Article

Snapse: A Visual Tool for Spiking Neural P Systems

Aleksei Dominic C. Fernandez, Reyster M. Fresco, Francis George C. Cabarle *, Ren Tristan A. de la Cruz, Ivan Cedric H. Macababayao, Korsie J. Ballesteros and Henry N. Adorna

Algorithms and Complexity, Department of Computer Science, University of the Philippines Diliman, Quezon City 1101, Metro Manila, Philippines; acfernandez4@up.edu.ph (A.D.C.F.); rmfresco@up.edu.ph (R.M.F.); radelacruz@up.edu.ph (R.T.A.d.l.C.); ivan.cedric10@gmail.com (I.C.H.M.); korsie.ballesteros@gmail.com (K.J.B.); hnadorna@up.edu.ph (H.N.A.)
* Correspondence: fccabarle@up.edu.ph

Abstract: Spiking neural P (SN P) systems are models of computation inspired by spiking neurons and part of the third generation of neuron models. SN P systems are equivalent to Turing machines and are able to solve computationally hard problems using a space-time trade-off. Research in SN P systems theory is especially active, more so in recent years as more efforts are directed towards their real-world applications. Usually, SN P systems are represented visually as a directed graph and simulated through mainly text-based simulations or tables. Thus, there is a need for tools that can simulate and create SN P Systems in a visual and easy-to-use manner. Snapse is such a tool which aims to hasten the speed and ease at which researchers may create and experiment with SN P systems. Furthermore, visual tools such as Snapse can help further bring SN P systems outside of theoretical computer science.

Keywords: membrane computing; spiking neural P systems; visual simulator

1. Introduction

The study of natural phenomena and their (potential) use in computer science and in the field of natural computing is an active area of research. Some of these phenomena gave birth to the branch known as membrane computing, giving rise to computing models that mimic the architecture of cells, making use of compartments called membranes [1]. Membrane computing paved the way for the creation of new devices, known as P systems, that may be used to improve current algorithms or techniques. One kind of P system is the spiking neural P system (SN P system) introduced in [2]. SN P systems are inspired by the workings of the human brain and models the way neurons work: they are mainly characterized by the spiking of information, i.e., passing spikes from one neuron to another through the connections known as synapses.

One advantage of P systems is their distributed and parallel nature, which membrane computing shares with many other areas of natural computing, e.g., DNA, quantum [3]. P systems, as early as 2001, are known to be able to efficiently solve computationally hard problems, by trading space or memory for time [4]. Another advantage of many P systems is their discreteness, e.g., chemicals are represented as symbols, cells are illustrated as ovals, which may contain inners cells or ovals, chemical reactions are represented as rewriting rules $u \rightarrow v$. In this way, P systems are also transparent and scalable even for non-computer scientists, for use in modelling and analyses, e.g., biology (including systems and synthetic), ecology, economics, linguistics. A book for such applications as early as 2006 is in [5], with a recent one in [6]. A handbook for theory and applications of P systems is in [7].

SN P systems, like other P systems, are Turing-complete [2] (i.e., they are algorithms) and able to solve NP-complete, or computationally hard, problems [8]. There are also works that focus on simulating SN P systems, as they are parallel in nature, in GPUs such as [9,10] and more recently in [11–13]. Much theoretical work has been done on SN P systems, e.g., their normal forms [14–16], formal representations [17–19], and their relations to classical

models of computation [20–25] with a short and recent survey in [26]. After much theoretical work, more recently the work to apply SN P systems to real-world problems becomes even more active, with some early works on image processing e.g., [27] and more recently in [28], use for cryptography [29–31], use of evolutionary algorithms to design SN P systems [32–34], in pattern recognition [35,36], computational biology [37], with a recent survey in [38].

In this paper, Snapse, a graphical user interface and visual simulator, is introduced. A common problem that researchers face is the lack of a tool that eases the difficulty of constructing and simulating SN P systems. In this paper and for the development of the Snapse tool, we study the functionalities and interfaces of established tools in creating Snapse for SN P Systems. In a survey on P-Lingua, a generic simulation framework for P systems, in [39] it is said that "in order to provide [researchers] with more usable mechanisms to experiment with models based on P systems, they would require more visual elements to help them save time and clarify the evolution of the systems under study". It was noted in the same paper that membrane computing models proved to be useful in fields outside of computer science, e.g., biology or economics. Researchers from these fields are not necessarily familiar with P systems, which is why there is a need for higher-level tools for such researchers without delving deeper into the more technical details of P systems, i.e., graphical user interfaces can play an essential role.

Snapse aims to contribute to this role by focusing on a more user-oriented interface: the design is simple enough to be understood by both new users and more experienced researchers. For those experienced in SN P systems, they are presented an avenue to create, edit, and simulate their own SN P systems with ease, providing them with a visual way to study such systems. For those not familiar with P systems, they are given the ability to load and modify models that other experts have developed to help understand the phenomena they are focused on.

The rest of the paper is structured as follows. Section 2 introduces some definitions on SN P systems. Section 3 explains the functionalities of *Snapse* and gives an overview of the syntax, technologies, simulation algorithms, and architecture used. In Section 4, Snapse is compared with similar existing tools such as MeCoSim, P-Lingua, and the like. Several examples of SN P Systems and how they are implemented in the program are then presented in Section 5. Finally, conclusions and possible future work are laid out in Section 6.

2. Preliminaries

2.1. Spiking Neural P Systems

A spiking neural P system (or SN P System) is a system of neurons, where neurons pass information among each other through the firing of spikes. Păun gives a tutorial on SN P systems [40], describing them as follows.

Definition 1 (SN P system). *A 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 $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:*

　　(a)　$n_i \geq 0$ is the initial number of spikes contained in σ_i ;
　　(b)　R_i is a finite set of rules of the following two forms:

　　　　i.　$E/a^c \rightarrow a^p; d$, where E is a regular expression over $\{a\}$ and $c \geq p \geq 1, d \geq 0$;
　　　　ii.　$a^s \rightarrow \lambda$, for $s \geq 1$, with the restriction that for each rule $E/a^c \rightarrow a^p; d$ of type (i) from R_i, we have a $s \notin L(E)$;

3.　*$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn, 1 \leq i, j \leq m$ (synapses between neurons);*
4.　*$in, out \in \{1, 2, \ldots, m\}$ indicate the input and the output neurons, respectively.*

Rules of type (*i*) are commonly known as firing or spiking rules and rules of type (*ii*) are known as forgetting rules. SN P systems whose firing rules are only $p = 1$ are said to

be standard SN P systems, firing only one spike when a firing rule is called. A firing rule $r_i \in R_i$ can be used when a neuron σ_i contains k spikes, and $a^k \in L(E)$, where $k \geq c$ and $L(E)$ is the language of expression E. The firing rule consumes (removes) c spikes from σ_i, leaving it with $k - c$ spikes. At the same time, it fires p spikes to the neurons with a synapse from neuron σ_i after a delay of d time steps. If $d = 0$, σ_i fires immediately. If $d \geq 1$, the neuron will be closed and it shall not receive spikes (all incoming spikes are lost) until it fires at the $t + d$ time step. A forgetting rule works under similar conditions as a firing rule: when a neuron σ_i contains s spikes, then $a^s \to \lambda$ shall be used, removing all spikes from σ_i.

As elaborated in [2], much inspiration is taken from spiking and biological neurons. The delay $d \geq 1$ is inspired by the refractory period such that a neuron which applied a rule with such a delay becomes closed, i.e., cannot fire or receive spikes. The constant c takes inspiration from the spiking threshold, while forgetting rules are inspired by leakage or decay of spikes in neurons over some duration of time. The constant p represents the spike magnitude emitted by neuron σ_i, where rules with $p = 1$ and $p > 1$ are known as standard and extended rules, respectively, as in [41].

It is important to note that the system assumes a global clock meaning that all applicable rules shall be at the same time and all spikes fired simultaneously. There may be cases where multiple rules in the same neuron are applicable. In that case, a rule is chosen in a non-deterministic manner. Note that by definition, a firing rule and a forgetting rule can not be applicable at the same time.

A configuration is a snapshot of the state of an SN P system for a given time step. Each configuration describes the number of spikes in each neuron and the time steps needed for each neuron to be open again. A configuration is of the form $< r_1/t_1, \ldots, r_m/t_m >$ where r_i denotes the number of spikes in the neuron and t_i denotes the number of time steps, with the initial configuration as $< n_1/0, \ldots, n_m/0 >$, i.e., all neurons are initially open.

SN P systems may be generative (i.e., the output neuron fires spikes to the environment), accepting (i.e., the input neuron receives spikes from the environment), or both, acting as a transducer [20]. Only the generative systems are considered in this paper, while the others are considered for future work. The result of a computation in a generative SN P system can be considered in various ways, halting or otherwise, e.g., the output is some number n which is the difference between specific and consecutive spikes, or by considering spike trains, where a time step with or without a spike is considered for example as a bit 0 or 1, respectively.

2.2. Spiking Neural P Systems with Extended Rules

Introduced in [41] is the notion of having a more generalized format for the rules by allowing both firing and forgetting rules to follow the format $E/a^c \to a^p; d$. Formally, [41] defines SN P systems with extended rules as:

Definition 2 (SN P systems with extended rules). *A spiking neural P system with extended rules of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_0)$, where:*

1. *$O = \{a\}$ is the singleton alphabet (a is called spike);*
2. *$\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:*

    (a) *$n_i \geq 0$ is the initial number of spikes contained in σ_i ;*
    (b) *R_i is a finite set of rules of the form $E/a^c \to a^p; d$, where E is a regular expression over $\{a\}$, $c \geq 1$, $p \geq 0$, and $d \geq 0$ with the restriction that $c \geq p$;*

3. *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn, 1 \leq i, j \leq m$ (synapses between neurons);*
4. *$i_0 \in \{1, 2, \ldots, m\}$ indicate the output neuron σ_{i_0}.*

A rule $E/a^c \to a^p; d$ is applied as follows. If the neuron σ_i contains k spikes, and $a^k \in L(E), k \geq c$, then the rule can fire, and its application means consuming (removing) c spikes (thus only $k - c$ remain in σ_i) and producing p spikes, which will exit the neuron

immediately. A global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized.

2.3. Matrix Representation of an SN P System

A representation of an SN P system through matrices is introduced by Zeng et al. [17]. It breaks down the different parts of an SN P system into vectors and matrices. Below are the vectors and matrices defined by their work.

