Spiking Neural P Systems: How to Use Them, and How to Design Them

Alberto Leporati*

University of Milan-Bicocca, Department of Informatics, Systems and Communication, Edificio U14 (ABACUS), Viale Sarca 336, 20126 Milano, Italy

Abstract

Spiking neural P systems have been introduced in 2006 as the first neural-like model of membrane systems. Similar to spiking neural networks, considered as the third generation of neural network models, they constitute a computational model inspired by the spiking mechanism of biological neurons. Among the advantages, compared to traditional neural networks, there is the promise of implementations that are much more energy efficient.

Over the years, spiking neural P systems have been investigated from both the computational power and applications points of view. In this extended abstract, after recalling what spiking neural P systems are, and how they work, we will address the problem of designing systems aimed at performing a prescribed computational task. In particular, we will present a framework that can be used to produce spiking neural P systems from a specification written in the WHILE programming language, and an evolutionary technique that evolves a family of spiking neural P circuits that "learn" a partially specified multi-output Boolean function. We will conclude with some considerations related to the relationship between spiking neural P systems and other models of spiking neural networks and binarized networks, machine learning algorithms, and will provide some possible research directions.

Keywords

Spiking neural P systems, Spiking neural P circuits, WHILE programming language, Evolutionary algorithms, Genetic algorithm

1. Introduction

Spiking neural P systems (SN P systems, for short) have been introduced in [1] as the first neural-like model of membrane systems [2, 3], abstracted from the spiking mechanism of biological neurons. Inspired by spiking neurons, SN P systems and variants are considered as the third generation of neural network models, similar to spiking neural networks [4]. Structurally, an SN P system is a directed graph composed of several neurons, where the nodes are spiking neurons, and the directed arcs indicate the synaptic connections between neurons. Depending on predefined conditions, neurons emit spike objects (which represent the electrical impulses emitted by biological neurons); these spikes are sent to adjacent neurons via synaptic connections. So doing, a computational model can be defined, that takes appropriately encoded input values from the environments and computes appropriately encoded output values.

Over the years, several variants have been studied that are inspired by different biological mechanisms. These include astrocytes [5], anti-spikes [6], the presence of multiple channels [7], inhibition rules [8], dendrite mechanisms [9], polarities [10], structural plasticity [11], communication on request [12], autapses [13], delay on synapses [14], plasmids [15], dynamic threshold [16], and coupling mechanism [17]. Most of these variants have been proven to work as Turing universal devices for generating or accepting sets of numbers, for computing functions, or for generating languages. Furthermore, SN P systems and variants have been used for some real-world application problems such as image fusion [18, 19], image segmentation [20], edge detection [21, 22, 23], medical image processing [24],

ITAT 2026: Information technologies – Applications and Theory, Workshop on Natural Computing, September 25–29, 2026, Biele Karpaty, Slovakia

*Corresponding author.

alberto.leporati@unimib.it (A. Leporati)

https://www.unimib.it/alberto-ottavio-leporati (A. Leporati)

0000-0002-8105-4371 (A. Leporati)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4323/invited3.pdf

financial time series prediction [25], sentiment analysis [26, 27, 28], recommendation systems [29], numerical optimization [30, 31], and classification problems [32].

In this extended abstract, after recalling what spiking neural P systems are, and how they work, we will address the problem of designing systems that perform a prescribed computational task. In particular, we will present a framework that can be used to produce spiking neural P systems from a specification written in the WHILE programming language, and an evolutionary technique that evolves a family of spiking neural P circuits that "learn" a partially specified multi-output Boolean function. We will conclude with some considerations related to the relationship between spiking neural P systems and other models of spiking neural networks and binarized networks, machine learning algorithms, and will provide some possible research directions.

2. Spiking Neural P Systems

Let us start by recalling the basic definition of spiking neural P systems (SN P systems, for short) given in [1].

Definition 1. A *Spiking Neural P system* of degree $m \geqslant 1$ is a construct of the form

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

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ are *neurons*, of the form $\sigma_i = (n_i, R_i), 1 \leqslant i \leqslant m$, where:
   a) $n_i \geqslant 0$ is the *initial number* of spikes contained in σ_i;
   b) R_i is a finite set of *rules* of one of the following two forms:
      (1) $E/a^c \rightarrow a; d$, where E is a regular expression over the alphabet $\{a\}$ and $c \geqslant 1$, $d \geqslant 0$ are natural numbers;
      (2) $a^s \rightarrow \lambda$, where $s \geqslant 1$ is a natural number, with the restriction that for each rule $E/a^c \rightarrow a; d$ of type (1) from R_i, we have $a^s \notin L(E)$, where $L(E)$ is the regular language defined by E;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $i \neq j$ for all $(i, j) \in syn$, $1 \leqslant i, j \leqslant m$, is the set of *synapses* between neurons;
4. $i_{in}, i_{out} \in \{1, 2, \ldots, m\}$ indicate the *input* and the *output* neuron, respectively.

The rules of type (1) are called *spiking* or also *firing* rules, while those of type (2) are called *forgetting* rules. The rules can be applied in each neuron to consume, produce and send spikes, at time steps marked by a global discrete clock. The system is *synchronous*, in the sense that at every time the clock clicks, all neurons having enabled rules *must* fire one (and only one) of them. We also stress the fact that synapses are oriented: a synapse (i, j) allows to send spikes from neuron σ_i to neuron σ_j, but not the other way round. If we need to send spikes in both directions, then two distinct synapses must be considered: one from σ_i to σ_j, and another from σ_j to σ_i.

