research is centered on addressing the previously mentioned deficiencies identified in WebSnapse v2.

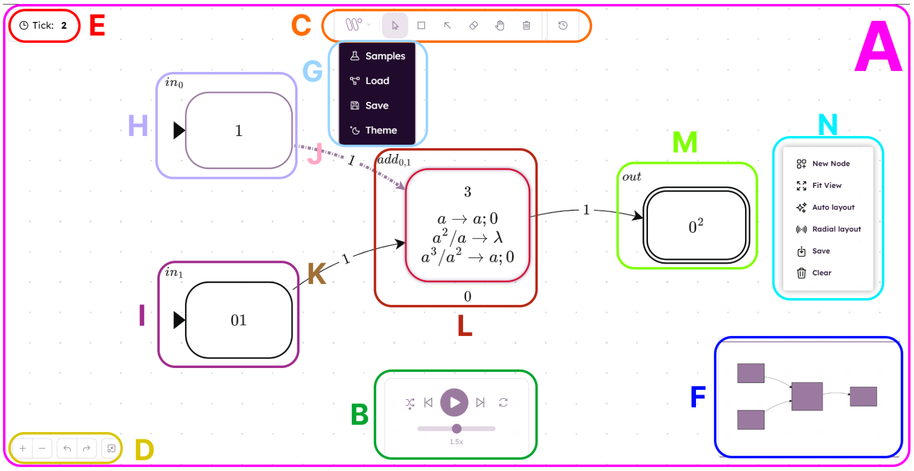

Fig. 2. WebSnapse Reloaded User Interface

4 WebSnapse Reloaded

The following section shows the features of WebSnapse Reloaded. This covers its basic functionalities, how the simulator computes each configuration of the system, and how the software was developed. The links to the source code, details for installation, experiments, and the application are available at the WebSnapse page [1].

4.1 Basic Functionalities

Figure 2 shows the user interface of WebSnapse, consisting of the following components labelled as such:

A **Workspace:** This renders the graphical representation of the system. This is also where the user can design their SN P system, connect neurons with directed synapses, zoom in and out, pan around the canvas, and view the simulation as it runs the system. This is also where the user can right-click to access the Context Menu (Label N).

B **Simulation Options:** This allows the user to run the simulation automatically or manually and select whether the system will run in Pseudorandom or Guided Mode which are defined later in the chapter. This is also where the user can adjust the speed of the simulation. The restart button allows the user to reset the simulation to its initial state.

C **Workspace Menu:** This contains the menu that can be accessed via the WebSnapse Reloaded logo. This is also where the different workspace states can be selected: View, Add Node, Add Synapse, Delete Element, Hand Mode, and Clear all. The Choice History Table, which displays the history of the simulation, can also be accessed through this menu. Each state is discussed further in Section 4.2.

D **Workspace Options:** This is where the user can zoom in or zoom out the canvas, undo or redo tasks, and change the node view.

E **Time Step Tracker:** This tracks the time step of the simulation.

F **Mini Map:** This serves as a thumbnail view of the whole SN P system present in the workspace which the user can navigate through mouse drag.

Other labels present in the figure are defined and discussed in the later subsections.

Neurons WebSnapse Reloaded divides neurons or nodes into three types. (1) Regular Nodes. As shown in Figure 2 Label L, these nodes contain an ID, the number of spikes, rules, and delay. Regular nodes can both send and receive outgoing synapses and spikes to and from other nodes. During simulation, the number below the node represents the current delay before it is able to send spikes again. (2) Input Spike Train Nodes. Figure 2 Label I shows this type of node. It can contain spikes with values $[0, 1, 2, ...n]$ of length k for any $k \geq 0$ which is specified by the user before the simulation runs. This node cannot receive, but it can send outgoing synapses and spikes to other nodes. The system reads the input spike train from left to right. For simplicity, the node displays multiple consecutive spike values as c_i^k where c_i is the spike value and k is the quantity. For example, 1111011 is displayed as $1^4 01^2$. (3) Output Spike Train Nodes. Figure 2 Label M shows this type of node. Like the previous type, it can contain spikes of the same format. However, this node acts like the environment, which means its values cannot be specified by the user. Instead, this node receives spikes from other connected nodes. This node cannot send, but it can receive outgoing synapses and spikes from other nodes. The output spike train result is read from right to left.

All types of nodes can be created, edited, and deleted in WebSnapse Reloaded. Upon adding a neuron, the system generates a random ID for the neuron which the user can modify before confirming the creation of the node. Type (1) node rules are displayed and written using LaTeX. Rules are written in the same form as in Definition 1.

During each step of a simulation, nodes can either be active or inactive. Active nodes are signified by a glowing purple outline which means they are currently sending out a spike in that time step. Inactive Type (1) and Type(2) nodes mean that there is no activity going on, and only apply the default appearance of the node. Meanwhile, active and inactive Type (1) nodes are discussed below.

Type (1) nodes can either be open or closed. Open nodes are able to send and receive spikes to and from other connected nodes. These nodes are signified by the delay value of 0 underneath it. Open nodes can be either active or inactive with the former executing a firing or forgetting rule and the latter not executing rules for that particular time step. However, it is still able to receive spikes from other connected nodes. Meanwhile, closed nodes are not able to both send and receive spikes. When other connected nodes send spikes to a closed node, these spikes are dropped and lost. Closed nodes are indicated by a red outline during simulation as shown in Figure 2 Label L. These nodes remain closed for the duration specified by the delay number under it.

Synapses Like that of WebSnapse v2, synapses in WebSnapse Reloaded are weighted and directed. By default, newly created synapses are set with weight value of 1. However, the user is free to edit this after creating the synapse by double-clicking the value. Like Open Nodes, synapses can either be one of two states during each simulation step. The Inactive state is the default state of the synapse signified by the default black color as shown in Figure 2 Label J. The Active state indicates that a synapse is currently sending a spike towards the node it is directed to. This is signified by a glowing purple animated broken line as shown in Figure 2 Label K.