Definition 3 (Configuration Vector)**.** *Let Π be an SN P system with m neurons, for any $k \in N$, the vector $C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)})$ is called the kth configuration vector of the system, where $n_i^{(k)}$ is the amount of spikes in neuron σ_i, $i = 1, 2, \ldots, m$ after the kth step of the computation. The initial configuration is denoted by the vector $C_0 = (n_1, n_2, \ldots, n_m)$*

Definition 4 (Spiking Vector)**.** *Let Π be an SN P system with m neurons and n rules referred to as $r_1, r_2, \ldots, r_n$. A spiking vector $s^{(k)}$ is defined as follows: $s^{(k)} = (r_1^{(k)}, r_2^{(k)}, \ldots, r_n^{(k)})$, where:*

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

The initial spiking vector is represented by $s^{(0)} = (r_1^{(0)}, r_2^{(0)}, \ldots, r_n^{(0)})$

Definition 5 (Spiking Transition Matrix)**.** *Let Π be an SN P system with m neurons and n rules, and $d : 1, \ldots, n$ be a total order given for all the n rules. The spiking transition matrix of the system Π, is defined as $M_\Pi = [a_{ij}]_{n \times m}$, where:*

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

Definition 6 (Net Gain Vector)**.** *Let Π be an SN P system with m neurons and n rules, and $C_k = (n_1^{(k)}, n_2^{(k)}, \ldots, n_m^{(k)})$ be the kth configuration vector of Π. The transition net gain vector at step k is defined as $NG^{(k)} = C_{k+1} - C_k$.*

2.4. Further Representations of SN P Systems

Further research on representing them in a non-visual way in CUDA is presented here [10]. Note that the representation in [10] is designed towards a parallel and GPU implementation. Snapse may be extended to include GPU computation which is discussed in Section 6.

Definition 7 (Status Vector)**.** *The kth status vector is denoted by $St^{(k)} = <st_1, \ldots, st_m>$ where for each $i \in \{1, 2, \ldots, m\}$,*

$$st_i = \begin{cases} 1 & \text{if neuron } i \text{ is open,} \\ 0 & \text{if neuron } i \text{ is closed.} \end{cases}$$

Definition 8 (Rule Representation)**.** *$R^{(k)} = <r_1, \ldots, r_n>$ where for each $i = 1, \ldots, n$, $r_i = (E, j, d', c)$ where E is the regular expression for rule i, j is the neuron that contains the rule r_i,*

$$d' = \begin{cases} -1 & \text{if the rule is not fired,} \\ 0 & \text{if the rule is fired,} \\ \geq 1 & \text{if the rule is currently on delay (i.e., neuron is closed).} \end{cases}$$

3. Snapse: A Graphical User Interface for Simulating and Creating Spiking Neural P Systems

3.1. Basic Functionalities and Limitations

The Snapse interface is shown in Figure 1 with labels for each section and presents the following elements:
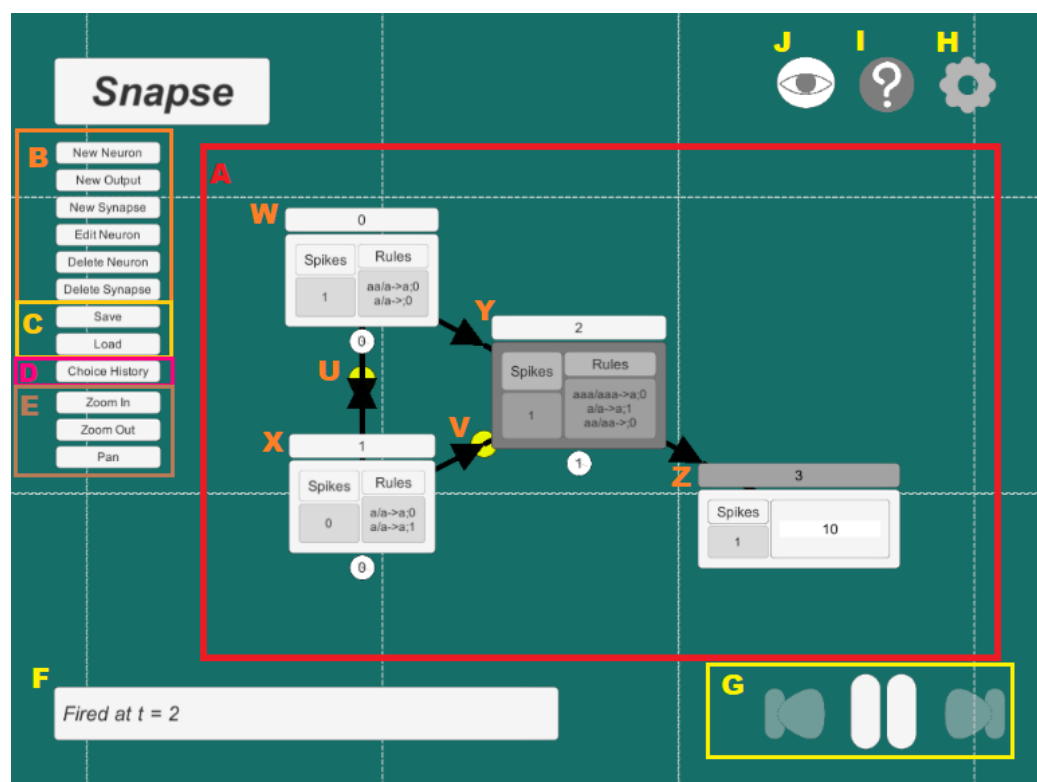

Figure 1. The basic Snapse interface.

- Workspace (labelled A): contains the entire SN P system.
- Neuron Editing Options (labelled B): contains all the buttons that are useful in modifying the SN P systems. This includes adding new neurons, output neurons, and synapses; removing neurons and synapses; modifying rules and spikes for each neuron.
- Saving and Loading (labeled C): contains buttons to save and load work.
- Choice History Window (labelled D): Opens the choice history window which lists all non-deterministic choices made.
- Viewing Options (labelled E): contains all the buttons that help in navigating through the SN P System. This includes panning, zooming in and out.
- Status Indicators (labelled F): a status bar that shows through text the current process undertaken by the program.
- Firing Options (labelled G): contains the buttons that control the firing of neurons. This includes continuous firing, moving forward one time step, and moving backward one time step.
- Settings (labelled H): Contains simulation modes and hide/view options for rules, labels, and animations.
- About/Help (labelled I): Contains useful information such as keyboard shortcuts and the output path.
- Hide interface (labelled J): Toggles the visibility of all unnecessary buttons surrounding the workspace to reduce clutter.

Snapse allows users to simulate and visualize SN P systems with delays and extended rules (see Definition 2) through the CPU. Section 6 makes recommendations on how to expand this scope.

To deal with non-determinism, Snapse employs two simulation modes: guided and pseudorandom. In guided mode, the user is given the freedom to choose between any number of applicable rules while in pseudorandom mode, Snapse automatically chooses the rule in a pseudorandom manner. Additionally, a history of all the rules chosen may be viewed in a window through the Choice History window.

3.2. Neuron and Output Representation

SN P Systems in Snapse are represented much like their usual graphical representation, through nodes and synapses. Each node contains a number of spikes, the rules they contain (which may be shown or hidden to reduce clutter), and their labels (See W, X, and Y in Figure 1). Neurons also change to a darker colour upon closing and return to normal colouring upon opening. Each group of spikes fired are represented by an animated signal in yellow colour, travelling along the synapse from the source neuron to the target neuron. They can be seen in the workspace in Figure 1 (See U and V).

Outputs are presented through both an output bitstring and the number of spikes received by the output neuron. As stated in Paun's tutorial in [40], the output can be interpreted differently for each SN P system. Some of the more classical ways to interpret the output include counting the distance between spikes, converting it to an integer, or looking at the spikes it receives. Outputs are displayed inside the workspace in Figure 1 (See Z) and can also be accessed in a text file stored by the application.

3.3. Rule Syntax

Snapse takes its syntax for its rules (both firing and forgetting) on the syntax used for SN P systems with extended rules defined in, i.e., for rules, the basic syntax is $E/c \to p; d$. For example, `a(aa)*/aaa -> aa;2` is a valid rule while `a(aa)*/a^3 -> a^2;2` and `a(aa)*/a3 -> a2;2` are not. This Snapse syntax is a result of keeping the format of the rules compatible with the syntax of regular expressions specified in [42].

The syntax also allows for the use of the inclusive disjunction or alternation operator "|" for example the rule, `a|aa|aaa/a->a;0`, will fire when the neuron either has 1, 2 or 3 spikes, this allows the users to check against any number of regular expressions.

A forgetting rule is specified by having p as 0 in the rule specification. Therefore the syntax is $E/c \to 0; 0$. Rule $a(aa)*/aaa \to 0; 0$ is an example of a valid forgetting rule. Note that all forgetting rules have a delay of 0. This syntax follows from the rule definition $E/a^c \to a^p; d$ in Definition 2, where forgetting rules are a special case of firing rules herein $p = 0$.

3.4. Configuration Syntax

Snapse also has configuration files it uses to save and load SN P systems. As a way to import SN P systems that were not created with Snapse, they are stored in a human-readable format. The first line is a list of all neurons, written as `neurons = [<name1>,<name2>,...,  <namen>]` where each element is a neuron name. Names are a combination of a letter and a number. The letter could be either an `N` or an `O` while the number should be an integer starting from 0 and increasing by increments of 1. `N` represents a normal, i.e., non-output, neuron while `O` represents an output neuron. The rest of the file is followed by the details of a neuron. Each neuron has the following syntax and attributes:

```
<name>{
        spikes = <int>:
        rules = {<rule1>, <rule2>,\dots, <rulen>}:
        outsynapses = [<name1>, <name2>, <name3>]:
        delay = <int>:
        storedGive = <int> <(default:0)>:
        storedConsume = <int> <(default:0)>:
        outputNeuron = <boolean>:
        position = (<float>,<float>,<float>) <(default:  (0, 0, 0))>:
}
```

`spikes` indicate the number of spikes in the neuron. Rules are stored in a list, named `rules`, with each rule following the Rule Syntax in Section 3.3. `outsynapses` contains the list of the names of each neuron it has a directed synapse to. `delay` is an integer that specifies the number of computation steps before a closed neuron fires and becomes open, with -1 as the default value if the neuron is open. `storedGive` is an integer that specifies the number of spikes a closed neuron produces when it fires (`storedGive = 0` if the neuron is open). `storedConsume` is an integer that specifies the number of spikes a closed neuron consumes when it fires (`storedConsume = 0` if the neuron is open). `outputNeuron` is a boolean that specifies whether the neuron is an output neuron or not; *True* makes it an output neuron and *False* makes it a normal neuron. `position` is the desired position of the neuron in a 3-dimensional coordinate system, the default representation for position in Unity3D. It should be noted that the z-axis only serves to position its "depth" or how close an object is to the user. An element with a lower z-axis coordinate would appear on top of an element with a higher z-axis coordinate. An example will be provided in Section 5. The configuration is designed to be human-readable so users could create SN P systems outside of the graphical editor. The non-optional attributes of the neuron (`spikes, rules, outsynapses, delay, outputNeuron`) are for specifying the elements of the system. The optional attributes, storedGive and storedConsume, are added to support saving the configuration of an SN P system, allowing users to save their work at any point in the simulation. The position attribute is for allowing the user to lay out the system how they see fit, it is also used to save the positions of the neurons so users do not need to re-position the neurons every time they load their work. The syntax took inspiration from P-Lingua and the matrix representation of SN P systems as in Section 2.3.