The applicability of a spiking rule $E/a^c \rightarrow a; d$ of neuron σ_i is determined by the regular expression associated with it, against which the number of spikes that are present in the neuron is checked. If the neuron at time t contains a number n of spikes such that $a^n \in L(E)$ and $n \geqslant c$, then the rule is *enabled* and can fire. The firing of this rule removes c spikes from σ_i (thus leaving $n - c$ spikes), and prepares one spike to be delivered to all the neurons σ_j connected to σ_i through synapses, that is $(i, j) \in syn$, with a *delay* d specified by the rule; spikes emitted by the output neuron are also sent to the environment. If $d = 0$, then the spike is immediately emitted, otherwise the process of sending the spike is delayed by d time units. If the rule is fired at step t and $d \geqslant 1$, then at steps $t, t + 1, t + 2, \ldots, t + d - 1$ the neuron is *idle* or *closed*, so that it is not able to receive new spikes (if a neuron has a synapse to a closed neuron and tries to send a spike along it, then that spike is lost), and cannot fire new rules. This time

interval is called in neurophysiology *refractory period*. At time $t + d$, the neuron spikes and becomes *open* again, so that it can receive spikes (which are added to its current content and can be used starting from the step $t + d + 1$) and select rules to be fired.

Analogously, a forgetting rule $a^s \to \lambda$, can be applied only if the neuron contains exactly s spikes, that are removed from the neuron; furthermore, the request that $a^s \notin L(E)$ for any expression E controlling a spiking rule in the same neuron implies that the forgetting rule can be enabled only if there are no enabled spiking rules in the neuron.

The rules to be executed at time t are chosen following two criteria:

1. *sequentiality at single neuron level*: if an open neuron contains two or more enabled rules (this is possible, since two firing rules, $R_1 : E_1/a^{c_1} \to a; d_1$ and $R_2 : E_2/a^{c_2} \to a; d_2$, can have $L(E_1) \cap L(E_2) \neq \varnothing$, and hence the content of the neuron can be described by both E_1 and E_2) it can apply only one of them, chosen in a non-deterministic way;
2. *maximal parallelism at system level*: all neurons evolve in parallel, and if an open neuron can use a rule at time t, then it *must* do it.

The *configuration* of the system at time t is denoted by $C_t = \langle k_1/t_1, \ldots, k_m/t_m \rangle$, where k_i, for $i = 1, 2, \ldots, m$, denotes the number of spikes contained in the neuron σ_i and $t_i \geqslant 0$ indicates the number of time steps to count down until the refractory period of the neuron ends; in particular, $t_i = 0$ means that σ_i is open. The execution of a set of rules, as defined above, changes the content of the neurons, removing or adding spikes, and their refractory periods, and hence the system reaches a new configuration C_{t+1} at time $t + 1$. As usual, we can define a *computation* as a sequence $C_0 \Rightarrow C_1 \Rightarrow \cdots \Rightarrow C_t \Rightarrow C_{t+1} \ldots$ of *transitions* from a configuration C_t to the next configuration C_{t+1}, starting from the initial configuration $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$, where, for every i, neuron σ_i contains n_i spikes, and is open. A computation is *halting* if it reaches a configuration C_t in which all the neurons are open and there are no enabled rules. A computation is considered *successful* if the output neuron spikes exactly twice during the computation; in such a case, the output of the system consists of the number of time steps elapsed between the two spikes of the output neuron. Similarly, we can specify the number n as input by sending two spikes from the environment to the input neuron i_{in}, in such a way that n time steps separate them.

SN P systems can be used to *generate* numbers (in which case no input is given, and a possibly different number is generated at each computation due to non-determinism), *accept* numbers given as input (the number is accepted if and only if the computation halts), or *compute a function* $f : \mathbb{N} \to \mathbb{N}$, using both the input and the ouput neuron. With a more conventional approach, we can also feed the system with a number n of spikes put in the input neuron at time t_0, and read the result (in unary form) as the number of spikes contained in the output neuron when the computation halts. In some papers, the result of a halting computation is instead defined as the total number of spikes sent into the environment by the output neuron during the computation.

As already proved in [1], the computational power of SN P systems is equivalent to that of Turing machines. The proof relies on the simulation of register machines, and has since then become a classic way to prove computational completeness of P systems variants.

A *register machine* is a tuple $M = (m, L, l_0, l_h, I)$, where m is the number of registers, L is the set of instruction labels (one label per instruction), l_0 is the label of the first instruction, l_h is the halt label, and I is the set of instructions. Registers contain non-negative integer values, and the instructions in I are of the forms:

- $l_1 : (\text{ADD}(r), l_2, l_3)$: increment register r by 1, then go to instruction l_1 or l_2 chosen in a non-deterministic way;
- $l_1 : (\text{SUB}(r), l_2; l_3)$: if the value of register r is positive then subtract 1 and go to instruction l_2, otherwise go to instruction l_3;
- l_h : HALT: ends the computation.

Computations start by executing the first instruction of M, and terminate when they reach the instruction l_h. The result of the computation is the number contained in register 1 when the computation