Other Functionalities JSON Support
In WebSnapse Reloaded, systems can be imported and exported using JSON files. The original configuration of the system is converted into JSON syntax, making it easier to read and understand. Users can modify the elements of the system by editing the JSON file using any text editor. They can add more neurons, edit node connections, and make other changes as needed. Once the JSON file is edited, it can be reloaded into WebSnapse as long as there are no formatting errors.

There are multiple ways to load and save configuration files. One way is to access these options via the Context Menu. Like its predecessors, WebSnapse Reloaded is able to simulate in Pseudorandom or Guided mode. Through the Pseudorandom mode, the system simulates non-determinism by randomly selecting which applicable rule to run at each time step. Meanwhile, the user can specify which rules to select during points of non-determinism by running the simulation in Guided mode. The speed of the simulation can also be set by adjusting the slider shown in Figure 2 Label B. Like its predecessors, WebSnapse Reloaded provides the option for users to view spike train values and the rules applied by each of the Regular Nodes per time step during the simulation. Entries in the Choice History Table are formatted in LaTeX. This option is found by clicking the rightmost button of the Workspace Menu (Labeled C from Figure 2). Finally, the simulator offers the option to clear all the elements at once using the Clear All functionality which is done by clicking the Trash Bin Button on the same menu.

Feature Parity with WebSnapse v2 Among the goals of WebSnapse Reloaded was to reach feature parity with WebSnapse v2. Given the discussed basic functionalities, WebSnapse Reloaded was able to reach feature parity through the following features: (1) Neuron actions such as adding regular, input spike train and output spike train neurons, editing and deleting neurons; (2) Synapse actions like adding, editing, and deleting weighted synapses; (3) Loading and saving configuration files; (4) Running the simulation in Guided Mode or Pseudorandom Mode; (5) and other features like Choice History Table and Clear all button.

4.2 Enhanced Features

Simplified Functionalities for Nodes and Synapses Websnapse v2 [6] implements a decentralized method of accessing element functionalities, which costs additional steps and time. WebSnapse Reloaded was able to simplify this by creating states in the workspace accessed through the Workspace Menu in Figure 2 or via keyboard shortcuts specified below. Each state has its dedicated features as follows. (1) *Select State (V).* This is the default state upon opening WebSnapse Reloaded. This state is dedicated to understanding and experimenting with the current SN P system on the workspace. (2) *Node State (N).* This state is primarily dedicated to adding new nodes. When the system is in this state, the user can click on any empty area in the workspace to add a new node. This state is ideal for when the user wants to create multiple nodes before arranging the system with synapses. (3) *Edge State (E).* This state is primarily dedicated to added new synapses or edges. (4) *Delete State (D).* This state is primarily dedicated to deleting elements one by one. When the system is in this state, the user can simply click on any element they want to delete. (5) *Hand State (H).* This state is primarily dedicated to navigating and exploring the SN P system. The user can drag anywhere in the workspace to view different parts of their created system. While in this state, the SN P system cannot be modified. WebSnapse Reloaded also accommodates other keyboard shortcuts which can be seen in Table 1.

Function	**Shortcut**	**Function**	**Shortcut**
Clear all	`Q`	Auto Layout	`CTRL + L`
Change View	`Y`	Radial Layout	`CTRL + SHIFT + L`
Save/Export to JSON	`CTRL + S`	Undo	`CTRL + Z`
Load/Import JSON	`CTRL + O`	Redo	`CTRL + SHIFT + Z`
Duplicate	`CTRL + D`		

Table 1. Keyboard Shortcuts in WebSnapse Reloaded

Adding New Nodes. In WebSnapse v2, creating a new node requires 3 steps: (1) clicking the "New Node" button, (2) filling out the required fields, and (3) dragging the new node to a specific spot. This means that creating 100 new nodes requires 300 steps. In WebSnapse Reloaded, the process is simplified by through the node creation mode which allows users to double-click on the desired location for the new node before filling in the required forms, reducing the number of steps to 2 per node after entering the Node state. This means that creating 100 new nodes in WebSnapse Reloaded only requires 201 steps, reducing the total number of steps by about 1/3.

Adding Synapses. In WebSnapse v2, adding synapses requires the user to click and drag from a source node to a destination node. This leads to a pop up that lets the user specify the weight of the synapse. This is ideal for situations when each synapse made is assumed to be distinct. However, for systems that all synapses are the same value (e.g. 1), this accumulates when the user wants to add multiple synapses. WebSnapse Reloaded adapted the same method for adding a new synapse by click and drag but setting the default weight to 1. If the user wants a different synapse weight, they can simply edit the synapse through double-clicking.

Editing and Deleting Elements. In WebSnapse v2, editing and deleting nodes and synapses is done through dedicated buttons that lead to a pop-up screen with a dropdown list of element IDs. This can be time-consuming and confusing when dealing with multiple elements. In WebSnapse Reloaded, the process is simplified by allowing users to edit an element by double-clicking on it, and to delete elements by entering the Delete State and clicking on the target elements. This eliminates the need to search through a list of IDs and makes the process more efficient.

Simplified Loading and Saving Configuration Files WebSnapse v2 allows loading and saving configuration files through the Menu Actions. WebSnapse Reloaded simplified this process by allowing keyboard shortcuts (see Table 1). These functionalities may also be accessed via other methods such as the Logo Menu (see Figure 2 Label G).

Simplified General Appearance of the User Interface Among the aims of WebSnapse Reloaded was to make the simulator more intuitive for the end-users, especially the ones who are unfamiliar with SN P systems. To do this, the interface was designed with more familiar visual elements and more area for the workspace. The Workspace Menu and Simulation Options were designed to take up less space so that the area allotted for the workspace is maximized. Colors were set to softer tones and variety was reduced towards a monochromatic theme so that distracting elements are minimized, potentially increasing user productivity.

