Parallel Programming in Spiking Neural P Systems with Anti-Spikes

Venkata Padmavati METTA\textsuperscript{1}, Kamala KRITHIVASAN\textsuperscript{2}

\textsuperscript{1} Institute of Computer Science and Research Institute of the IT4Innovations Centre of Excellence, Silesian University in Opava, Czech Republic
\textsuperscript{2}Indian Institute of Technology, Chennai, India
E-mail: vmetta@gmail.com, kamala@iitm.ac.in

Abstract. In spiking neural P systems with anti-spikes, integers can be represented as the number of spikes/anti-spikes present in a neuron. Thus it is possible to represent increment, decrement, relational operations and assignment statements using spiking neural P systems. With these basic structures, it is also possible to represent programming language constructs like conditional, iterative and different types of execution constructs. In this paper we consider a general-purpose parallel programming language that handles integer variables and is implemented using spiking neural P systems with anti-spikes.

Key-words: spiking neural P systems with anti-spikes, Occam, simulation.

1. Introduction

Spiking neural P systems (in short, SN P systems) \cite{4} are computational models inspired by the spiking activity of neurons in the brain. An SN P system is represented as a directed graph where nodes correspond to the neurons having spiking and forgetting rules. The rules involve the spikes present in the neuron in the form of occurrences of a symbol $a$. SN P systems operate in a locally sequential and globally maximal manner using a global clock. That is, in each neuron, at each step, if more than one rule is enabled, then only one of them is applied non-deterministically. All neurons fire in parallel at the system level. An SN P system is used as a computing device in various ways – acceptor, transducer and generator.

Spiking neural P systems with anti-spikes (in short, SN PA system) \cite{7} work in the same way as standard SN P systems but deal with two types of objects called spikes...
(a) and anti-spikes ($\pi$). There is also a highest priority annihilation rule ($a\pi \to \lambda$) that is implicitly present in each neuron of an SN PA system. Because of the use of two types of objects, the system can encode binary digits in a natural way and hence can represent the formal models more efficiently and naturally than the SN P systems. The power of SN PA systems as language generators is studied in [5]. In [6], the SN PA systems in transducer mode are used to simulate Boolean circuits. Here we demonstrate that SN PA systems are not only efficient in implementing hardware components but also software components like programming language constructs.

Both spiking neural P systems and artificial spiking neural networks are computational devices inspired by the concept of spiking neurons. There are several relations between these systems. For example, an SN P model for Hebbian learning using concepts borrowed from neuroscience and artificial neural network theory is presented in [3]. The present paper borrows some concepts present in [1], where spiking neural networks are used to represent variables and different constructs of a parallel programming language. In this paper we try to simulate the programming language constructs like if, while, seq, par and alt using SN PA systems.

This paper is organised as follows. We start with Section 2 by giving a brief introduction to the SN P systems with anti-spikes. In Section 3, we describe a very weak version of Occam and the language constructs considered for implementation. Occam is a parallel programming language used in the development of the VLSI chip called transputer that can execute concurrent processes [2]. The representation of integer variables, increment, decrement operations and assignment statements are considered in Section 4. In Section 5, we implement all considered programming language constructs.

2. Spiking Neural P System with Anti-Spikes

We briefly introduce the SN PA systems used in this paper.

**Definition 2.1** A spiking neural P system with anti-spikes, of degree $m \geq 1$, is a construct

$$\Pi=(O, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m, \text{syn}, \text{IN}, \text{OUT})$$

where

1. $O = \{a, \pi\}$ is a binary alphabet. $a$ is called spike and $\pi$ is called an anti-spike.
2. $\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_m$ are neurons, of the form

$$\sigma_i=(n_i, R_i), 1 \leq i \leq m,$$

(a) $n_i$ is the number of spikes or anti-spikes contained in the neuron $\sigma_i$ and if $n_i > 0$ then the neuron is having $n_i$ spikes and if $n_i < 0$ then the neuron is having $n_i$ anti-spikes;
(b) $R_i$ is a finite set of rules of the form $E/b^r \to d^p$ where $b, d \in \{a, \pi\}, r \geq 1, p \geq 0$, with the restriction that $r \geq p$ and $E$ is either a regular expression over $a$ or a regular expression over $\pi$.
Like in [7], we avoid using rules $\pi^r \rightarrow \pi^p$, but not the other three types, corresponding to the pairs $(a,a)$, $(a,\overline{a})$, $(\overline{a},a)$.