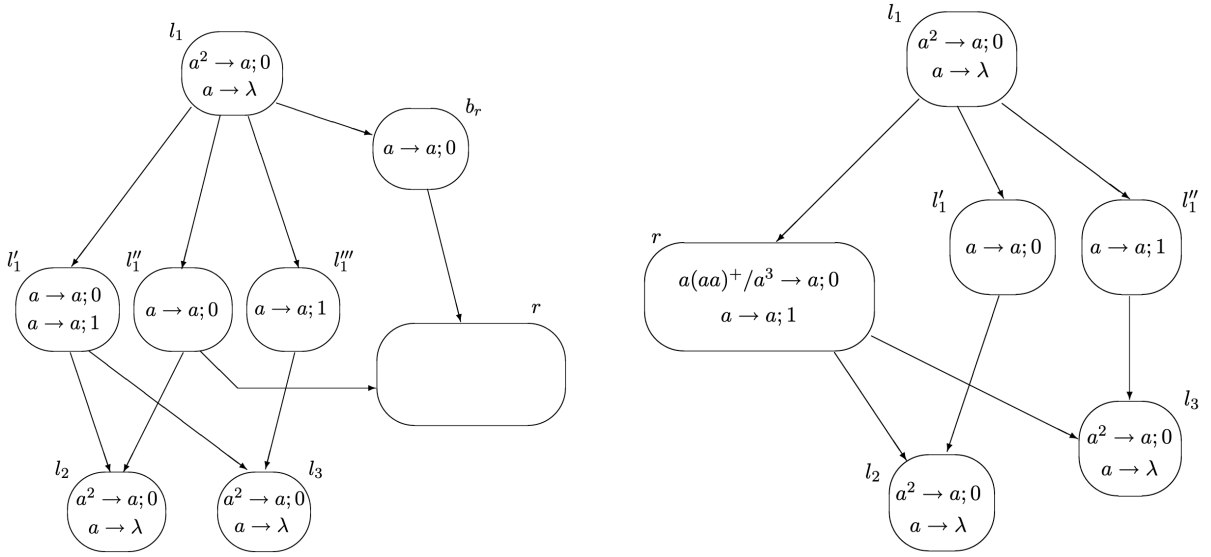


Figure 1: Left: module ADD, simulating $l_1 : (\text{ADD}(r), l_2, l_3)$. Right: module SUB, simulating $l_1 : (\text{SUB}(r), l_2, l_3)$.

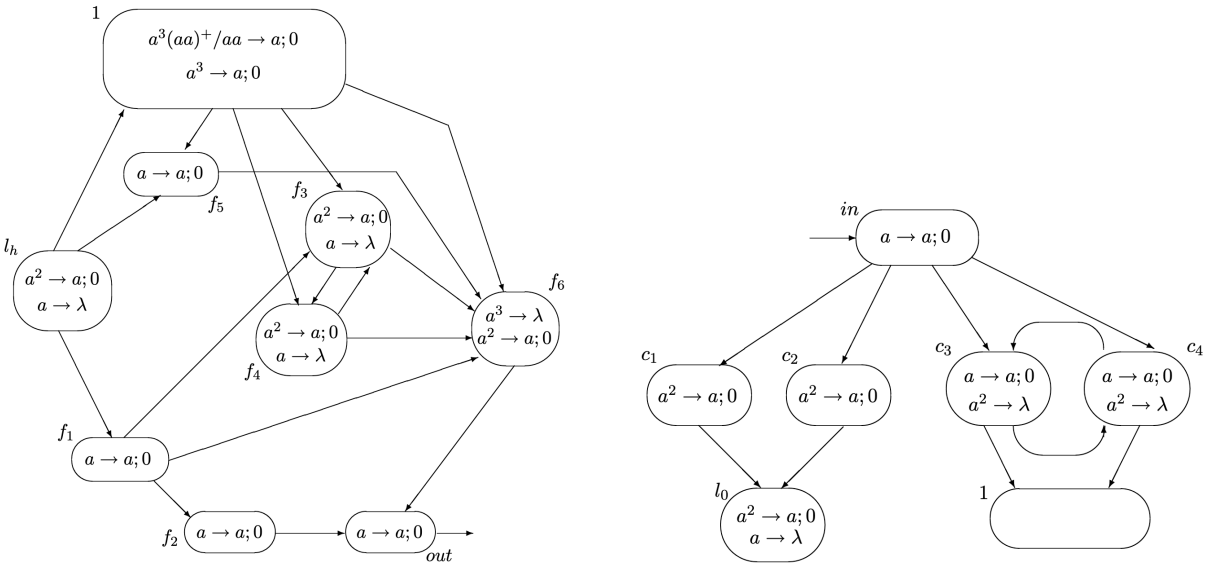


Figure 2: Left: module FIN, ending the computation of a generarive SN P system. Right: module INPUT, starting the computation of an accepting SN P system.

halts. Without loss of generality, this register is never decremented during the computation. Register machines provide a simple universal computational model, as proved in [33].

Figure 1 illustrates modules made of spiking neurons that simulate the register machine instructions $l_1 : (\text{ADD}(r), l_2, l_3)$ and $l_1 : (\text{SUB}(r), l_2, l_3)$, respectively. By appropriately composing these modules, SN P systems can simulate any register machine. Like SN P systems, also register machines can work in generative, accepting, or computing mode. In the generative case, the FIN module illustrated in the left part of Figure 2 can be used to send the result (the number of spikes contained in register 1) to the environment, encoded as the number of time steps elapsed between two emitted spikes. Similarly, in the accepting case, the input number is encoded as the number of time steps between the spikes sent from the environment to the input neuron; the INPUT module illustrated in the right part of Figure 2 converts this temporal sequence into the corresponding number of spikes. When computing a function, SN P systems employ both modules.

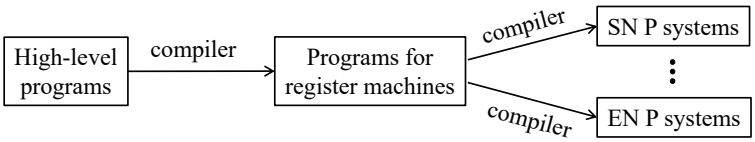


Figure 3: Compiling programs written in a high-level programming language to SN P systems (or other P system models), passing from register machines.

Some simplifications can be performed when simulating specific register machines, such as the small universal register machines proposed by Korec [34], producing as a result corresponding small universal SN P systems. As stated above, several extensions of SN P systems have been proposed in the literature; usually, their Turing equivalence is proved by simulating register machines.

3. Designing Spiking Neural P Systems: the WHILE compiler

Suppose we want to design a spiking neural P system that computes a certain function, say $f(n) = n^2$. It is certainly possible to build the system by working directly, adding and modifying neurons, but to do this requires a certain amount of experience, which only those who have been working with SN P systems for a long time have.

Another possibility is that we first write a program for a register machine that computes the desired function, and then build the SN P system by composing the ADD and SUB modules that simulate the register machine. This is certainly a simpler way to proceed, and it also has two desirable features: (1) the transformation from register machines to SN P systems can be done mechanically, and (2) the principle applies not only to basic SN P systems, but in general to all bio-inspired computational models for which Turing equivalence has been proved by simulating register machines.