4.3 New Features

Researchers of WebSnapse v2 recommended a list of additional functionalities which can further improve the experience of using the simulators. These recommendations alongside other new features were added by WebSnapse Reloaded as follows: (1) $\LaTeX$ Support in presenting neuron rules. This allows for a cleaner and more familiar format in the interface; (2) Context Menu (Figure 2 Label N), accessible by right-clicking the workspace, allows users to easily execute common tasks; (3) Selecting multiple elements at once; (4) Duplicating elements; (5) Undo and Redo actions; (6) Auto Layout and Radial Layout for complex and cluttered systems; (7) Mini map navigation; (8) Focus node; (9) Fit View feature for large systems; (10) Simplify View which hides the rules of all the nodes, showing only the number of spikes contained in each node. (11) Change theme, where the user can use the simulator in light or dark mode; (12) Size-adaptation for nodes with more or less number of rules which maximizes the workspace; (13) Spike Train String Folding which aids in input and output analysis and optimizes space; and (14) Save Graph to Image (PNG Format);

4.4 Technologies and Development

Visual Simulation A more robust visual simulation system was created for WebSnapse Reloaded using the VueJS framework for the front-end, as it outperforms ReactJS in booting, redrawing, and bundling size benchmarks. The rendering issue for larger systems in both versions of WebSnapse was addressed by replacing CytoscapeJS with G6, an alternative graphing library that offers many features out of the box and is easier to implement. Additionally, $\LaTeX$ support was added to improve the representation of neurons and make it easier for users to visualize how the system works. This was achieved by converting the MathJax SVG DOM element to its data URI scheme values and adding it to the neuron object in the canvas.

Simulation Engine API The next configuration of the system is computed using the matrix representation of SN P Systems with Delay provided in [3] which reduces its algorithm as a series of matrix operations. This was implemented using the Python Framework Fast API alongside NumPy, a library optimized for matrix operations.

4.5 Architecture

Modularity The codebase of WebSnapse was structured with modularity in mind in order to improve the developer experience and make it easier to extend the application in the future. This was done by separating the software into the client and server sides of the application. Visualized in Figure 3 is the client-server architecture with each of their respective responsibilities.

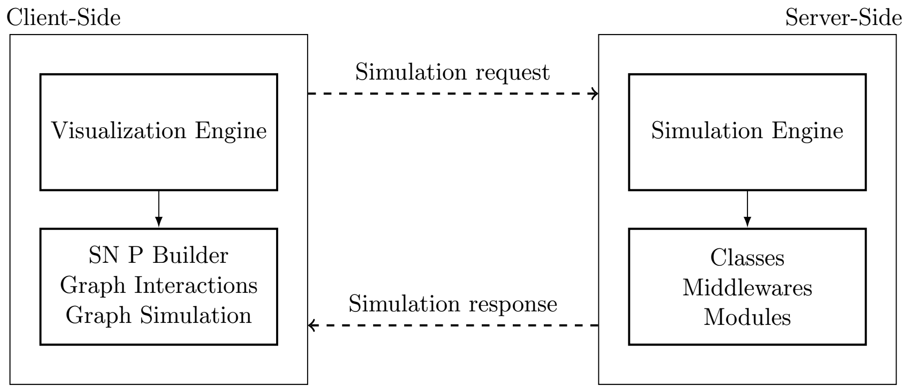

Fig. 3. Visualization of Client-Server Architecture

Communication Channel Websockets serve as the communication medium between the client and server, facilitating real-time and bidirectional data exchange. This technology enables efficient and instantaneous communication between the client-side and server-side components of the application. The utilization of websockets enhances the efficiency and speed of communication, making it ideal for applications that require instant updates, such as real-time collaboration tools, chat applications, and live data streaming.

Benefits and Versatility of the Architecture The utilization of the Simulation Engine API enables the server to store the context of the simulation, eliminating the need to save system change values in client storage and resolving the storage issue. The adoption of a Client-Server Architecture brings about additional benefits, such as enhanced scalability of the server, allowing for the seamless execution of simulations without compromising on quality or speed. The versatility of the Simulation Engine API extends to researchers who wish to develop their own clients, empowering them to tailor their own user interfaces and incorporate the API's functionality into their custom applications. The architecture introduces possible integration of new servers specifically designed for variants of SN P systems into the WebSnapse client application, allowing for the expansion of the platform's capabilities and the inclusion of diverse computational models. By integrating these new servers, the WebSnapse client application becomes a comprehensive and adaptable tool, offering a wide range of computational possibilities to researchers in the field.

4.6 Testing

Testing the reliability of the application was done using SN P systems from the literature that also included the benchmarks provided in the previous papers for WebSnapse. Evaluation metrics for the simulation of SN P Systems include its stability, which was determined through stress testing to find the maximum number of simulation steps the system can handle. Another metric is correctness, which was assessed by testing the simulation output against sample systems.

General Testing To test the correctness of the simulator, existing SN P Systems from literature were used. These test cases were compiled and formatted into JSON files and are available in the links provided at the WebSnapse page [1]. These tests are discussed further in Section 5. To test the functionality of the simulator, manual quality assurance and testing methods were conducted with the support of other fellow researchers in the Algorithms and Complexity Lab. Testing results showed that most user actions were working as intended, particularly the ones specified under Section 4. These major features included adding nodes, submitting forms, changing states, and running the simulation in Pseudorandom and Guided modes.

Some minor features occasionally encountered bugs such as adjusting the simulation speed, which sometimes causes the simulation to run much faster the specified speed. In a few cases, some features do not work which requires the user to refresh the browser to acquire the desired behavior. These are (1) selecting synapses, (2) playing/stopping simulation, (3) saving the state of the graph when refreshing the browser, and (4) recording choice history. The researchers infer that these bugs are likely caused by issues in memory handling of the graphing library used. This similar possible cause can also be traced from the results of the stress testing below.

Stress Testing Stress testing was done to see how many visual elements (i.e. neuron, rules, and synapses) can be rendered on the system, along with its performance according to the number of rendered elements. This testing was done only for the purpose of comparing the simulator's performance from its predecessor, WebSnapse v2. Four SN P systems were used as benchmarks. These were the original benchmarks used by WebSnapse [7] and WebSnapse v2 [6], the predecessors of WebSnapse Reloaded. These benchmarks are One Spike Chain Π_{osc}, All Spike Chain Π_{asc}, Benchmark Complete Π_{bc}, and Simple Complete Π_{sc} which are discussed below. Although the same benchmarking systems from its predecessors were used, WebSnapse Reloaded used a different method of testing the performance. Both WebSnapse and WebSnapse v2 measured the number of time steps reached by the benchmarks before crashing. However, the benchmarks given are expected to be non-halting systems. Upon testing, the simulator displayed this same behavior. Hence, instead of measuring the performance by the time steps, the researchers measured the time it takes for the simulator to perform certain tasks. This was compared with the node count, or the number of neurons present in the system being tested.