3.5. Graphical User Interface

The graphical interface is mainly influenced by the programs Java Formal Languages and Automata Package (JFLAP) and Snoopy. JFLAP inspired the display of the neurons and synapses, and drove the decision for the drag-and-drop functionality for the neurons. The idea for animating the spikes as they travel across the synapses came from Snoopy. The buttons were laid out with their functionalities in mind. Buttons performing similar functionalities, e.g., New Neuron and New Synapse, were placed next to each other. A button for hiding other buttons was also added to help maximize screen space for the SN P system.

3.5.1. Choice History UI

The application also tracks the rule choices made, per neuron, during points of non-determinism during the course of the system's simulation at each timestep. The history also tracks the other applicable rules that were not applied.

For example, we can have the SN P system in Figure 2 from [2]. This is an SN P system that can generate natural numbers greater than one (which will be discussed in further detail in Section 5).
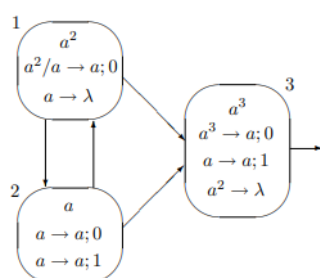
Figure 2. A spiking neural P (SN P) system that generates all natural numbers greater than one.

It can be seen that there is only one point of non-determinism in this SN P system, which is at neuron 2. If we wanted to get the output string 10001, we would need to choose the rule $a \rightarrow a; 0$ at $t = 1$ and $t = 2$ and then choose the rule $a \rightarrow a; 1$ at $t = 3$.

Figure 3 shows us exactly the history of all nondeterministic choices made. We are presented with the time step when a choice is made (labelled B), the Neuron that made this choice (labelled C), the rule that was chosen (labelled D), and the rule/s that were not chosen (labelled E) and a button (labelled A) to go back to the original configuration or $t = 0$. Clearly we can see that at $t = 1$ and $t = 2$, neuron $N1$ fires the rule $a/a \rightarrow a; 0$ and ignores the rule $a/a \rightarrow a; 1$ (by convention, we have neuron 1 as $N0$, neuron 2 as $N1$, and so on). At $t = 3$, we see neuron $N1$ choose the rule $a/a \rightarrow a; 1$ and ignore the rule $a/a \rightarrow a; 0$.
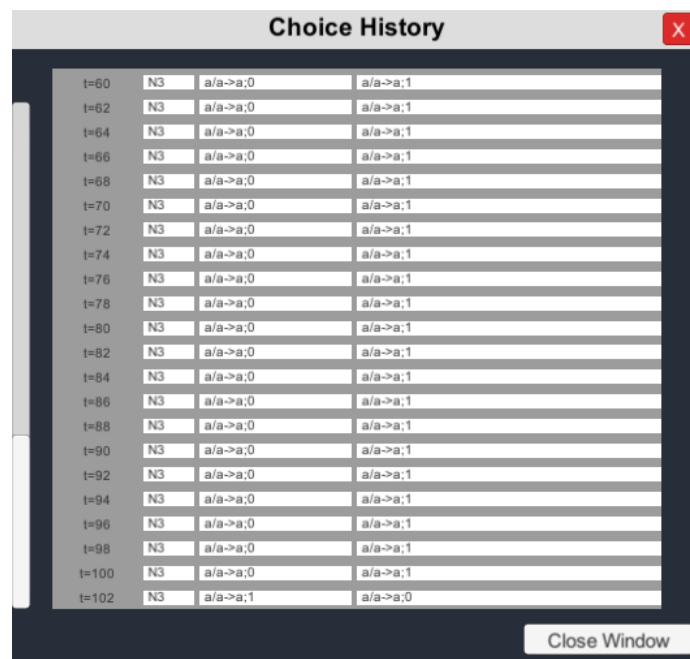

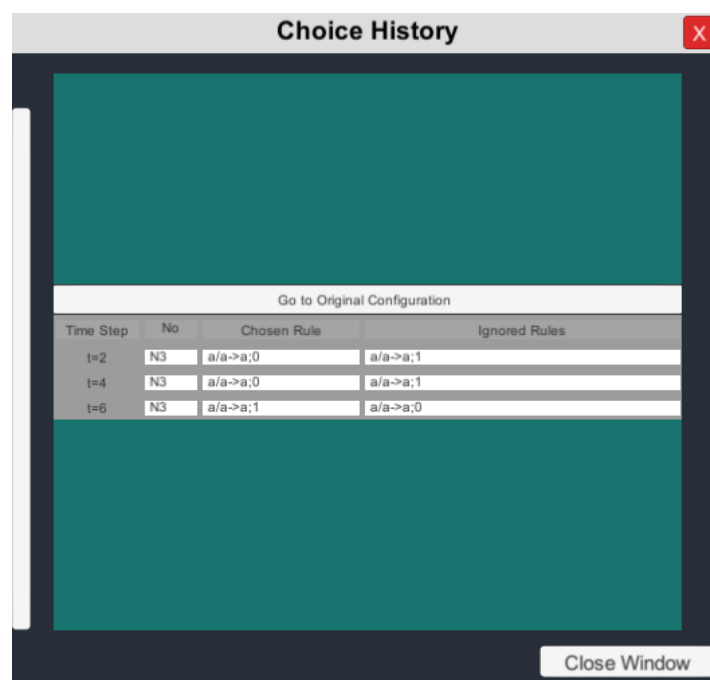**Figure 3.** Choice history.

In general, the choice history logs all non-deterministic choices for both guided and pseudorandom non-determinism. Choice history logs all choices until some timestep `t`. An example of the choice history reaching a relatively large timestep is shown in Figure 4. The choice history does not take into account whether the choice was made by guided or by pseudorandom non-determinism. The interface will look the same regardless of the setting used. In Figure 5, we changed the setting from pseudorandom to guided non-determinism at timestep `t = 3` and we reproduced the same interface when done, and vice versa.

Choice History

t=60	N3	a/a->a;0	a/a->a;1
t=62	N3	a/a->a;0	a/a->a;1
t=64	N3	a/a->a;0	a/a->a;1
t=66	N3	a/a->a;0	a/a->a;1
t=68	N3	a/a->a;0	a/a->a;1
t=70	N3	a/a->a;0	a/a->a;1
t=72	N3	a/a->a;0	a/a->a;1
t=74	N3	a/a->a;0	a/a->a;1
t=76	N3	a/a->a;0	a/a->a;1
t=78	N3	a/a->a;0	a/a->a;1
t=80	N3	a/a->a;0	a/a->a;1
t=82	N3	a/a->a;0	a/a->a;1
t=84	N3	a/a->a;0	a/a->a;1
t=86	N3	a/a->a;0	a/a->a;1
t=88	N3	a/a->a;0	a/a->a;1
t=90	N3	a/a->a;0	a/a->a;1
t=92	N3	a/a->a;0	a/a->a;1
t=94	N3	a/a->a;0	a/a->a;1
t=96	N3	a/a->a;0	a/a->a;1
t=98	N3	a/a->a;0	a/a->a;1
t=100	N3	a/a->a;0	a/a->a;1
t=102	N3	a/a->a;1	a/a->a;0

Close Window

Figure 4. Choice history at timestep $t = 102$.

Choice History

Go to Original Configuration

Time Step	No	Chosen Rule	Ignored Rules
t=2	N3	a/a->a;0	a/a->a;1
t=4	N3	a/a->a;0	a/a->a;1
t=6	N3	a/a->a;1	a/a->a;0

Close Window

Figure 5. Choice history interface with different settings.

3.6. Architecture

An SN P system with extended rules as defined in Definition 2 must have (1) a singleton alphabet, (2) neurons where each neuron contains an initial number of spikes and a finite set of rules, (3) synapses, and (4) input and output neurons. There are a few important elements to take note of to simulate a time step in terms of Snapse's architecture. The program uses a controller that communicates with the neurons and the user through the interface. The controller stores all synapses, neurons, and rules to be applied. Each neuron stores its own spikes and rules. At the start of a firing step, the controller notifies all neurons that it must check for applicable rules. All applicable rules are stored in the rules storage of the controller or immediately in the neuron if the rule is deterministic or if Snapse is in a

pseudo-random mode. The rules storage tells the controller to wait for choices when not all neurons have a chosen rule in which case, the controller must wait for user input. Once the user has chosen rules for all non-deterministic choices in guided mode, the controller notifies all neurons to simultaneously fire their chosen rules. For a graphical representation of this description, see Figure 6. This architecture applies to all software builds of Snapse, i.e., the Windows and Linux builds.
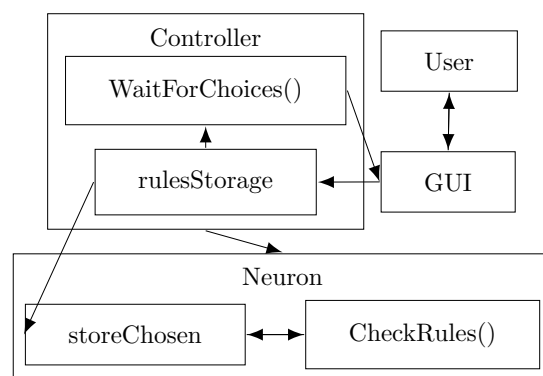
Figure 6. Snapse simulation architecture.