However, working with register machines is also uncomfortable (although simpler than working directly with SN P systems), and so we ask: is it possible to write the definition of the function to be computed in a high-level programming language, which is then compiled into an equivalent program for a register machine, and subsequently into a SN P system? This process is illustrated in Figure 3, and it involves at least two compilers: (1) from programs written in the high-level programming language to equivalent programs running on register machines, and (2) from programs written for register machines to equivalent SN P systems (or another P systems-based computing model). Of course, both compilers would run programmatically, without human intervention. The output of the second compiler could be provided in a format that is suitable for running the computation via a simulator of SN P systems, for example, in the P-Lingua format [35].

Writing these compilers is an ongoing project, carried out with Lorenzo Rovida, a post-doc at the Polytechnic University of Turin. The WHILE language was chosen as the high-level programming language due to its simplicity, being an educational language adopted in a famous book on theoretical computer science by Martin D. Davies, Ron Sigal, and Elaine J. Weyuker [36]. The choice fell on this language to avoid the practical problems associated with writing a compiler for a sophisticated language. Indeed, the language is very simple. It allows to define variables x_j, for $j \in \mathbb{N}$, each containing a non-negative integer value, and there are only three kinds of commands:

- *assignment* commands, of the type:

$$x_k := 0 \qquad x_k := x_j + 1 \qquad x_k := x_j \dot{-} 1 \quad \text{(truncated decrement)}$$

- *while* commands:

$$\text{while } x_k \neq 0 \text{ do } C$$

where C is an arbitrary command;

⟨*program*⟩ → [⟨*include*⟩]
'`Program`' `[a-zA-Z][a-zA-Z0-9]*` '`;`'
[⟨*description*⟩]
[⟨*input*⟩]
⟨*statements-list*⟩
⟨*output*⟩

⟨*include*⟩ → '`include`' → `[a-zA-Z0-9_.-]*` '`;`' [⟨*include*⟩]

⟨*description*⟩ → '`/*`' `[a-zA-Z0-9_.-\s]*` '`*/`'

⟨*input*⟩ → '`input`' ⟨*registers*⟩

⟨*composed-statement*⟩ → '`begin`' ⟨*statements*⟩ '`end`'

⟨*output*⟩ → '`output`' '`x_`' ⟨*id*⟩

⟨*statements*⟩ → ⟨*statement*⟩ [⟨*statement*⟩]

⟨*statement*⟩ → ⟨*assignment*⟩
| ⟨*inc*⟩
| ⟨*dec*⟩
| ⟨*while*⟩
| ⟨*macro*⟩
| ⟨*comment*⟩

⟨*assignment*⟩ → '`x_`' ⟨*id*⟩ '`=`' `[0-9]*` '`;`'

⟨*inc*⟩ → '`x_`' ⟨*id*⟩ '`=`' '`x_`' ⟨*id*⟩ '`+`' '`1`' '`;`'

⟨*dec*⟩ → '`x_`' ⟨*id*⟩ '`=`' '`x_`' ⟨*id*⟩ '`-`' '`1`' '`;`'

⟨*while*⟩ → '`while`' '`x_`' ⟨*id*⟩ '`!=`' '`0`' '`do`' ⟨*composed-statement*⟩
| '`while`' '`x_`' ⟨*id*⟩ '`!=`' '`0`' '`do`' ⟨*statements*⟩ '`end while`'

⟨*comment*⟩ → '`//`' `.*`

⟨*registers*⟩ → '`x_`' ⟨*id*⟩
| '`x_`' ⟨*id*⟩ '`,`' ⟨*registers*⟩

⟨*id*⟩ → `[0-9]{1, 2}`

Figure 4: The grammar used to parse WHILE programs.

- *compound* commands:

$$\text{begin } C_1; C_2; \ldots; C_m; \text{ end} \qquad (m > 0)$$

  where $C_1; C_2; \ldots; C_m$ are arbitrary commands.

Finally, a *program* is a compound command.

Of course, the WHILE language is Turing-complete. The language can be extended through macros of the kind $x_i = Op(x_j, x_k)$, which make it easier to write programs; for example, Op can be Sum, Product, TruncatedSum, IntegerDivision, Mod, CantorPairingFunction, etc.

Figure 4 shows the grammar used to parse WHILE programs. Each command then has its translation into a sequence of instructions for a register machine. As an example, Figure 5 shows the translation of the assignment command $x_i := x_j + 1$, for $i \neq j$, and a while loop.

The proposed method allows to obtain an SN P system in a few minutes. Of course, the system will be far from optimized, but it can be used to quickly perform tests, even by those who are not particularly experienced in SN P system design. On the other hand, for those who are experienced the produced system will be a good starting point for hand-made optimizations.

4. Evolving Spiking Neural Circuits

Another possibility for designing SN P systems is obtaining them via evolutionary algorithms. While some attempts have been made in the literature applying neuro-evolution techniques on basic SN P

$$T_{W\to RM}(x_i := x_j + 1) = \begin{cases} l_0: & \texttt{SUB}(r_i), l_0, l_1 \\ l_1: & \texttt{ADD}(r_{22}), l_2, l_2 \\ l_2: & \texttt{ADD}(r_i), l_3, l_3 \\ l_3: & \texttt{SUB}(r_j), l_1, l_4 \\ l_4: & \texttt{SUB}(r_{22}), l_4, l_6 \\ l_5: & \texttt{ADD}(r_j), l_4, l_4 \\ l_5: & \texttt{SUB}(r_{22}), l_5, l_6 \\ l_6: & \ldots \end{cases} \qquad T_{W\to RM}(\texttt{while } x_i \neq 0 \texttt{ do } C) = \begin{cases} l_0: & \texttt{SUB}(r_i), l_1, l_{n+3} \\ l_1: & \texttt{ADD}(r_i), l_2, l_2 \\ l_2: & C_1 \\ l_3: & C_2 \\ \ldots & \\ l_{n+1}: & C_n \\ l_{n+2}: & \texttt{SUB}(r_{22}), l_0, l_0 \\ l_{n+3}: & \ldots \end{cases}$$

Figure 5: Left: compilation of the WHILE assignment command $x_i := x_j + 1$, with $i \neq j$. Right: compilation of a while command.

systems (see, for example, [37]), especially to solve computationally hard combinatorial problems, here will focus on the model of *SN P circuits* recently introduced in [38].