To gauge the performance of the simulator, three main tasks measuring system performance and load management were done on each benchmark. Console

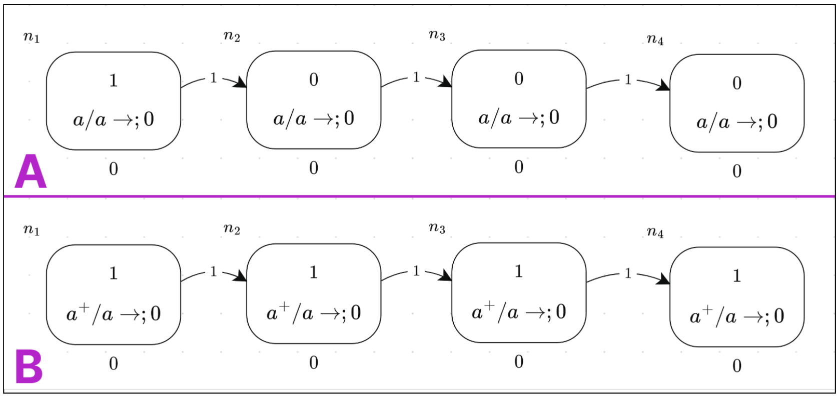

Fig. 4. Example of (A) One-Spike Chain and (B) All-Spike Chain with 4 nodes in WebSnapse Reloaded

timers were added to measure the time it took to complete these tasks. The first task measures the Initial Load Time which is the time it takes to import a system. The second task measures the Next Configuration Compute Time which is the time it takes to compute for the next configuration. Finally, the third task computes for the Average Re-render Time which is the time it takes to update elements of the graph. The four systems used for stress testing are detailed as follows:

One Spike Chain Π_{osc}
An example of this system can be found in Figure 4 Label A. This SN P system shows neurons connected in a unidirectional manner, with each regular neuron containing the same single rule: $a/a \to; 0$. For this system, only the first neuron contains a spike. Its behavior is similar to the game "Pass The Message", where the message is passed one at a time. Completion of the test happens when the spike reaches the end of the system. This happens at the time step $n - 1$, with n being the total number of neurons.

All Spike Chain Π_{asc}
It is similar to the previous Π_{osc} except that all neurons start with a single spike as shown in Figure 4 Label B. The recurring rule is also different, which is $a^+/a \to; 0$, in order to facilitate continuous spiking. As mentioned in [7], the purpose of testing this type of system was to see if the number of time steps that can be completed was only dependent on total neuron count, or if the number of neurons spiking at a time would serve as a factor.

Benchmark Complete Π_{bc}
This is a fully connected graph from Figure 4 of [4]. An example is shown in Figure 5 Label A. It contains two spiking rules: $(a^2)^*/a \to a$ and $(a^2)^*/a^2 \to a^2$, as shown in Figure 5 Label B, which cause non-determinism whenever the

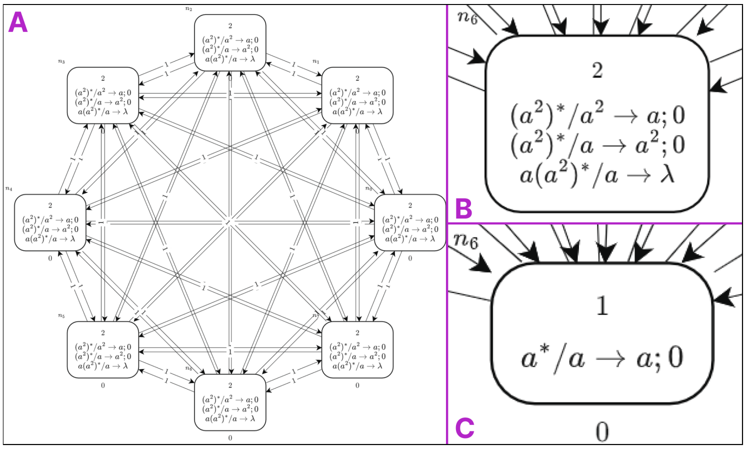

Fig. 5. Example of (A) Benchmark Complete System with 8 nodes, (B) each node in Benchmark Complete, and (C) each node in Simple Complete in WebSnapse Reloaded

number of spikes in a neuron is a multiple of 2. Originally, this was used as a tool for stress testing the parallel implementation of an SN P simulator in a GPU. Theoretically, this system does not halt, hence the performance of the simulator using this system is assessed by the number of time steps reached before the simulator crashes. Due to its non-determinism, the researchers simulated this system automatically through Pseudorandom mode.

Simple Complete Π_{sc}
This system from [7] is a modified version of the Benchmark Complete graph above and does not contain non-determinism. An example of a node from this system is found in Figure 5 Label C. This was created to assess if non-determinism also plays a role in the performance of the system.

One Spike Chain vs All Spike Chain Upon testing the first two benchmarks Π_{osc} and Π_{asc}, Figure 6 shows how the number of neurons affects the amount of time it takes to load a system. This is due to the initial rendering of image objects of the LaTeX representation of rules and node labels. Furthermore, the time it took for All Spike Chain systems to load are slightly higher than that of One Spike Chain systems which suggests that the number of rules in a system also affects its initial load time. Essentially, larger systems take longer to initially load in the simulator. For the computation time of the next configuration, Figure 7 shows an almost linear trend for both benchmarks. This suggests that as more neurons are added, the computation time for All Spike Chain systems moves farther away from that of One Spike Chain and that the number of spiking neurons affect the computation time for the next configuration. Finally, for the re-render time, both systems show increasing trends. This means that simultaneous spiking does not result in any significant difference in the re-render time between the two different systems.