3.7. Simulation Algorithm

To simulate a time step for the SN P System, Snapse goes through each neuron that has an outward synapse (`outsynapse`) and checks the neuron for applicable rules. The rules are stored in a controller. When all neurons are checked, all chosen rules will be fired. In the case that `guidedMode` is set to true, the program is forced to wait for all the rule choices of the user. See Algorithm 1 for reference.

We also take a few representations of SN P systems from Sections 2.3 and 2.4. However, we break up most of these vectors and matrices between the controller and the neurons. For example, we split the configuration vector (Definition 3) into the neurons with each one holding their own spikes instead. This is done because it is much simpler to hold these values within each neuron. In Unity (the main technology used), neurons are represented by objects and contain their own attributes. In this way, a neuron can simply reference its own attributes when updating its interface.

In line 1 of Algorithm 1, we subtract from the neuron's internal timer. In lines 2 and 13, we check the neuron's internal timer which takes inspiration from the rule representation (Definition 8) in Section 2.4 and the status vector (Definition 7). This timer serves as the refractory period of the neuron. While the timer is greater than 1, the neuron remains closed (i.e., cannot apply any rule or receive any spikes), a timer less than -1 signifies that the neuron can apply a rule, and 0 means the neuron should fire. In Line 4, we check all rules and store them in *matchedRules* that we shall later pass back to the controller. In Lines 6 to 10, we emulate deterministic choices and pseudorandom mode by immediately storing the rule. Line 9 tells the neuron to wait for all neurons to select their rules before firing so that the system fires all spikes "simultaneously". The whole algorithm returns the matched rules, the chosen rule, and the neuron name to the controller.

Algorithm 1 Snapse Rule Checking Algorithm.

---

1 *neuron.timer = neuron.timer − 1;*
2 **if** *neuron.timer <= −1* **then**
3   **if** *neuron.hasOutSynpase* **then**
4   *matchedRules = neuron.checkRules();*
5   **if** *!guidedMode or matchedRules.Count == 1* **then**
6   *chosenRule ← RandomIn(matchedRules);*
          (consume, give, delay) ← parse(chosenRule);
7   *neuron.timer ← delay*
        if *delay == 0* **then**
8   *neuron.spikes = neuron.spikes − consume*
9   *waitForEnd()*
10   **end**
11   **end**
12   **end**
13 **end**
14 **if** *neuron.timer == 0* **then**
15   *neuron.spikes = neuron.spikes − consume*
16   *neuron.Fire(targets, give)*
17 **end**
18 **return** *(matchedRules, chosenRule, neuronName)*

---

4. Comparison with Other Tools

This section is dedicated to discussing the nature and the important functionalities of similar software. These technologies include visual simulators made specifically for P Systems and also graphical tools that focus on designing and animating systems using other models. To illustrate the innovations of Snapse, Table 1 is an overview and compares the functionalities of each tool discussed in this section to Snapse.

Table 1. Comparison of Snapse with other tools.

	Snapse	PLingua	JFLAP	Snoopy	MeCoSim	UPSimulator
Simulate SN P Systems	✓	✓			✓	✓
General Solution to P Systems		✓			✓	✓
Graphical Presentation	✓		✓	✓	✓	
Graphical Design	✓		✓	✓		
Pseudorandom mode	✓	✓	✓		✓	✓
Guided mode	✓					
Animation	✓			✓		

4.1. P-Lingua

P-Lingua is a programming language for membrane computing which aims to be a standard to define P systems. It and its associated tools have been developed by members of the Research Group on Natural Computing, at the University of Seville, Spain. P-Lingua was first introduced by Diaz-Pernil in [43]. It started from the idea of creating a uniform simulation framework for most (if not all) P systems. Now, it has developed to be a flexible framework that may be extended to include newer simulators. It makes use of a programming language through which users can input information on their models to be simulated. P-Lingua [44] partly inspired the syntax for the configuration files for the SN P systems.

4.2. JFLAP

Java Formal Languages and Automata Package (JFLAP) [45] is an interactive educational software written in Java for experimenting with topics in the computer science

area of formal languages and automata theory. JFLAP allows its users to create and simulate structures, such as programming a finite state machine, and experiment with proofs, such as converting a nondeterministic finite automaton (NFA) to a deterministic finite automaton (DFA).

JFLAP features a drag and drop interface allowing the creation of deterministic finite automata (DFA) and nondeterministic finite automata (NFA) among other structures. The application allows the user to create nodes and edges. The application also allows the user to set an initial and final node by right-clicking a node.

The user has various options for simulation. The user can go through the simulation step-by-step and the state of the automaton is shown in the interface. As the simulations goes through the states of the automaton, the states are shown as a series of still images. The user also has the option to make a "Fast Run" that does not go through the steps of the simulation and informs the user whether the automaton accepts or rejects the input given by the user. When a NFA is simulated, the simulator selects—in a pseudorandom manner—the series of rules to apply to reach an accepted solution, the simulator then gives the user an option to look for other paths until the accepting paths are exhausted. JFLAP partly inspired the program's interface and the pseudorandom simulation mode.

4.3. Snoopy

Snoopy [46] is a software tool to design and animate hierarchical graphs, among others Petri nets. The tool has been developed at the University of Technology in Cottbus, Dept. of Computer Science, "Data Structures and Software Dependability". The tool is in use for the verification of technical systems, especially software-based systems, as well as for the validation of natural systems, i.e., biochemical networks as metabolic, signal transduction, gene regulatory networks. Snoopy inspired the program's animation of the spikes fired from the neurons.

4.4. MeCoSim

Membrane Computing Simulator (MeCoSim) [47] is a software developed in the University of Seville that offers the users a general purpose application to model, design, simulate, analyze and verify different types of models based on P systems. It uses P-Lingua to represent P systems. MeCoSim allows viewing of graphs for verification purposes. The graph must be created via a configuration file then imported to the application to view it. The program implemented the graphic editor as an alternative to creating SN P systems using configuration file as it is more accessible for users who have not much experience working with computers. Users who prefer to create SN P systems by writing configuration files can still do so.

4.5. UPSimulator

UPSimulator [48] is a general P systems simulator developed in Chongqing University in China. The simulator extends from P-Lingua. The simulator uses its own language called UPLanguage to represent P systems. The simulator supports cell-like, tissue-like, and neuron-like P systems. The simulator however does not support graphic visualization for the P systems.

5. A Few Examples

To better illustrate how Snapse works, a few examples are presented in this section using the program. For reference, all examples discussed here are available in the github repository (https://github.com/reysterf/SNP-Editor). An SN P system in Figure 2 from [2] generates all natural numbers greater than 1. As discussed in Section 3.4, the configuration file should start with the neurons. In Figure 2, there are three neurons in the system. However, we add another neuron to emulate the environment which we designate as the output neuron. As such, the first line of the configuration file should

be `neurons = [N0, N1, N2, O3]`: with `N0, N1, N2` as the three neurons and `O3` as our environment or output neuron.

Following the configuration syntax presented earlier, each neuron should have `spikes`, `rules, outsynapses, delay, storedGive, storedConsume, outputNeuron`, and `position`. `spikes` must be equal to the number of initial spikes in the system (which for `N0` is 2) so we write `spikes = 2:`. `rules` should cover all the rules in the neuron. In `N0`, those are $a^2/a \rightarrow a; 0$ and $a \rightarrow \lambda$. The rule syntax in Section 3.3 reformats spike representations a^p into a string over the alphabet {a} whose length is equal to the number of spikes p. We write the `rules` of `N0` as `rules = {[aa/a->a;0], [a/a->;0]}`. `outsynapses` should contain all the neurons the current neuron has a synapse directed towards (in the case of `N0`, that is `N1` and `N2`) which is written as `outsynapses = [N1, N2]`. `delay, storedGive`, and `storedConsume` are set to their default values as shown in Section 3.4. `outputNeuron` should be set to false for all non-output neurons (in this example: `N0, N1`, and `N2`). `position` could be set to the default (`0, 0, 0`) for all neurons. Alternatively, it could be set to any reasonable 3-dimensional coordinate to avoid overlapping. This could be done for both `N1` and `N2`, as well. Snapse uses 3-dimensional coordinates, as opposed to 2-dimensional coordinates, because it matches how Unity represents positions internally.

While all of the attributes of a neuron are also present in the output neuron, there are a few key differences. Since `N3` is an output neuron, `outputNeuron` should be set to `True`. Additionally, we add the lines `storedReceived = 0` and `bitstring = null`: before `position` and after `outputNeuron`. A sample configuration file for the SN P system in Figure 2 is as follows:

In Snapse, this SN P system would look something like Figure 7. In the first step, `N2` will fire a spike towards the output neuron, giving it a 1. At each time step, `N0` and `N2` will send a spike to `N2` which will be forgotten because of the rule `aa/aa->0;0`. As long as `N1` chooses the rule `a/a->a;0`, no spike will be sent to the output neuron. When `N1` chooses the rule `a/a->a;1`, only one spike will be sent to `N2`, which will close `N2`. The spike that will be sent by `N1` in the next step will not be received since `N2` is closed. Finally, `N2` will release its spike.

To interpret the output of this specific example, we must look at the output bitstring at the end of the simulation. For this example, in particular, the number can be gathered from taking the distance between the first two "1" symbols in the output bitstring. If at the first step `N1` chooses the rule `a/a->a;1`, the resulting bitstring will be 101, i.e., the distance between the two "1" symbols and the result is the number 2. This can be seen in the Appendix in Figure A3. If `N1` chooses the rule `a/a->a;1` at t=2 instead, the resulting bitstring is 1001, i.e., the result is the number 3. Alternatively, if `N1` chooses the rule `a/a->a;1` at time step 3, the resulting bitstring is 10001, i.e., the result is 4, and so on. Thus, the SN P sytem can generate any natural number greater than or equal to 2. A more detailed description is laid out by Ionescu et al in [2]. Screenshots of Snapse for this example can be seen in Appendix A.1.

We also take an example of an increasing comparator that can be seen in Figure A4 in the Appendix A.2. Ceterchi and Tomescu's in [49] discuss how an increasing comparator can be represented as an SN P system. The comparator is used to create a network of parallel and bitonic sorters of natural numbers. For this example, we compare the numbers 8 and 5, respectively. The configuration file can be found in Appendix A.4. This SN P system has two output neurons as seen in Appendix A.3 where the top output neuron would represent the smaller of the two numbers after comparing is done.