Differently from traditional SN P systems, SN P circuits are *acyclic*, that is, composed of layers, each one made of *SN P gates*. Similarly to what happens with traditional Boolean circuits made of AND/OR/NOT gates, a n-input/m-output SN P circuit computes a n-input/m-output Boolean function $f : \{0,1\}^n \to \{0,1\}^m$. In general, given a Boolean function, there may be several SN P circuits that compute it. This depends also on the fact that the Boolean functions we are interested in are only *partially defined*; that is, not all input/output $(\mathbf{x}, \mathbf{y})$ pairs will be specified. This makes SN P circuits suitable for solving some machine learning problems, as a partially defined Boolean function can be seen as the training/testing dataset for a classification problem. Hence, finding a SN P circuit that correctly computes the given input/output pairs of a Boolean function corresponds to finding a circuit that solves a given classification problem. In [38], we have investigated how difficult is to automatically find one of these circuits, using genetic algorithms (GA).

An *SN P circuit* is a circuit composed of SN P gates. A *spiking neural P gate* (*SN P gate*, for short) is a spiking neuron that contains a subset of the following spiking rules:

$$a^i \to a \qquad \text{for all } i \in \{0, 1, \ldots, \ell\}$$

for a fixed integer value $\ell \geqslant 1$. Note that a SN P gate is a *symmetric* gate[1], defined by the set $G \subseteq \{0, 1, \ldots, \ell\}$ such that the neuron contains a rule of the kind $a^i \to a$ if and only if $i \in G$. Clearly, the value of ℓ may differ among SN P gates. As recalled above, the SN P gates of a SN P circuit can be arranged in layers, such that each gate takes its input values only among the output values of the previous layer (from the input nodes, for the first layer). A layer is made up of SN P gates that are not connected to each other. That is, if two SN P gates belong to the same layer, then the output of one of the two gates is not used as input by the other gate. Evaluating a circuit thus amounts to computing the Boolean vector produced by each layer, starting from the first layer and proceeding up to the output layer. Precisely, suppose that an SN P circuit computes a function $f : \{0,1\}^n \to \{0,1\}^m$, and that it is made up of k layers. First, we prepare an *input vector* for the circuit, consisting of n binary values. The SN P gates of the first layer take their inputs from this input vector, according to their own input lines. The number of spikes arriving at an SN P gate through its input lines are summed, and possibly activate a firing rule or an (implicitly defined) forgetting rule. Consequently the gate fires or just cancels its spikes, thus producing one or zero spikes as output, corresponding to the Boolean value 1 or 0, respectively. All the Boolean values produced in this way by the gates of the first layer form a vector of binary values, which constitute the input values for the SN P gates of the second layer. All the SN P gate layers of the circuit are evaluated in this way, one after the other, until the k-th layer is evaluated, which produces the circuit's vector of output Boolean values.

[1]That is, the output value only depends upon the number of 1s given as input. An n-input symmetric gate can be defined by specifying the subset $G \subseteq \{0, 1, \ldots, n\}$ such that the output value is 1 if and only if the number of 1s given as input is in G.

Algorithm 1 Genetic Algorithm for finding a SN P circuit that computes a given Boolean function f

procedure EVOLVE(f)
  ▷ f is (possibly partially) defined by a list of pairs $(\mathbf{x}, \mathbf{y}) \in \{0,1\}^n \times \{0,1\}^m$
  $P \leftarrow$ initial population of randomly generated SN P circuits
  FIT $\leftarrow$ fitness values of each circuit in P
  save the maximum and average fitness values in a list
  while stopping criterion is not verified **do**
    ▷ P' and FIT$'$ constitute the new generation
    $P' \leftarrow$ best p percent circuits from P ▷ p typically is in $[0, 10]$
    FIT$' \leftarrow$ fitness values of circuits in P'
    for $i \leftarrow 1$ to $(1 - p/100) \cdot |P|$ **do**
      parent1, parent2 $\leftarrow$ FITNESSPROPORTIONATESELECTION(P)
      child $\leftarrow$ CROSSOVER(parent1, parent2) ▷ only the fittest among the two children is kept
      add MUTATION(child) to P'
      add the fitness of child to FIT$'$
    end for
    save the maximum and average fitness values
    $P \leftarrow P'$ ▷ substitute P with P'
  end while
end procedure

A Boolean function $f : \{0,1\}^n \to \{0,1\}^m$ is said to be *partially defined* if we only know some pairs $(\mathbf{x}, \mathbf{y})$ such that $\mathbf{x} \in \{0,1\}^n$, $\mathbf{y} \in \{0,1\}^m$, and $f(\mathbf{x}) = \mathbf{y}$. For the input values $\mathbf{x} \in \{0,1\}^n$ not comprised in this list of pairs, the function may assume any output value. As said above, our aim is to apply genetic algorithms (GA) to find SN P circuits that compute a given (partially defined) Boolean function. Notice that the fewer the pairs that partially define the Boolean function to be calculated, the more Boolean functions will be consistent with it, and the easier it will therefore be to find an SN P circuit that calculates it. Conversely, the more pairs there are that partially define the Boolean function to be calculated, the more difficult it will be to find an SN P circuit that calculates it.