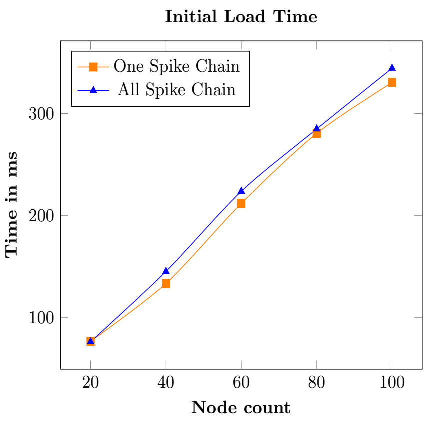

Fig. 6. Load time for importing Spiking Chain systems

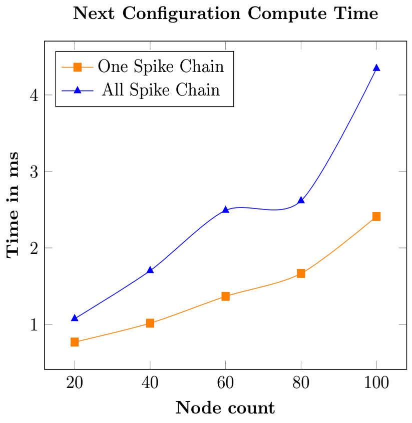

Fig. 7. Configuration Computation time for Chain Systems

Benchmark Complete vs Simple Complete

Initial loading for Benchmark Complete Π_{bc} systems showed higher completion time compared to Simple Complete Π_{sc} systems as shown in Figure 8. This follows the same conclusion from comparing the spike chain systems above, wherein it states that larger systems take longer to initially load in the simulator. Both complete graphs were able to load systems with 4 nodes in less than 100 ms. Meanwhile, loading Π_{sc} with 64 nodes lasted for 1 second while loading Π_{bc} with the same number of nodes lasted 1.3 seconds. It is also worth noting that a 1000-node chain system has a total of 999 edges while the 64-node complete system has 4032. This suggest that the number of edges in the graph also possibly influences the time it takes to initially load it to the system. For the Next Configuration Compute Time, Π_{sc} remains constant with 1 ms to 2.4 ms while Π_{bc} displayed a sharp increase from 4 ms with 8 nodes to more than 800 ms with 16 nodes, being completely unable to run the simulation at 32 nodes. The same behavior by Π_{bc} can be seen in the plots for Average Re-render Time in Figure 9.

This suggests two things: (1) the use of non-deterministic rules affects the computation time and (2) the number of synapses affects the computation time. This also suggests a potential for optimization of the current implementation of the matrix representation, especially for systems with non-determinism.

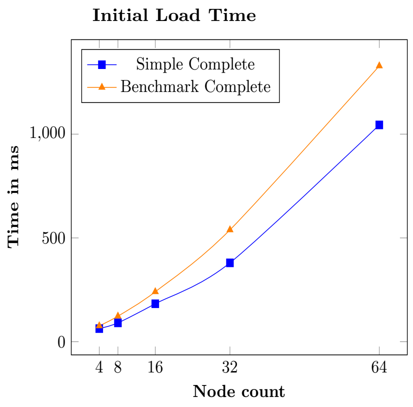

Fig. 8. Load time for importing Complete Systems

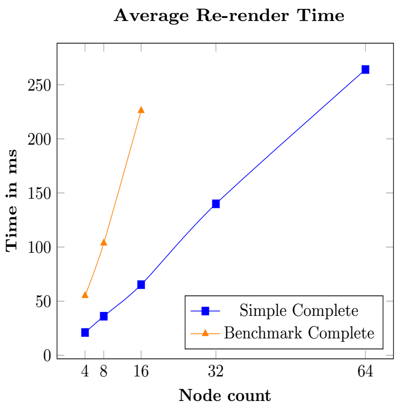

Fig. 9. Re-render time per time step of Complete Systems

4.7 Comparison with Other Tools

The first two benchmarking systems in Section 4.6 were used in WebSnapse Reloaded to compare its performance with WebSnapse v2. Unlike in the previous section, this section measured the performance of WebSnapse Reloaded in time steps as it was done in WebSnapse v2. It should be noted that the plots for WebSnapse Reloaded in the following figures do not represent the number of time steps it took for the simulator to crash. Due to the non-halting behavior of the benchmarking systems, we observed that WebSnapse Reloaded also displayed some non-halting behaviors during testing. Because of this, the researchers decided to manually stop the simulation once a system reaches the number of time steps equivalent to the number of neurons. It can be noted however that the simulator is able to go beyond these numbers and the researchers were not yet able to determine the maximum time steps for both One Spike Chain and All Spike Chain systems. The highest value acquired so far reaches at least 100,000 time steps for the Simple Complete system with 10 neurons.

Extensive testing of WebSnapse Reloaded has demonstrated its ability to surpass the limitations of previous versions of WebSnapse. As shown in Figure 10 and Figure 11, WebSnapse v2 peaks at 149 time steps at 150 neurons [6]. Meanwhile, WebSnapse Reloaded was able to sustain an increasing trend up until 1000 neurons.

WebSnapse Reloaded surpassed its predecessors in simulating the Spike Chain systems and was able to accommodate a much higher number of simulation steps. It was also able to load more neurons for the Simple Complete system, simulating 64 neurons as compared to 32 neurons in the tests done by its predecessors. The researchers were successful in implementing the computational model, that is the matrix representation for SN P systems. It showed