3. $\text{syn} \subseteq \{1, 2, 3, \ldots, m\} \times \{1, 2, 3, \ldots, m\}$ with $(i, i) \notin \text{syn}$ for $1 \leq i \leq m$ (synapses among cells);

4. $\text{IN, OUT} \subseteq \{1, 2, 3, \ldots, m\}$ are the set of input and output neurons respectively.

A rule $E/b^r \rightarrow d^p$ is applied as follows. If the neuron $\sigma_i$ contains $k$ spikes when $b = a$ (or $k$ anti-spikes when $b = \overline{a}$), and $b^k \in L(E)$, $k \geq r$, then the rule can fire, and upon application, $r$ spikes (or anti-spikes) are consumed (thus only $k-r$ remain in $\sigma_i$) and $p$ spikes (or anti-spikes) are released when $d = a$ (or $d = \overline{a}$), which will immediately exit the neuron. The spikes (or anti-spikes) emitted by the neuron $\sigma_i$ will pass immediately to all neurons $\sigma_j$ such that $(i, j) \in \text{syn}$. That means transmission of spikes/anti-spikes takes no waiting time (since the rules do not specify a time delay), the spikes/anti-spikes will be available in neuron $\sigma_j$ in the next step. There is an additional restriction that $a$ and $\pi$ cannot stay together, they annihilate each other. If a neuron has either objects $a$ or objects $\overline{a}$, and further objects of either type (maybe both) arrive from other neurons, such that we end with $a^s$ and $\overline{a}^q$ inside, then immediately an annihilation rule $a\pi \rightarrow \lambda$ (which is implicit in each neuron), is applied in a maximal manner, so that either $a^{s-r}$ or $(\overline{a})^{q-s}$ remain for the next step, provided that $q \geq s$ or $s \geq q$, respectively. This mutual annihilation of spikes and anti-spikes takes no waiting time and the annihilation rule has priority over spiking and forgetting rules, so each neuron always contains either only spikes or anti-spikes. If we have a rule $E/b^r \rightarrow d^p$ with $L(E) = \{b^r\}$, then we write it in the simplified form as $b^r \rightarrow d^p$. If $p = 0$ in $E/b^r \rightarrow d^p$, then we write as $E/b^r \rightarrow \lambda$, where $\lambda$ represents the empty string. These rules are similar to the forgetting rules of the standard SN P system, but here the forgetting rules also have regular expressions associated with them.

The configuration of the system is described by $C = (\beta_1, \beta_2, \ldots, \beta_m)$, where $\beta_i$ is the number of spikes/anti-spikes present in neuron $\sigma_i$. The initial configuration is $C_0 = (n_1, n_2, \ldots, n_m)$.

A global clock is assumed and in each time unit, each neuron which can use a rule should do it (the system is synchronized), but the work of the system is sequential locally: only (at most) one rule is used in each neuron. For example, if a neuron $\sigma_i$ has two firing rules, $E_1/b_1^{r_1} \rightarrow d_1^{p_1}$ and $E_2/b_2^{r_2} \rightarrow d_2^{p_2}$ with $L(E_1) \cap L(E_2) \neq \emptyset$, then it is possible that each of the two rules can be applied, and in that case only one of them is chosen non-deterministically. Thus, the rules are used in the sequential manner in each neuron, but neurons function in parallel with each other. In each step, all neurons which can use a rule of any type, spiking or forgetting, have to evolve, using a rule.

Using the rules in this way, we pass from one configuration of the system to another configuration; such a step is called a transition. For two configurations $C$ and $C'$ of $\Pi$ we denote by $C \Rightarrow C'$, if there is a direct transition from $C$ to $C'$ in $\Pi$.

A computation of $\Pi$ is a finite or infinite sequence of transitions starting from the initial configuration, and every configuration appearing in such a sequence is called
reachable. A computation halts if it reaches a configuration where no rule can be used. An SN PA system can be used as a computing device in various ways. In the generative mode, one of the neuron is considered as output neuron and it sends output to the environment. The moments of time when a spike is emitted by the output neuron are marked with 1, the moments of time when an anti-spike emitted is marked with 0 and no output moments are just ignored. This binary sequence is called the spike train of the system— it might be infinite if the computation does not stop. With halting configurations, we associate the language generated by the system as the set of binary sequence describing the spike trains.