Algosithm 1 illustrates the pseudocode of the GA. The algorithm maintains a population of SN P circuits, which at the beginning of the execution is initialized with randomly generated individuals. The *fitness* of each circuit c is defined as:

$$\text{fitness}(c, f) = 1 - \frac{1}{k}\left(\sum_{i=1}^{k} \big(c(\mathbf{x}_i) \neq \mathbf{y}_i\big)\right) \tag{1}$$

where $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_k, \mathbf{y}_k)$ are the pairs that (partially) define the Boolean function $f : \{0,1\}^n \to \{0,1\}^m$, and $c(\mathbf{x}_i)$ is the output produced by circuit c on input $\mathbf{x}_i$. The sum counts the number of times the output of circuit c differs from the expected output $\mathbf{y}_i$. Such a number is averaged over the number k of input values, and the result is subtracted from 1, so that the fitness value is always between 0 and 1, with 1 indicating that the circuit always gives the correct answer, and 0 indicating that the circuit gives the wrong answer every time.

To build the population for the next iteration of the algorithm, we first apply *elitism*: the next generation will contain a percentage p (10%, in our implementation) of the highest-fitness individuals of the current generation. The rest of the new generation is populated by new individuals, obtained through crossover and mutation operators. To apply the *crossover* operator, first two circuits are chosen from the current population, according to *fitness proportionate selection*. That is, each individual has a probability of being selected equal to the value of its fitness divided by the sum of the fitness values of all individuals in the current population. Having chosen the two parent circuits, inspired by *one-point crossover*, we cut each circuit before a randomly chosen layer; we then combine the first piece of the first

circuit with the second piece of the second circuit, and the second piece of the first circuit with the first piece of the second circuit. So doing, the two parents produce two circuits; however, at the end of the process we will keep only the one that has the highest fitness value, discarding the other. Notice that if the layers where the two parent circuits were cut do not have the same number of gates, there might be an invalid input connection to some neurons in one of the layers of the new circuits. In such a case we apply a CLEAN procedure, that iteratively checks whether all neurons have connections to existing neurons; if a pending connection is found, it is removed. If, after executing this procedure, the neuron has no edges left, it is removed. Eventually, if a layer remains without neurons, it will be removed, too.

After the crossover operation, and before computing the fitness, the newly produced individuals undergo a *mutation* phase, implemented as the following types of random changes:

- Adding or removing a rule to the set of rules in a randomly chosen neuron;
- Adding/removing a new gate in a randomly chosen layer;
- Adding/removing a new input line to a randomly chosen gate;

plus two more "drastic" changes:

- Adding/removing a new random layer of gates.

Notice that, after the application of some mutations (e.g., removing an existing gate), the CLEAN procedure must be executed in order to fix invalid edges, gates or layers. Each mutation is applied according to a corresponding value of probability. It is well known that applying mutations with high probabilities may lead to random search behavior, while applying a small number of mutations may cause premature convergence to sub-optimal solutions. Fine-tuning the mutation rate and the type of mutation operators is often an important part of optimizing a genetic algorithm for a specific problem. For this reason, during the experiments we have adopted a grid-search approach to find optimal probability values for each mutation. The same technique was applied to the other parameters that rule the behavior of the GA, such as population size, the number of layers and gates in randomly generated circuits, and crossover probability.

In [38], we performed some preliminary experiments, trying to learn (or fit) the completely defined XOR functions of $n = 5$ and $n = 8$ inputs, and two 5-input Boolean functions having a high non-linearity degree, expressed in Algebraic Normal Form. For each experiment, we tested several combinations of population size, minimum and maximum allowed number of layers, number of iterations to be used as the stopping criterion, crossover probability, and mutation probabilities. Encouraging results have been obtained, for which we refer the reader to [38]. Of course, many more experiments and investigation are needed to fully understand the potentiality and utility of this approach.

5. Related Models and Machine Learning

As we saw in the previous section, spiking neural P systems can also be used in machine learning tasks. Of course, the one described above is not the only possible approach. In particular, several learning models for SN P systems have been proposed in the literature. Here we just mention a few of them, just to whet the reader's appetite and encourage him/her to delve deeper into literature.

For example, in [39] a variant called *SN P systems with learning functions* is defined, which can dynamically strengthen and weaken connections among neurons during the computation. Starting from this, a class of SN P systems with a simple Hebbian learning function is constructed to recognize English letters. In [40], the authors introduce *Convolutive Kernel-Guarded* Spiking Neural P systems (CK-SNP systems), as a trainable framework in which the applicability of spiking rules may depend on local kernel responses computed over ordered neighborhoods of spike multiplicities. In particular, they connect the discrete symbolic formal model to machine learning by introducing differentiable surrogate gates (sigmoid approximations) during training, which transition to hard-threshold rules during inference. This is a very important topic, since SN P systems are discrete models, and the equations describing their dynamics are not differentiable. This same problem is encountered in the more general model of *Spiking*

Neural Networks (SNN), for which some gradient descent and backpropagation-based algorithms have already been proposed, to allow perfoming supervised learning tasks. These algorithms usually rely on techniques such as *Surrogate Gradient Descent*, that approximate activation functions, thus allowing to calculate the gradient and enabling the use of the backpropagation algorithm. After the network has been trained, if the number of inputs is not large then an interesting option is to transform the network into a Boolean circuit, hence going back to a discrete model, which allows to save computational power and also energy when performing inferences. Similar ideas are also pursued in models inspired from *Binarized Neural Networks* [41], where only binary inputs, outputs, and weights are used.

We conclude this section by reporting the existence of models of continuous SN P systems [42], in which each neuron is associated with a real numerical value rather than a discrete number of spikes, and in which the number of spikes consumed and produced by the firing rules are replaced by non-linear functions. These models have been used to build different architectures of neural networks, and have been applied to several problems of classification, regression, image segmentation, etc.