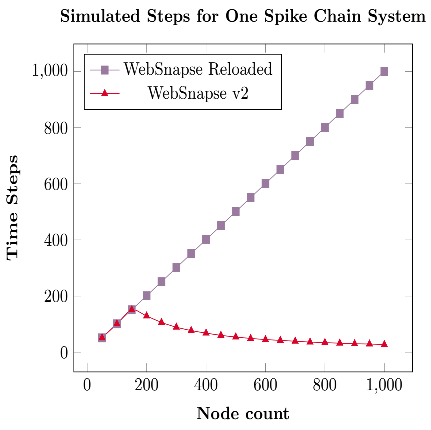

Fig. 10. Π_{osc} Simulated Steps on WebSnapse Reloaded vs WebSnapse v2

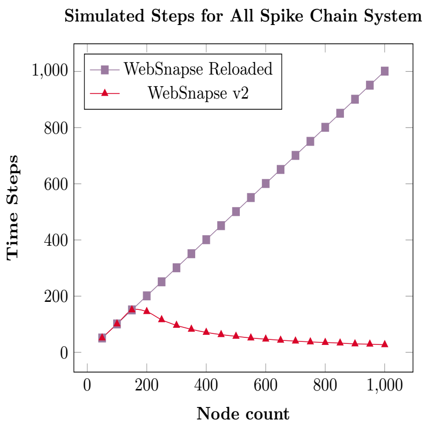

Fig. 11. Π_{asc} Simulated Steps on WebSnapse Reloaded vs WebSnapse v2

low compute times for the next configuration, ranging from 1 ms to 4ms for the Spike Chain systems of up to 100 neurons, and 1 ms to 2 ms for the Simple Complete systems of up to 64 neurons. However, results show an exception for the Benchmark Complete with 32 neurons, where WebSnapse Reloaded displayed its limitation to simulate the system. This can be traced to a possible bottleneck in computing for systems with non-determinism. Thus, the creation of an alternative method to obtain a pseudorandom generation of the decision vector is thereby suggested.

5 Case Studies

To further understand the capabilities of the software, several SN P systems will be covered to explain its core features. These tests were used as simple tests for correctness that check the simulator's capability to handle non-determinism and closed neurons with correct outputs. This section discusses three problems that can be solved by SN P systems and explains the results generated by WebSnapse Reloaded.

5.1 Even Positive Integer Generator

Predecessors of WebSnapse Reloaded used this SN P system to test the non-determinism capabilities of the simulator. This same system is used on WebSnapse Reloaded to test its capability to handle non-determinism. Its formal definition can be found on Figure 1 of [26]. The result of this system is obtained by computing the distance between the two 1s in the output spike train.

Running the simulator in Pseudorandom Mode allows the system to select which rule to execute at each corresponding timestep. A sample run displays a result after 9 ticks, which is the value $10^5 10^2$ in the output neuron. Ignoring the two trailing zeroes, this translates to a distance of 6 between the two 1s, giving us a result of the positive even integer 6. Running the simulator in Guided Mode transfers the control of selecting the rule to the user. When the system encounters non-determinism, a pop-up screen lets the user select which rule to execute for each node containing non-determinism. Selecting the rule without delay allows the system to run another cycle while selecting the rule with delay causes the system to halt. For the sample execution, the researchers first selected the rule without delay, followed by selecting the rule with delay. This sequence of selections resulted in a system that halted after 7 ticks with output spike train $10^3 10^2$. Ignoring the two trailing zeroes, this translates to a distance of 4 between the two 1s, giving us a result of the positive even integer 4. This shows the capability of WebSnapse Reloaded to handle non-determinism correctly.

Test Case	**No Delay**	**With Delay**	**Expected**	**Actual**
$subset\_sum(\emptyset, 7)$	-	-	Non-halting	Non-halting
$subset\_sum([1,2,3], 5)$	$[2,3]$	$[1]$	Halting	Halting
$subset\_sum([1,2,4,8], 15)$	$[1,2,4,8]$	$\emptyset$	Halting	Halting
$subset\_sum([1,3,5], 2)$	-	-	Non-Halting	Non-Halting
$subset\_sum([5], 5)$	$[5]$	$\emptyset$	Halting	Halting
$subset\_sum([9], 6)$	-	-	Non-Halting	Non-Halting

Table 2. Results of Subset Sum Test Cases on WebSnapse Reloaded

5.2 Subset Sum

The Subset Sum problem tests the ability of the simulator to handle rules with delay. As defined in Section 4.1, nodes that execute a rule with delay is considered closed for the duration specified by the delay. While in this state, the node should not be able to send a spike. The main reference for this problem is found in Figure 3 of [17] which shows a standard, non-uniform solution to Subset Sum. Slight modifications were applied to accommodate specific input values. Subset Sum takes two inputs: a list of integers L and a sum s. The system halts if and only if there exists a subset of L whose sum is s. If N is the number of subsets of L whose sum is s, then the probability of halting in Pseudorandom Mode is $\frac{N}{2^{|L|}}$. Running in Guided Mode helps with acquiring a deterministic result. In Guided Mode, the system is expected to halt when a subset S of L whose sum is s executes the rules without delay and the complement S' executes the rules with delay. Table 2 reflects the results of running the Subset Sum test cases in WebSnapse Reloaded Guided Mode.

The first test case was expected to not halt because there exists no subset $\emptyset$ that sums up to 7. The second test case, on the other hand, contains a subset that satisfies the requirements of the problem. Given that $2 + 3 = 5$, c_2 and c_3 corresponding to values 2 and 3 were set to execute the rule without delay. Meanwhile, c_1 corresponding to 1 was set to execute the rule with delay. These results from Table 2 show that WebSnapse Reloaded was able to run instances of the Subset Sum as expected, therefore displaying its capability to handle rule delays correctly.

5.3 Multiples of *k* Generator

Closed neurons in an SN P system are not only unable to send spikes for the duration of its delay, but should also be unable to receive incoming spikes from other connected neurons [26]. This was among the limitations of WebSnapse v2. Upon testing, it was observed that closed neurons continued to receive spikes from connected nodes even before the delay ended. Given this, the researchers aimed to resolve this in WebSnapse Reloaded.

To test the simulator's implementation of closed neurons, SN P systems that generate multiples of k were used by the researchers. This system was obtained from fellow researchers Tim Kristian Llanto and Joshua Amador who designed the system.

The researchers first ran the systems generating multiples of 3 in Guided Mode. At time step 8, neurons 1 and 2 are closed as shown in Figure 12. Since neuron 1 only has delay equals 1 remaining, it will fire one spike neighboring neurons at the next tick. At time step 9, neuron 1 opens and fires its spike to neurons 2 and 3. Neuron 3 received this spike. However, during this time, neuron 2 is still closed for one more time step and thus, no changes were made to its number of spikes. The same behavior was also observed for neuron 3 when it becomes closed at time step 10. After selecting the terminating rule, the system was able to halt at time step 14 with the result $10^{11}10$. Ignoring the trailing zeroes and computing the distance between two 1s gives us the result of integer 12, which is a multiple of 3. Proving that WebSnapse Reloaded was able to handle closed nodes correctly.