If we consider both input and output neurons, then the SN PA system works as a transducer. In this paper we consider SN PA systems working in transducer mode.

3. A Simple Parallel Programming Language

The language described below is a very weak version of Occam handling only integer variables, and all variables are global. The language considered is similar to the one in [1].

We will use strings of characters starting with upper case letters as variables for integers.

The language consists of the following statements.

1. If $X$ is a variable, then $X := 0$, $X := X + 1$, and $X := X - 1$ are statements.
   
   We will say that two statements are independent if no variable which may be used in one statement is referred to in the other.

2. If $X$ and $Y$ are variables, then $Y := X$ and $Y := -X$ are statements. They are called assignment statements.

3. If $X$ is a variable, then $X = 0$, $X \neq 0$ and $X > 0$ are the tests.

4. If $T$ is a test, and $P$ and $Q$ are statements, then $if T then P else Q$ is a statement.

5. If $T$ is a test and $P$ is a statement, then $while (T) P$ is a statement.

6. If $P_1, P_2, \ldots, P_n$ are statements, then $seq\{P_1; P_2; \ldots; P_n;\}$ is a statement.

7. If $P_1, P_2, \ldots, P_n$ are independent statements, then $par\{P_1; P_2; \ldots; P_n;\}$, and $alt\{P_1; P_2; \ldots; P_n;\}$ are statements.

Statements in this language have a natural denotational semantics, defined as follows. We define a valuation to be an assignment of integral values to some subset of the variables. Then each statement denotes a possibly non-deterministic, possibly non-terminating, transformation from one valuation to another.

These denotations can be defined by structural recursion on the statements, as usual. So, for example, the transformation denoted by $par\{P_1; P_2; \ldots; P_n;\}$ is obtained by running the transformations denoted by $P_1, P_2, \ldots, P_n$ in parallel, and it only terminates when all of them terminate. On the other hand, the transformation denoted
Parallel Programming in Spiking Neural P Systems with Anti-Spikes

by \( \text{alt}\{ P_1; P_2; \ldots; P_n; \} \) is obtained by running all of the denoted transformations in parallel, and it terminates just when one of its constituents terminates, the choice being non-deterministic if several of them terminate.

We show in the next section how the transformations denoted by the statements in this programming language can be implemented in spiking neural P systems with anti-spikes.

4. Representation of Variables and Implementation of Various Operators

Each variable \( X \) is represented using neuron \( \sigma_X \). We use a representation where variable \( X \) has a non-negative value \( n \) if the corresponding neuron \( \sigma_X \) has \( 2n \) spikes, \( X \) has a value \( -n \) if it has \( 2n \) anti-spikes and \( X = 0 \) if \( \sigma_X \) has no spikes/anti-spikes. The four spiking rules in \( \sigma_X \), \( a(aa)^+ / a^2 \rightarrow \pi^2, \pi(\pi\pi)^+ / \pi^2 \rightarrow a^2, a \rightarrow \pi \) and \( \pi \rightarrow a \) are used to read the value of \( X \). These rules are fired when there is an odd number of spikes/anti-spikes in \( \sigma_X \).

The neuron \( \sigma_X \) is fired when a spike or an anti-spike is sent in. If the variable \( X \) has a positive value say \( n \), then the number of spikes present in \( \sigma_X \) will be \( 2n \). When \( \sigma_X \) is activated by sending a spike, the number of spikes in \( \sigma_X \) becomes odd. So the rule \( a(aa)^+ / a^2 \rightarrow \pi^2 \) is used until it is left with a spike. Then it uses the rule \( a \rightarrow \pi \) to clear its contents. In a similar way, rules \( \pi(\pi\pi)^+ / \pi^2 \rightarrow a^2 \) and \( \pi \rightarrow a \) are used when \( X \) has a negative value.

The SN PA system for each statement or construct will have a group of spiking neurons. To implement the constructs in a uniform way, the SN PA system is triggered by a spike arriving to an input neuron \( \sigma_{in} \) and upon completion it activates some termination neuron \( \sigma_{out} \), which passes the control to the next statement in the program.