6. Conclusions

In this survey we reviewed spiking neural P systems, a very interesting neural-like model of computation. After seeing how the basic model is defined, we recalled some of its computational properties, in particular its equivalence to Turing machines.

We then explored two very different ways of designing SN P systems that compute a desired function. In the first case we write this function in a high-level programming language and then, using two compilers, we transform the program into an SN P system, passing from register machines. In the second case we use evolutionary algorithms (in particular, the genetic algorithm) to evolve a population of SN P circuits in search of a circuit that calculates a partially specified Boolean function.

Both of these approaches already suggest several possible lines of future research. For example, instead of using the educational WHILE language we could use a programming language more similar to those used in the real world. To exploit the inherent parallelism of SN P systems, it would also be interesting to start from a parallel programming language. The extension of the second compiler to different models of SN P systems is another possibility, as is producing a description of the model that can be simulated using one of the simulators present in the literature.

Concerning the evolutionary approach, many possibilities arise. In [38] only few initial experiments have been performed, many more are needed. For example, evolving many-output Boolean functions, functions with more inputs, and functions having cryptographic properties such as correlation immunity, non-linearity, and so on. Last but not the least, GA is not the only possibility: many more evolutionary algorithms exist, each with its own behavior and hyperparameters that drive the dynamics. A lot of fun is waiting for us!

Declaration on Generative AI

The author has not employed any Generative AI tools.

References

[1] M. Ionescu, Gh. Păun, T. Yokomori, Spiking Neural P Systems, Fundamenta Informaticae 71 (2006) 279–308.

[2] Gh. Păun, Introduction to Membrane Computing, Springer Berlin Heidelberg, 2006, pp. 1–42. doi:10.1007/3-540-29937-8_1.

[3] Gh. Păun, G. Rozenberg, A. Salomaa, The Oxford Handbook of Membrane Computing, Oxford University Press, Inc., 2010.

[4] W. Maass, Networks of spiking neurons: The third generation of neural network models, Neural Networks 10 (1997) 1659–1671. doi:10.1016/S0893-6080(97)00011-7.

[5] Gh. Păun, Spiking Neural P Systems with Astrocyte-Like Control, JUCS - Journal of Universal Computer Science 13 (2007) 1707–1721. doi:10.3217/jucs-013-11-1707.
[6] L. Pan, Gh. Păun, Spiking Neural P Systems with Anti-Spikes, International Journal of Computers Communications & Control 4 (2009) 273–282.
[7] H. Peng, J. Yang, J. Wang, T. Wang, Z. Sun, X. Song, X. Luo, X. Huang, Spiking neural P systems with multiple channels, Neural Networks 95 (2017) 66–71. doi:10.1016/j.neunet.2017.08.003.
[8] H. Peng, B. Li, J. Wang, X. Song, T. Wang, L. Valencia-Cabrera, I. Pérez-Hurtado, A. Riscos-Núñez, M. J. Pérez-Jiménez, Spiking neural P systems with inhibitory rules, Knowledge-Based Systems 188 (2020) 105064. doi:https://doi.org/10.1016/j.knosys.2019.105064.
[9] H. Peng, T. Bao, X. Luo, J. Wang, X. Song, A. Riscos-Núñez, M. J. Pérez-Jiménez, Dendrite P systems, Neural Networks 127 (2020) 110–120. doi:https://doi.org/10.1016/j.neunet.2020.04.014.
[10] T. Wu, A. Păun, Z. Zhang, L. Pan, Spiking Neural P Systems With Polarizations, IEEE Transactions on Neural Networks and Learning Systems 29 (2018) 3349–3360. doi:10.1109/TNNLS.2017.2726119.
[11] F. G. C. Cabarle, H. N. Adorna, M. J. Pérez-Jiménez, T. Song, Spiking neural P systems with structural plasticity, Neural Computing and Applications 26 (2015) 1905–1917. doi:10.1007/s00521-015-1857-4.
[12] T. Wu, F. Neri, L. Pan, On the tuning of the computation capability of spiking neural membrane systems with communication on request, International Journal of Neural Systems 32 (2022) 2250037.
[13] X. Song, L. Valencia-Cabrera, H. Peng, J. Wang, Spiking neural P systems with autapses, Information Sciences 570 (2021) 383–402. doi:https://doi.org/10.1016/j.ins.2021.04.051.
[14] Y. Li, B. Song, X. Zeng, Spiking neural P systems with weights and delays on synapses, Theoretical Computer Science 968 (2023) 114028. doi:https://doi.org/10.1016/j.tcs.2023.114028.
[15] F. G. C. Cabarle, X. Zeng, N. Murphy, T. Song, A. Rodríguez-Patón, X. Liu, Neural-like P systems with plasmids, Information and Computation 281 (2021) 104766. doi:https://doi.org/10.1016/j.ic.2021.104766.
[16] H. Peng, J. Wang, M. J. Pérez-Jiménez, A. Riscos-Núñez, Dynamic threshold neural P systems, Knowledge-Based Systems 163 (2019) 875–884. URL: https://www.sciencedirect.com/science/article/pii/S0950705118305021. doi:https://doi.org/10.1016/j.knosys.2018.10.016.
[17] H. Peng, J. Wang, Coupled Neural P Systems, IEEE Transactions on Neural Networks and Learning Systems (2018) 1–11. doi:10.1109/TNNLS.2018.2872999.
[18] B. Li, H. Peng, J. Wang, X. Huang, Multi-focus image fusion based on dynamic threshold neural P systems and surfacelet transform, Knowledge-Based Systems 196 (2020) 105794. doi:https://doi.org/10.1016/j.knosys.2020.105794.
[19] L. Bo, H. Peng, X. Luo, J. Wang, S. Xiaoxiao, M. Pérez-Jiménez, A. Riscos-Núñez, Medical Image Fusion Method Based on Coupled Neural P Systems in Nonsubsampled Shearlet Transform Domain, International Journal of Neural Systems 31 (2020) 2050050. doi:10.1142/S0129065720500501.
[20] Y. Cai, S. Mi, J. Yan, H. Peng, X. Luo, Q. Yang, J. Wang, An unsupervised segmentation method based on dynamic threshold neural P systems for color images, Information Sciences 587 (2022) 473–484. doi:https://doi.org/10.1016/j.ins.2021.12.058.
[21] J. Yan, L. Zhang, X. Luo, H. Peng, J. Wang, A novel edge detection method based on dynamic threshold neural P systems with orientation, Digital Signal Processing 127 (2022) 103526. doi:https://doi.org/10.1016/j.dsp.2022.103526.
[22] R. Xian, R. Lugu, H. Peng, Q. Yang, X. Luo, J. Wang, Edge Detection Method Based on Nonlinear Spiking Neural Systems, International Journal of Neural Systems 33 (2023) 2250060. doi:10.1142/S0129065722500605.
[23] R. Xian, X. Xiong, H. Peng, J. Wang, A. R. de Arellano Marrero, Q. Yang, Feature fusion method based on spiking neural convolutional network for edge detection, Pattern Recognition 147 (2024) 110112. doi:https://doi.org/10.1016/j.patcog.2023.110112.
[24] B. Yang, L. Qin, H. Peng, C. Guo, X. Luo, J. Wang, SDDC-Net: A U-shaped deep spiking neural P