The last SN P system we consider in this section is a bit adder. This SN P system provides the sum of two natural numbers in binary form. Gutierrez-Naranjo and Leporati provide an SN P system for such an adder in [50]. The example provided in Appendix A.5 adds the numbers 2 and 3, in binary format. Snapse does not allow an input of bitstrings which shall be discussed in Section 6. Instead, a couple of nodes are created to simulate this as seen in Figure A7. The configuration file can be found in Appendix A.6

```
neurons = [N0, N1, N2, O3]:
N0{
        spikes = 2:
        rules = {[aa/a->a;0], [a/a->0;0]}:
        outsynapses = [N1, N2]:
        delay = -1:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = False:
        position = <(-3,1.5,0)>:
}
N1{
        spikes = 1:
        rules = {[a/a->a;0], [a/a->a;1]}:
        outsynapses = [N0, N2]:
        delay = -1:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = False:
        position = <(-3,-1.5,0)>:
}
N2{
        spikes = 3:
        rules = {[aaa/aaa->a;0], [a/a->a;1], [aa/aa->0;0]}:
        outsynapses = [O3]:
        delay = -1:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = False:
        position = <(0,0,0)>:
}
N3{
        spikes = 0:
        rules = {[]}:
        outsynapses = []:
        delay = -1:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = True:
        storedReceived = 0:
        bitString = null:
        position = <(3,0,0)>:
}
```
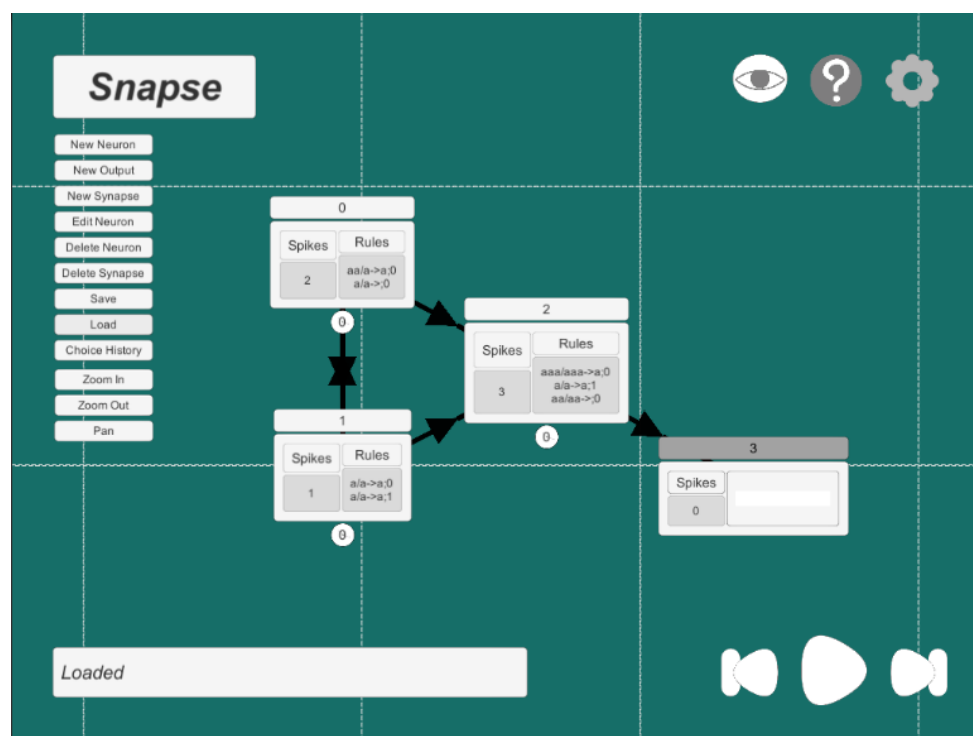
Figure 7. The SN P system in Figure 2 in Snapse.

6. Final Remarks

In this paper, we presented Snapse, a tool which provides users the ability to create, design, and modify their models through graphical means. We discuss in Section 1, that although research regarding SN P systems is active, their development could be made even easier and faster using a visual tool. Snapse takes a step in this direction by offering many features that ease the difficulty of customizing SN P systems. Section 3 lists down these features. Aside from storing work, it gives researchers the ability to control the run of their models through the introduction of guided non-determinism and editing between simulations. In Section 5, three useful examples are presented to show how Snapse can be used by experts in SN P systems. Additionally, the graphical presentation and visual indicators such as in Figure 1 make Snapse more accessible to people new to SN P systems.

While this work is a preliminary one to introduce the visual tool Snapse for SN P systems, the Snapse version as of now can be used, at least in part, in some applications. Aside from the examples in Section 5 and their extensions or generalization, Snapse could be used in (parts of) the systems for skeletonizing images as in [27,28], aiding in the design of some systems in [30,32–34], computational biology in [37], and other applications in [26,38].

Next we list a few and natural directions for further work. The algorithm and the architecture of Snapse may be optimized to allow it to run faster when simulating larger systems. In particular, allowing users to use bitstrings as inputs in place of spike trains, and to use regular expressions that use variables such as that used in the Encoder in [51] are helpful. We aim to support the use of input neurons, in order to simulate SN P systems as transducers as in [22,23] and more recently in [25]. Snapse can also be extended to allow support for other variants of SN P systems like SN P systems with rules on synapses [52], with anti-spikes [53] and dynamic variants such as [54–56].

There is also room for various improvements with the user experience e.g., adding more input/output options, such as input neurons or taking in spike trains from a (perhaps bit string) file, allowing the use of exponents in the rule declarations, adding support for systems written in P-Lingua and other formats, or allowing the user to navigate and jump to specific timesteps through the simulation choice history via a tree. It is also interesting to be able to use Snapse to work with parallel and text-based simulators such as those

from [11,12]. In simulating larger systems, the suggestions for optimizations in [12,57] are likely to be useful also.

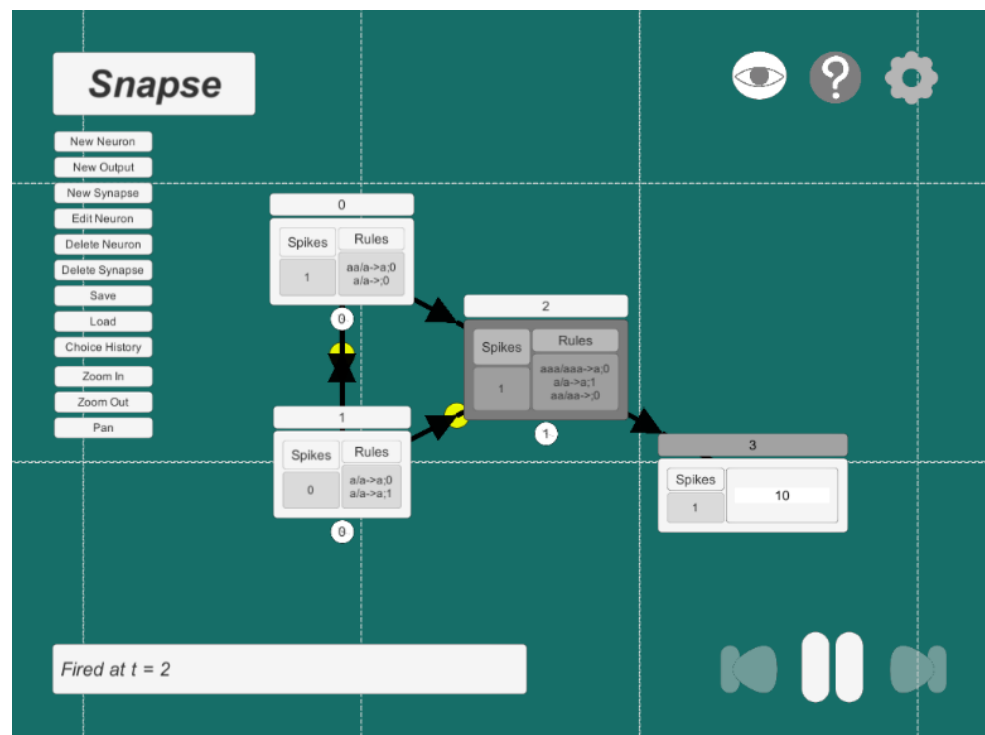**Conflicts of Interest:** The authors declare no conflict of interests.

Appendix A. Examples of SN P Systems in Snapse

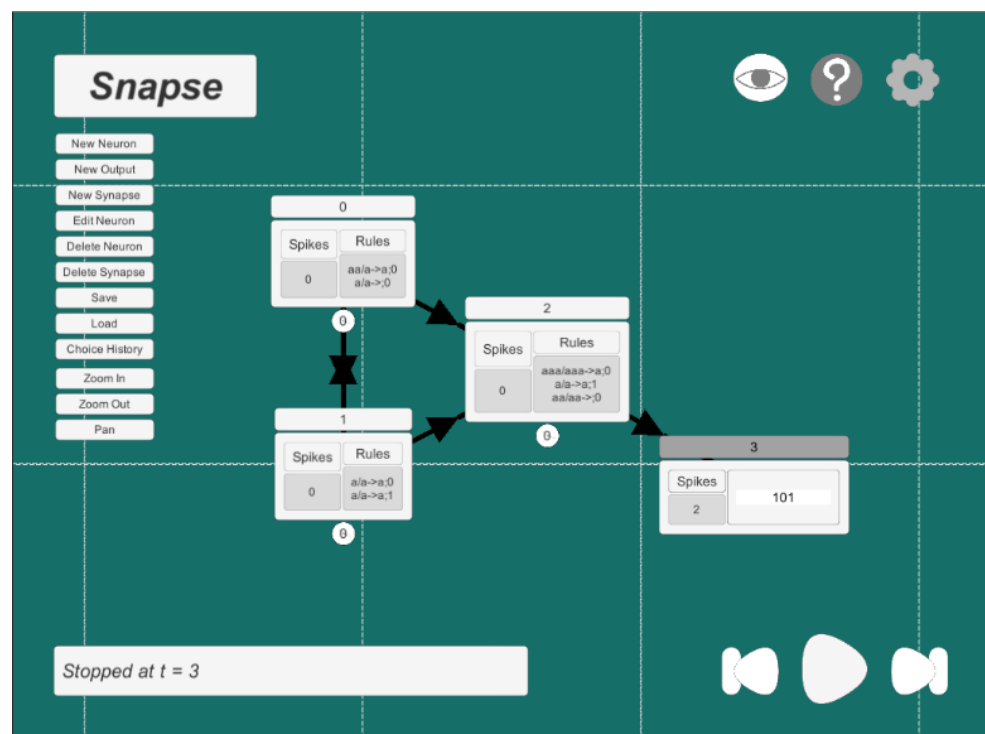*Appendix A.1. Example of an SN P System That Generates all Natural Numbers Greater than One in Snapse*