6 Final Remarks

WebSnapse Reloaded optimized use of storage by implementing the client-server architecture. Through this, computations and displays are now in separate layers, making it easier for future developers to extend, modify, and maintain the simulator. Furthermore, it was successful in correctly implementing the matrix representation for SN P systems by passing all of the test cases for correctness and resolving a computation error that existed in WebSnapse v2 in handling closed neurons. However, a bottleneck was found in simulating systems with

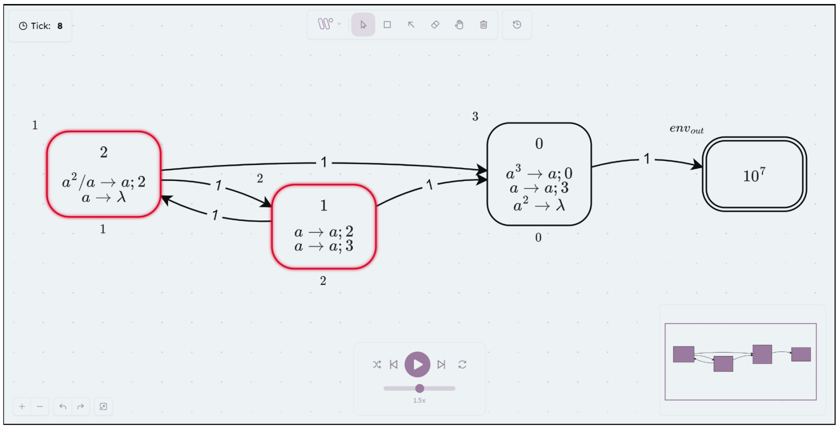

Fig. 12. Time step 8 of an SN P system that generates multiples of 3

non-determinism in Pseudorandom mode, and future research is recommended to revise the algorithm.

WebSnapse Reloaded improved the user experience by reaching feature parity with WebSnapse v2, with minor differences in input file format, user guides, and tutorial mode. The user interface was improved using Vue.js and G6, simplifying task execution, adding new features, and maximizing the workspace area for simulation. However, limitations were found in G6, and it is recommended to explore other graphing libraries or integrating WebGL to further optimize the front-end aspect of the simulator. General and stress testing were mostly done manually, and it is suggested to utilize automated testing technologies.

Recommendations from WebSnapse [7] may also be revisited as many of their insightful suggestions on user experience and architecture were not yet implemented by the researchers. Among these recommendations include (1) allowing users to create patterns for the rules before running the simulation in Guided Mode, (2) providing a computation tree to view the history of the number of spikes, and (3) creating a custom rule input parser to support more variants of SN P systems. Adding new features may also be explored for ease of use. Among these are adding an option to apply Pseudorandom and Guided Modes on a node-by-node basis; and allowing the execution of steps that focuses only on a specific set of nodes, skipping the time steps when these selected nodes are idle.

Additionally, leveraging the advantages of the client-server architecture, it is highly recommended to develop dedicated APIs for other variants of SN P Systems and integrate support for them into future iterations of WebSnapse Reloaded. These notable variants encompass Homogenized SN P Systems by [18] and NSNP Systems by [12], which our colleagues from the UPD-ACLab have been actively researching as well.

Finally, WebSnapse Reloaded has attained most of its initial goals in creating a visual simulator for SN P systems with client-server architecture on the web for both beginners and experts. With this, we look forward to future researchers and developers who wish to further optimize the system, utilize its potential, and extend it to other variants towards a centralized SN P system simulator on the web.

7 Acknowledgments

J. Luzada acknowledges the financial support received for this thesis through the DOST-SEI scholarship. 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. The authors also thank Louie Gallos and Rey Christian Delos Reyes for their significant insights and knowledge in constructing some of the mathematical aspects of our methods. Finally, the researchers value the constructive feedback from the two anonymous reviewers, which improved this paper.