convolutional network for retinal vessel segmentation, Digital Signal Processing 136 (2023) 104002. doi:https://doi.org/10.1016/j.dsp.2023.104002.

[25] Y. Zhang, Q. Yang, Z. Liu, H. Peng, J. Wang, A Prediction Model Based on Gated Nonlinear Spiking Neural Systems, International journal of neural systems 33 (2023) 2350029. doi:10.1142/S0129065723500296.

[26] Y. Huang, H. Peng, Q. Liu, Q. Yang, J. Wang, D. Orellana-Martín, M. J. Pérez-Jiménez, Attention-enabled gated spiking neural P model for aspect-level sentiment classification, Neural Networks 157 (2023) 437–443. doi:https://doi.org/10.1016/j.neunet.2022.11.006.

[27] Y. Huang, Q. Liu, H. Peng, J. Wang, Q. Yang, D. Orellana-Martín, Sentiment classification using bidirectional LSTM-SNP model and attention mechanism, Expert Systems with Applications 221 (2023) 119730. doi:https://doi.org/10.1016/j.eswa.2023.119730.

[28] Q. Liu, Y. Huang, Q. Yang, H. Peng, J. Wang, An Attention-Aware Long Short-Term Memory-Like Spiking Neural Model for Sentiment Analysis, International Journal of Neural Systems 33 (2023) 2350037. doi:10.1142/S0129065723500375.

[29] X. Bai, Y. Huang, H. Peng, J. Wang, Q. Yang, D. Orellana-Martín, A. R. de Arellano, M. J. Pérez-Jiménez, Sequence recommendation using multi-level self-attention network with gated spiking neural P systems, Information Sciences 656 (2024) 119916. doi:https://doi.org/10.1016/j.ins.2023.119916.

[30] G. Zhang, H. Rong, F. Neri, M. J. Pérez-Jiménez, An optimization spiking neural P system for approximately solving combinatorial optimization problems, International Journal of Neural Systems 24 (2014) 1440006. doi:10.1142/S0129065714400061.

[31] M. Zhu, Q. Yang, J. Dong, G. Zhang, X. Gou, H. Rong, P. Paul, F. Neri, An Adaptive Optimization Spiking Neural P System for Binary Problems, International Journal of Neural Systems 31 (2020). doi:10.1142/S0129065720500549.

[32] G. Zhang, X. Zhang, H. Rong, P. Paul, M. Zhu, F. Neri, Y.-S. Ong, A Layered Spiking Neural System for Classification Problems, International Journal of Neural Systems 32 (2022) 2250023. doi:10.1142/S012906572250023X.

[33] M. L. Minsky, Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[34] I. Korec, Small universal register machines, Theoretical Computer Science 168 (1996) 267–301. doi:10.1016/S0304-3975(96)00080-1.

[35] D. Díaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, A. Riscos-Núñez, P-lingua: A programming language for membrane computing, in: Proceedings of the 6th Brainstorming Week on Membrane Computing, Fénix Editora, 2008, pp. 135–155.

[36] M. Davis, R. Sigal, E. J. Weyuker, Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science, 2nd ed., Morgan Kaufman, 1994.

[37] L. L. Custode, H. Mo, A. Ferigo, G. Iacca, Evolutionary optimization of spiking neural P systems for remaining useful life prediction, Algorithms 15 (2022). doi:10.3390/a15030098.

[38] A. Leporati, L. Rovida, An evolutionary approach to the design of spiking neural P circuits, Journal of Membrane Computing 7 (2025) 221–235. doi:10.1007/s41965-024-00180-x.

[39] T. Song, L. Pan, T. Wu, P. Zheng, M. L. D. Wong, A. Rodríguez-Patón, Spiking neural P systems with learning functions, IEEE Transactions on NanoBioscience 18 (2019) 176–190. doi:10.1109/TNB.2019.2896981.

[40] D. Constantin, C. Bălcău, Convolutive kernel-guarded spiking neural P systems for local feature computation, Big Data and Cognitive Computing 10 (2026). doi:10.3390/bdcc10070218.

[41] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. Bengio, Binarized neural networks, in: Proceedings of the 30th International Conference on Neural Information Processing Systems, NIPS'16, Curran Associates Inc., Red Hook, NY, USA, 2016, p. 4114–4122.

[42] H. Peng, Z. Lv, B. Li, X. Luo, J. Wang, X. Song, T. Wang, M. J. Pérez-Jiménez, A. Riscos-Núñez, Nonlinear spiking neural P systems, International Journal of Neural Systems 30 (2020) 2050008. doi:10.1142/S0129065720500082.