See Figures A1–A3 .
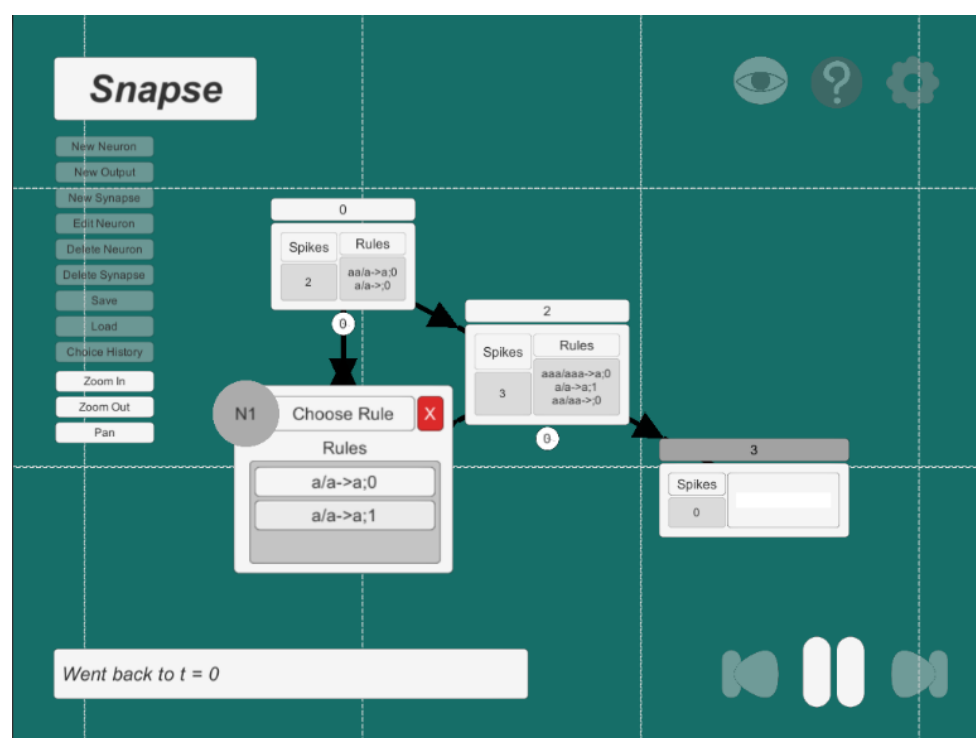
Figure A1. An example of a choice in an SN P system that generates all numbers greater than one in Snapse.
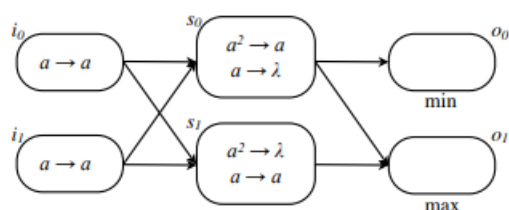
Figure A2. An example of neurons firing in an SN P system that generates all numbers greater than one in Snapse.

Figure A3. An example an SN P system that generates all numbers greater than one at the end of its simulation in Snapse.

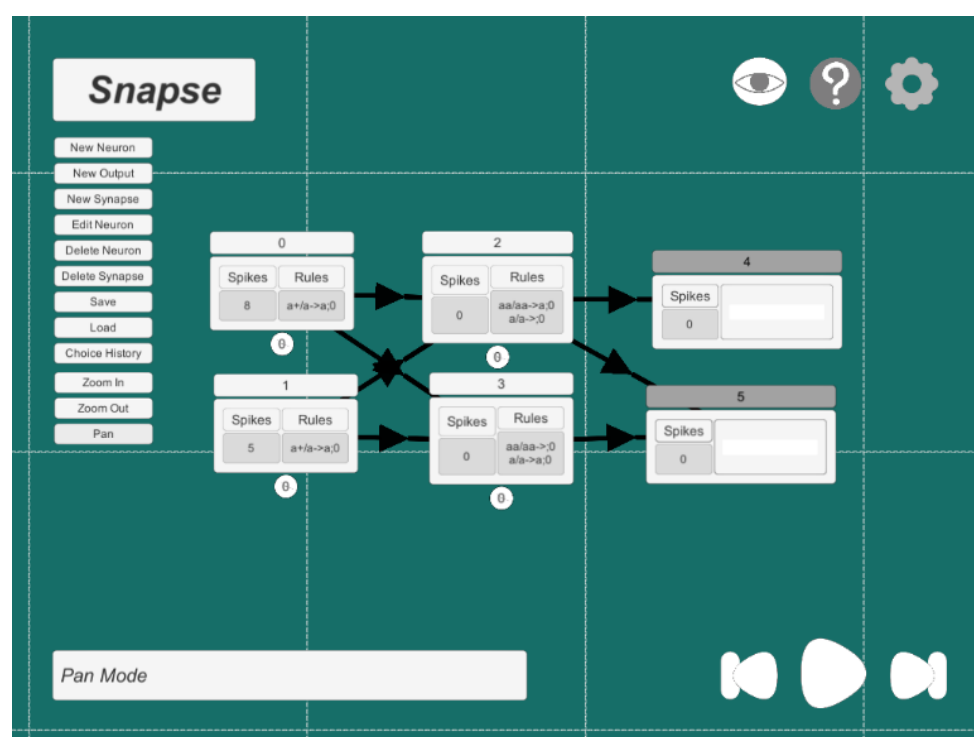*Appendix A.2. Graphical Representation of an Increasing Comparator*

See Figure A4.

Figure A4. An increasing comparator presented graphically.

Appendix A.3. An example of an increasing comparator in Snapse

See Figure A5 for an example that compares the numbers 8 and 5. See Figure A6 for the resulting outputs after simulation.

Figure A5. An example of an increasing comparator that compares 8 and 5 in Snapse.
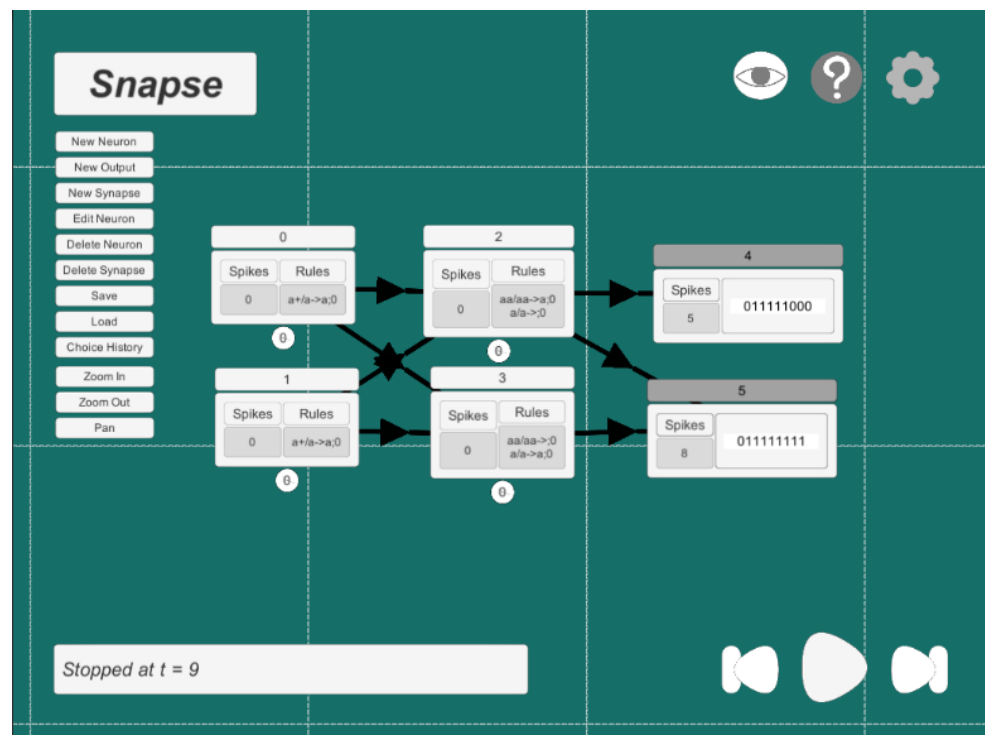
Figure A6. An example of an increasing comparator in Snapse after simulation.

Appendix A.4. An Example of a Configuration File of an Increasing Comparator in Snapse

```
    neurons = [N0, N1, N2, N3, O4, O5]:
N0{
        spikes = 8:
        rules = {[a+/a->a;0]}:
        outsynapses = [N2, N3]:
        delay = 0:
        storedGive = 1:
        storedConsume = 1:
        outputNeuron = False:
        position = (-3,1,0):
}
N1{
        spikes = 5:
        rules = {[a+/a->a;0]}:
        outsynapses = [N2, N3]:
        delay = 0:
        storedGive = 1:
        storedConsume = 1:
        outputNeuron = False:
        position = (-3,-1,0):
}
N2{
        spikes = 0:
        rules = {[aa/aa->a;0], [a/a->0;0]}:
        outsynapses = [O4, O5]:
        delay = 0:
        storedGive = 1:
        storedConsume = 2:
        outputNeuron = False:
        position = (0,1,0):
}
```

```
N3{
        spikes = 0:
        rules = {[aa/aa->0;0], [a/a->a;0]}:
        outsynapses = [O5]:
        delay = 0:
        storedGive = 0:
        storedConsume = 2:
        outputNeuron = False:
        position = (0,-1,0):
}
N4{
        spikes = 0:
        rules = {[]}:
        outsynapses = []:
        delay = -4:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = True:
        storedReceived = 0:
        bitString = null:
        position = (3,1,0):
}
N5{
        spikes = 0:
        rules = {[]}:
        outsynapses = []:
        delay = -4:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = True:
        storedReceived = 0:
        bitString = null:
        position = (3,-1,0):
}
```