References

1. Websnapse page. https://aclab.dcs.upd.edu.ph/productions/software/websnapse.
2. Henry N. Adorna. Computing with sn p systems with i/o mode. *Journal of Membrane Computing*, 2(4):230–245, Nov 2020.
3. Henry N. Adorna. Matrix representations of spiking neural p systems: Revisited. *arXiv preprint arXiv:2211.15156*, 11:146–166, 2022.
4. Francis George C Cabarle, Henry N Adorna, Miguel Ángel Martínez del Amor, and Mario de Jesús Pérez Jiménez. Improving gpu simulations of spiking neural p systems. *Romanian Journal of Information Science and Technology, 15 (1), 5-20.*, 2012.
5. Jym Paul Carandang, Francis George C Cabarle, Henry Natividad Adorna, Nestine Hope S Hernandez, and Miguel Ángel Martínez-del Amor. Handling nondeterminism in spiking neural p systems: Algorithms and simulations. *Fundamenta Informaticae*, 164(2-3):139–155, 2019.
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 Dupaya, Anica Galano, Francis Cabarle, Ren Tristan de la Cruz, Korsie Ballesteros, and Prometheus Lazo. A web-based visual simulator for spiking neural p systems. *Journal of Membrane Computing*, 4:1–20, 03 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. Aleksei Fernandez, Reyster Fresco, Francis Cabarle, Ren Tristan de la Cruz, Ivan Cedric Macababayao, Korsie Ballesteros, and Henry Adorna. Snapse: A visual tool for spiking neural p systems. *Processes*, 9:72, 12 2020.
10. Pierluigi Frisco, Marian Gheorghe, and Mario J Pérez-Jiménez. *Applications of membrane computing in systems and synthetic biology.* Springer, 2014.
11. Louie Gallos, Jose Lorenzo Sotto, Francis George C. Cabarle, and Henry N. Adorna. Websnapse v3: Optimization of the web-based simulator of spiking neural p system using matrix representation, webassembly and other tools. 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 492–510, 2023.
12. Reannu Emmanuel Instrella and John David Vidad. Nsnapse: A visual web-based simulator of numerical spiking neural p systems using matrix representation algorithms. CS 199 Report, Department of Computer Science, University of the Philippines Diliman, 2023. Accessed 2023-07-04.
13. Mihai Ionescu, Gheorghe Păun, and Takashi Yokomori. Spiking neural p systems. *Fundamenta informaticae*, 71(2-3):279–308, 2006.
14. Mihai Ionescu and Dragos Sburlan. Some applications of spiking neural P systems. *Comput. Informatics*, 27(3+):515–528, 2008.
15. Alberto Leporati, Giancarlo Mauri, and Claudio Zandron. Spiking neural p systems: main ideas and results. *Natural Computing*, pages 1–21, 2022.
16. 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, 2009.
17. 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, 2009.
18. Tim Kristian Llanto and Joshua Amador. Analyses and implementation of a homogenisation algorithm for spiking neural p systems. CS 199 Report, Department of Computer Science, University of the Philippines Diliman, 2023. Accessed 2023-07-04.
19. 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.
20. Ayla Nikki Lorren Odasco, Mathew Lemuel Rey, and Francis George C. Cabarle. Improving gpu web simulations of spiking neural p systems. In *Pre-proceedings of the 11th Asian Conference on Membrane Computing, Quezon City, Philippines*, pages 167–194, 09 2022.
21. Linqiang Pan, Xiangxiang Zeng, Xingyi Zhang, and Yun Jiang. Spiking neural p systems with weighted synapses. *Neural Processing Letters*, 35:13–27, 2012.
22. Jym Paul, A Carandang, John Matthew, B Villaflores, Francis George, C Cabarle, Henry Adorna, and Miguel Martinez-Del-Amor. Cusnp: Sn p systems simulator in cuda cusnp: Spiking neural p systems simulators in cuda. *Romanian Journal of Information Science and Technology*, 20(1):57–70, 2017.
23. Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108–143, 2000.
24. Gheorghe Păun. From cells to (silicon) computers, and back. *New Computational Paradigms: Changing Conceptions of What is Computable*, pages 343–371, 2008.
25. Gheorghe Păun, Gzegorz Rozenberg, and Arto Salomaa, editors. *The Oxford Handbook of Membrane Computing.* Oxford Univeristy Press, 2010.

26. Gheorghe Păun. Spiking neural p systems. a tutorial. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 01 2007.
27. Gheorghe Păun and Grzegorz Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, Sep 2002.
28. Luis Valencia-Cabrera, Ignacio Pérez-Hurtado, and Miguel Á. Martínez-del Amor. Simulation challenges in membrane computing. *Journal of Membrane Computing*, 2(4):392–402, Oct 2020.
29. Xiangxiang Zeng, Henry Adorna, Miguel Ángel Martínez-del Amor, Linqiang Pan, and Mario J. Pérez-Jiménez. Matrix representation of spiking neural p systems. *Membrane Computing*, page 377–391, 2010.

8 Appendix

Algorithm 1 Compute Indicator Vector at step k

Input: Decision Vector $Dv^{(k)}$, Delayed Indicator Vector $DIv^{(k)}$, Delay Status Vector $DSv^{(k)}$
Output: Indicator Vector $Iv^{(k)}$

```
1: procedure getIndicatorVector
2:     for $i \leftarrow 0$ to $m$ do
3:         $Iv_i \leftarrow (Dv_i \vee DIv_i) \wedge \neg DSv_i$
4:     end for
5:     return $Iv$
6: end procedure
```

Algorithm 2 Compute Delayed Indicator Vector at step k

Input: Delayed Indicator Vector $Dv^{(k-1)}$, Indicator Vector $Iv^{(k)}$, Decision Vector $D^{(k)}$, Delay Status Vector $DSv^{(k)}$
Output: Delayed Indicator Vector $DIv^{(k)}$

```
1: procedure getDelayedIndicatorVector
2:     for $i \leftarrow 0$ to $m$ do
3:         $DIv_i \leftarrow (DIv_i \wedge \neg Iv_i) \vee (Dv_i \wedge \neg DSv_i)$
4:     end for
5:     return $DIv$
6: end procedure
```

Algorithm 3 Get Next Configuration

Input: Configuration Vector $C^{(k)}$, Status Vector $St^{(k)}$, Indicator Vector $Iv^{(k)}$, Spiking Transition Matrix M
Output: Configuration Vector $Iv^{(k)}$

```
1: procedure getNextConfiguration
2:     $NG \leftarrow STv$
3:     for $i \leftarrow 0$ to $m$ do
4:         for $j \leftarrow 0$ to $n$ do
5:             $NG_i \leftarrow NG_i + (Iv_i \times M_{ij})$
6:         end for
7:     end for
8:     for $i \leftarrow 0$ to $n$ do
9:         $C_i \leftarrow C_i + (St_i \times NG_i)$
10:    end for
11:    return $C$
12: end procedure
```

Algorithm 4 Compute Delay Status Vector at step k

Input: Decision Vector $Dv^{(k)}$, Delayed Indicator Vector $DIv^{(k)}$, Rule Delay Vector RDv
Output: Delay Status Vector $DSv^{(k)}$

```
1: procedure getDelayStatusVector
2:     $m \leftarrow COUNT(rules \in \Pi)$
3:     $DSv \leftarrow [0]_{1\times m}$
4:     for each $(i, delayed) \in DIv$ do
5:         if $delayed$ then
6:             $n \leftarrow$ getMappedNeuron(i)
7:             $rules \leftarrow$ getMappedRules(n)
8:             for each $r \in rules$ do
9:                 $DSv_r \leftarrow DSv_r - 1$
10:            end for
11:        end if
12:    end for
13:    for each $(i, decision) \in Dv$ do
14:        if $decision$ then
15:            $delay \leftarrow RDv_i$
16:            $n \leftarrow$ getMappedNeuron($i$)
17:            $rules \leftarrow$ getMappedRules($n$)
18:            for each $r \in rules$ do
19:                $DSv_r \leftarrow delay$
20:            end for
21:        end if
22:    end for
23:    return $DSv$
24: end procedure
```

Open Access This chapter is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International License (http://creativecommons.org/licenses/by-nc/4.0/), which permits any noncommercial use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.