Appendix A.5. An Example of a Bit Adder in Snapse

See Figure A7 for an example that adds the numbers 2 and 3. See Figure A8 for the resulting outputs after simulation.
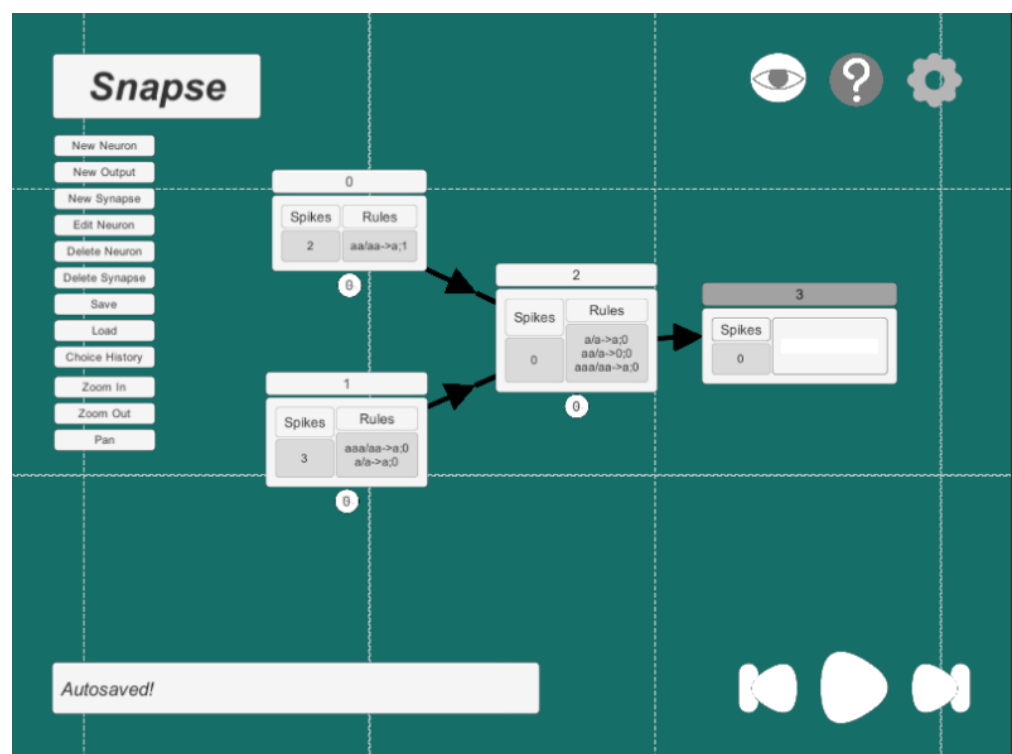
Figure A7. An example of a bit adder that adds 2 and 3 in Snapse.
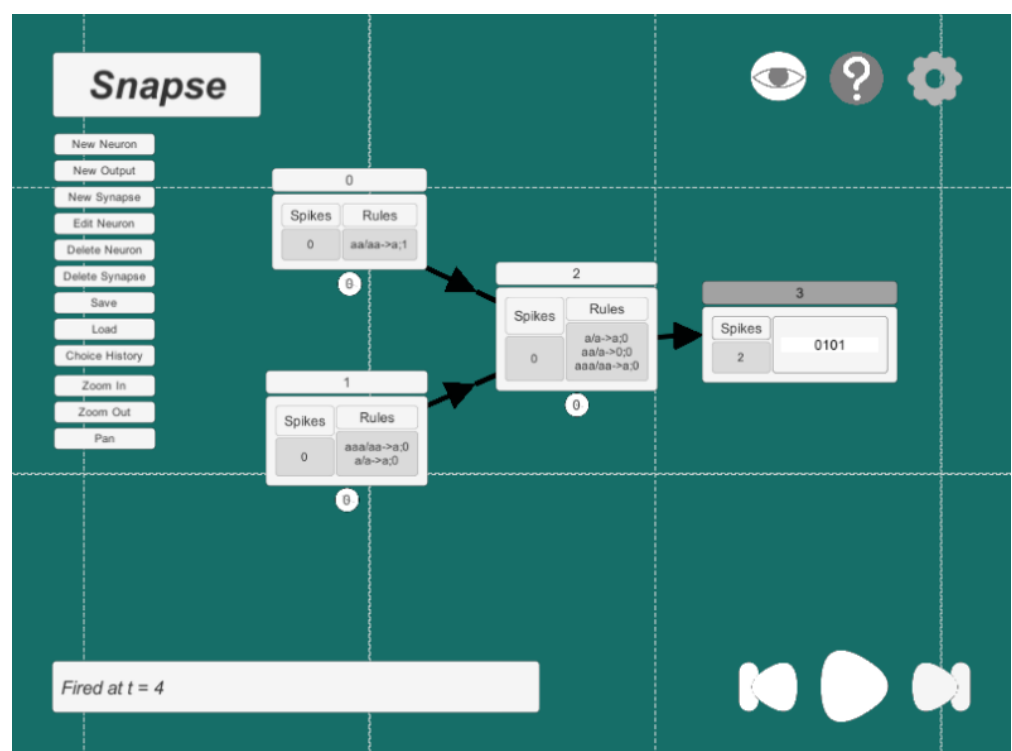

Figure A8. An example of a bit adder in Snapse after simulation.

Appendix A.6. An Example of a Configuration File of a Bit Adder in Snapse

```
neurons = [N0, N1, N2, O3]:
N0{
        spikes = 2:
        rules = {[aa/aa->a;1]}:
        outsynapses = [N2]:
        delay = 0:
        storedGive = 1:
        storedConsume = 2:
        outputNeuron = False:
        position = (-3,1,0):
}
N1{
        spikes = 3:
        rules = {[aaa/aa->a;0], [a/a->a;0]}:
        outsynapses = [N2]:
        delay = 0:
        storedGive = 1:
        storedConsume = 1:
        outputNeuron = False:
        position = (-3,-1,0):
}
N2{
        spikes = 0:
        rules = {[a/a->a;0], [aa/a->0;0], [aaa/aa->a;0]}:
        outsynapses = [O3]:
        delay = 0:
        storedGive = 1:
        storedConsume = 1:
        outputNeuron = False:
        position = (0,0,0):
}
N3{
        spikes = 0:
        rules = {[]}:
        outsynapses = []:
        delay = 0:
        storedGive = 0:
        storedConsume = 0:
        outputNeuron = True:
        storedReceived = 0:
        bitString = null:
        position = (3,0,0):
}
```

References

1. Păun, G. Computing with membranes. *J. Comput. Syst. Sci.* **2000**, *61*, 108–143.
2. Ionescu, M.; Păun, G.; Yokomori, T. Spiking Neural P Systems. *Fundam. Inform.* **2006**, *71*, 279–308.
3. Calude, C.S.; Păun, G. *Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing*; Taylor & Francis/Hemisphere: 1900 Frost Rd. Suite 101 Bristol, PA, United States 2001.
4. Păun, G. P Systems with Active Membranes: Attacking NP-Complete Problems. *J. Autom. Lang. Comb.* **2001**, *6*, 75–90.
5. Ciobanu, G.; Păun, G.; Pérez-Jiménez, M.J. *Applications of Membrane Computing*; Springer: Berlin/Heidelberg, Germany, 2006; Volume 17.
6. Frisco, P.; Gheorghe, M.; Pérez-Jiménez, M.J. *Applications of Membrane Computing in Systems and Synthetic Biology*; Springer: Berlin/Heidelberg, Germany, 2014.
7. Paun, G.; Rozenberg, G.; Salomaa, A. *The Oxford Handbook of Membrane Computing*; Oxford University Press, Inc.: New York, NY, USA, 2010.

8.   Leporati, A.; Mauri, G.; Zandron, C.; Păun, G.; Pérez-Jiménez, M. Uniform solutions to SAT and Subset Sum by spiking neural P systems. *Nat. Comput.* **2009**, *8*, 681–702.

9.   Cabarle, F.G.C.; Adorna, H.N.; Martínez del Amor, M.Á.; Pérez Jiménez, M.d.J. Improving GPU simulations of spiking neural P systems. *Rom. J. Inf. Sci. Technol.* **2012**, *15*, 5–20.

10. Carandang, J.; Villaflores, J.M.B.; Cabarle, F.G.C.; Adorna, H.N.; Martínez-del Amor, M.A. CuSNP: Spiking neural P systems simulators in CUDA. *Rom. J. Inf. Sci. Technol.* **2017**, *20*, 57–70.

11. Carandang, J.P.; Cabarle, F.G.C.; Adorna, H.N.; Hernandez, N.H.S.; Martínez-del Amor, M.Á. Handling non-determinism in spiking neural P systems: Algorithms and simulations. *Fundam. Inform.* **2019**, *164*, 139–155.

12. Aboy, B.C.D.; Bariring, E.J.A.; Carandang, J.P.; Cabarle, F.G.C.; Cruz, R.T.D.L.; Adorna, H.N.; Martínez-del-Amor, M.A. Optimizations in CuSNP Simulator for Spiking Neural P Systems on CUDA GPUs. In Proceedings of the 2019 International Conference on High Performance Computing Simulation (HPCS), Dublin, Ireland, 15–19 July 2019; pp. 535–542.

13. Cabarle, F.G.C.; de la Cruz, R.T.A.; Cailipan, D.P.P.; Zhang, D.; Liu, X.; Zeng, X. On solutions and representations of spiking neural P systems with rules on synapses. *Inf. Sci.* **2019**, *501*, 30–49.

14. Ibarra, O.H.; Păun, A.; Păun, G.; Rodríguez-Patón, A.; Sosík, P.; Woodworth, S. Normal forms for spiking neural P systems. *Theor. Comput. Sci.* **2007**, *372*, 196–217.

15. Pan, L.; Păun, G. Spiking neural P systems: An improved normal form. *Theor. Comput. Sci.* **2010**, *411*, 906–918.

16. Macababayao, I.C.H.; Cabarle, F.G.C.; dela Cruz, R.T.A.; Adorna, H.N.; Zeng, X. Notes on Improved Normal Forms of Spiking Neural P Systems and Variants; Pre-Proc. In Proceedings of the Asian Conference on Membrane Computing (ACMC2019), Xiamen, China, 14–17 November 2019.

17. Zeng, X.; Adorna, H.; Martínez-del Amor, M.; Pan, L.; Pérez-Jiménez, M. Matrix Representation of Spiking Neural P Systems. In Proceedings of the International Conference on Membrane Computing, Jena, Germany, 24–27 August 2010; Volume 6501, pp. 377–391.

18. Verlan, S.; Freund, R.; Alhazov, A.; Ivanov, S.; Pan, L. A formal framework for spiking neural P systems. *J. Membr. Comput.* **2020**, *2*, 355–368.

19. Jimenez, Z.B.; Cabarle, F.G.C.; de la Cruz, R.T.A.; Buño, K.C.; Adorna, H.N.; Hernandez, N.H.S.; Zeng, X. Matrix representation and simulation algorithm of spiking neural P systems with structural plasticity. *J. Membr. Comput.* **2019**, *1*, 145–160.

20. Chen, H.; Freund, R.; Ionescu, M.; Păun, G.; Pérez-Jiménez, M.J. On string languages generated by spiking neural P systems. *Fundam. Inform.* **2007**, *75*, 141–162.

21. Cabarle, F.G.C.; Adorna, H.N. On Structures and Behaviors of Spiking Neural P Systems and Petri Nets. In *CMC 2012: Membrane Computing, Proceedings of the International Conference on Membrane Computing, Chişinău, Republic of Moldova, 20–23 August 2013*; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7762, pp. 145–160.

22. Ibarra, O.H. Pérez-Jiménez, M.J.; Yokomori, T. On spiking neural P systems. *Nat. Comput.* **2010**, *9*, 475–491.

23. Cabarle, F.G.C.; Adorna, H.N.; Pérez-Jiménez, M.J. Notes on spiking neural P systems and finite automata. *Nat. Comput.* **2016**, *15*, 533–539.

24. De la Cruz, R.T.A.; Cabarle, F.G.; Adorna, H.N. Generating context-free languages using spiking neural P systems with structural plasticity. *J. Membr. Comput.* **2019**, *1*, 161–177.

25. Adorna, H.N. Computing with SN P systems with I/O mode. *J. Membr. Comput.* **2020**, *2*, 230–245.

26. Rong, H.; Wu, T.; Pan, L.; Zhang, G. Spiking neural P systems: Theoretical results and applications. In *Enjoying Natural Computing*; Springer: Berlin/Heidelberg, Germany, 2018; pp. 256–268.

27. Díaz-Pernil, D.; Peña-Cantillana, F.; Gutiérrez-Naranjo, M.A. A parallel algorithm for skeletonizing images by using spiking neural P systems. *Neurocomputing* **2013**, *115*, 81–91.

28. Song, T.; Pang, S.; Hao, S.; Rodríguez-Patón, A.; Zheng, P. A parallel image skeletonizing method using spiking neural P systems with weights. *Neural Process. Lett.* **2019**, *50*, 1485–1502.

29. Liu, X.; Li, Z.; Suo, J.; Liu, J.; Min, X. A uniform solution to integer factorization using time-free spiking neural P system. *Neural Comput. Appl.* **2015**, *26*, 1241–1247.

30. Ochirbat, O.; Ishdorj, T.O.; Cichon, G. An error-tolerant serial binary full-adder via a spiking neural P system using HP/LP basic neurons. *J. Membr. Comput.* **2020**, *2*, 42–48.

31. Wang, H.; Zhou, K.; Zhang, G.; Paul, P.; Duan, Y.; Qi, H. The Application of Weighted Spiking Neural P Systems with Rules on Synapses for Breaking RSA Encryption. In Proceedings of the Branch of International Conference on Membrane Computing (ACMC2018), Auckland, New Zealand, 10–14 December 2018; p. 191.

32. Moredo, C.; Supelana, R.; Cailipan, D.; Cabarle, F.; de La Cruz, R.; Adorna, H.; Zeng, X.; Martínez-Del-Amor, M.A. Framework for Evolving Spiking Neural P Systems with Rules on Synapses; Pre-Proc. In Proceedings of the Asian Conference on Membrane Computing (ACMC2019), Xiamen, China, 14–17 November 2019.

33. Zarate, C.C.R.; Cabarle, F.G.C.; Macababayao, I.C.; la Cruz, R.T.D. Evolving Spiking Neural P Systems by Fixing Neurons, and Varying Rules and Synapses. *Philipp. Comput. J.* **2020**, *14*, 21–30.

34. Casauay, L.J.; Macababayao, I.C.H.; Cabarle, F.G.C.; de la Cruz, R.T.A.; Adorna, H.N.; Zeng, X.; Martínez-Del-Amor, M.Á. A Framework for Evolving Spiking Neural P Systems. *Int. J. Unconv. Comput.* **2020**, *14*, in press.

35. Song, T.; Pan, L.; Wu, T.; Zheng, P.; Wong, M.D.; Rodríguez-Patón, A. Spiking neural P systems with learning functions. *IEEE Trans. Nanobiosci.* **2019**, *18*, 176–190.

36. Ma, T.; Hao, S.; Wang, X.; Rodríguez-Patón, A.A.; Wang, S.; Song, T. Double Layers Self-Organized Spiking Neural P Systems With Anti-Spikes for Fingerprint Recognition. *IEEE Access* **2019**, *7*, 177562–177570.
37. Chen, Z.; Zhang, P.; Wang, X.; Shi, X.; Wu, T.; Zheng, P. A computational approach for nuclear export signals identification using spiking neural P systems. *Neural Comput. Appl.* **2018**, *29*, 695–705.
38. Fan, S.; Paul, P.; Wu, T.; Rong, H.; Zhang, G. On Applications of Spiking Neural P Systems. *Appl. Sci.* **2020**, *10*, 7011.
39. Pérez-Hurtado, I.; Orellana-Martín, D.; del Amor, M.Á.M.; Valencia-Cabrera, L.; Riscos-Núñez, A.; Pérez-Jiménez, M.J. 11 years of P-Lingua: A backward glance. In Proceedings of the 20th International Conference on Membrane Computing (CMC20), Curtea de Arges, Romania, 5–8 August 2019; pp. 451–462.
40. Păun, G. Spiking neural P systems. A Tutorial. *Bull. Eur. Assoc. Theor. Comput. Sci.* **2007**, *91*, 145–159.
41. Chen, H.; Ishdorj, T.O.; Paun, G.; Pérez Jiménez, M.d.J. Spiking neural P systems with extended rules. In Proceedings of the Fourth Brainstorming Week on Membrane Computing, Sevilla, Spain, 30 January–3 February 2006; Volume I, pp. 241–265.
42. Microsoft. NET Regular Expressions. 2020. Available online: https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expressions (accessed on 26 November 2020).
43. Díaz Pernil, D.; Pérez Hurtado de Mendoza, I.; Pérez Jiménez, M.d.J.; Riscos Núñez, A. P-lingua: A programming language for membrane computing. In Proceedings of the Sixth Brainstorming Week on Membrane Computing, Sevilla, Spain, 4–8 February 2008; pp. 135–155.
44. Macías-Ramos, L.F.; Pérez-Hurtado, I.; García-Quismondo, M.; Valencia-Cabrera, L.; Pérez-Jiménez, M.J.; Riscos-Núñez, A. A P–Lingua Based Simulator for Spiking Neural P Systems. In *Membrane Computing*; Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 257–281.
45. Rodger, S.H. *JFLAP: An Interactive Formal Languages and Automata Package*; Jones and Bartlett Publishers, Inc.: Boston, MA, USA, 2006.
46. Heiner, M.; Herajy, M.; Liu, F.; Rohr, C.; Schwarick, M. Snoopy—A Unifying Petri Net Tool. In *Application and Theory of Petri Nets*; Haddad, S., Pomello, L., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 398–407.
47. Pérez-Hurtado, I.; Valencia-Cabrera, L.; Pérez-Jiménez, M.J.; Colomer, M.A.; Riscos-Núñez, A. MeCoSim: A general purpose software tool for simulating biological phenomena by means of P systems. In Proceedings of the 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), Changsha, China, 23–26 September 2010; pp. 637–643.
48. Guo, P.; Quan, C.; Ye, L. UPSimulator: A general P system simulator. *Knowl.-Based Syst.* **2019**, *170*, 20–25.
49. Ceterchi, R.; Tomescu, A.I. *Spiking Neural P Systems—A Natural Model for Sorting Networks*; In Proceedings of the Sixth Brainstorming Week on Membrane Computing, 93-105. Sevilla, ETS de Ingeniería Informática, Sevilla, Spain, 4–8 February 2008.
50. Gutiérrez Naranjo, M.Á.; Leporati, A. Performing arithmetic operations with spiking neural P systems. In Proceedings of the Seventh Brainstorming Week on Membrane Computing, Sevilla, Spain, 2–6 February 2009; Volume I, pp. 181–198. .
51. Wang, F.; Zhou, K.; Qi, H. Using an SN P System to Compute the Product of Any Two Decimal Natural Numbers. In Proceedings of the International Conference on Bio-Inspired Computing: Theories and Applications, Harbin, China, 1–3 December 2017; He, C., Mo, H., Pan, L., Zhao, Y., Eds.; Springer: Singapore, 2017; pp. 194–206.
52. Song, T.; Pan, L.; Păun, G. Spiking neural P systems with rules on synapses. *Theor. Comput. Sci.* **2014**, *529*, 82–95.
53. Ibarra, O.H.; Păun, A.; Rodríguez-Patón, A. Sequential SNP systems based on min/max spike number. *Theor. Comput. Sci.* **2009**, *410*, 2982–2991.
54. Pan, L.; Păun, G.; Pérez-Jiménez, M.J. Spiking neural P systems with neuron division and budding. *Sci. China Inf. Sci.* **2011**, *54*, 1596.
55. Cabarle, F.G.C.; Adorna, H.N.; Pérez-Jiménez, M.J.; Song, T. Spiking neural P systems with structural plasticity. *Neural Comput. Appl.* **2015**, *26*, 1905–1917.
56. Cabarle, F.G.C.; Adorna, H.N.; Jiang, M.; Zeng, X. Spiking neural P systems with scheduled synapses. *IEEE Trans. Nanobiosci.* **2017**, *16*, 792–801.
57. Martínez del Amor, M.Á.; Orellana Martín, D.; Cabarle, F.G.C.; Pérez Jiménez, M.d.J.; Adorna, H.N. Sparse-matrix representation of spiking neural P systems for GPUs. In Proceedings of the BWMC 2017: 15th Brainstorming Week on Membrane Computing, Sevilla, Spain, 31 January–3 February 2017; pp. 161–170 .