Assigning a zero value to the variable \( X \) means clearing the contents of the neuron \( \sigma_X \). The implementation is presented in Fig. 1(a). It is done through sending a spike to the neuron \( \sigma_X \). The number of spikes/anti-spikes in \( \sigma_X \) becomes odd, so the rules are applied until all spikes/anti-spikes in \( \sigma_X \) are cleared. The output neuron sends a spike when \( \sigma_X \) becomes empty. A constant value can be assigned to the variable \( X \) by first clearing its contents and bringing a number of spikes/anti-spikes equal to twice the constant value.

4.1. Implementation of Increment and Decrement Statements

Incrementing the variable \( X \) will mean adding two spikes to neuron \( \sigma_X \) and decrementing the variable \( X \) will mean adding two anti-spikes to neuron \( \sigma_X \). The implementations of increment and decrement operations are shown in Fig. 1(b) and Fig. 1(c) respectively. In each case after completion, the output neuron sends a spike to the next system.
4.2. Implementation of Testing Statements

The test $X = 0$ is implemented with the SN PA system $\Pi_T$ in Fig. 2(a), with one input neuron and two output neurons. The output neuron $\sigma_y$ is fired if the test is true and output neuron $\sigma_n$ is fired if the test is false. In the first step, the input neuron outputs a spike after it receives a spike from the previous system. The spike is sent to $\sigma_1$, $\sigma_3$ and $\sigma_X$. Now, we consider all the three cases.
1. If $X = 0$, then the neuron $\sigma_X$ is initially empty. After the arrival of a spike from the input neuron, neurons $\sigma_1$, $\sigma_3$ and $\sigma_X$ will have a spike. In the second step, neurons $\sigma_1$ and $\sigma_X$ use their rule $a \rightarrow \pi$ and $\sigma_3$ uses its rule $a \rightarrow a$. The anti-spike from $\sigma_X$ and the spike from $\sigma_3$ are annihilated in both the neurons $\sigma_y$ and $\sigma_n$. Neuron $\sigma_2$ will have an anti-spike. $\sigma_X$ receives an anti-spike from $\sigma_1$. In the third step $\sigma_X$ uses its rule $\pi \rightarrow a$ and sends a spike to $\sigma_2$, $\sigma_y$ and $\sigma_n$. $\sigma_2$ uses its annihilation rule after it receives a spike from $\sigma_X$. In the last step, $\sigma_n$ forgets its spike and $\sigma_y$ spikes by using its rule $a \rightarrow a$, which means that the test is true.

2. If $X > 0$, say $n$, then the neuron $\sigma_X$ contains an even number of spikes. After the arrival of a spike from the input neuron, neurons $\sigma_1$ and $\sigma_3$ have a spike in each and $\sigma_X$ has odd number $(2n + 1 \geq 3)$ of spikes. $\sigma_x$ fires in the second step using its rule $a(aa)^+ / a^2 \rightarrow \pi^2$ leaving $2n - 1$ spikes. The two anti-spikes are sent to $\sigma_2$, $\sigma_y$ and $\sigma_n$. In the same step $\sigma_1$ fires by sending an anti-spike to $\sigma_X$ and $\sigma_3$ fires by sending a spike to $\sigma_y$ and $\sigma_n$. Now the neuron $\sigma_X$ has even number $(2n - 2 \geq 0)$ of spikes since it receives an anti-spike from $\sigma_1$. After the annihilation, both $\sigma_y$ and $\sigma_n$ will have an anti-spike. In the third step, $\sigma_X$ will not spike while $\sigma_2$, $\sigma_y$ and $\sigma_n$ use their rules $\pi^2 \rightarrow a^2$, $\pi \rightarrow \lambda$ and $\pi \rightarrow a$ respectively. At the end, $\sigma_X$ is left with $2n$ spikes and $\sigma_n$ spikes, which means that the test $X = 0$ is false.

3. If $X < 0$, say $-n$, then the neuron $\sigma_X$ contains an even number of anti-spike. After the arrival of a spike from the input neuron, neurons $\sigma_1$ and $\sigma_3$ have a spike in each and $\sigma_X$ has odd number $(2n - 1 \geq 1)$ of anti-spikes. Depending on the number of anti-spikes it has, neuron $\sigma_X$ fires in the second step using one of its rules $\pi(\pi a)^+ / \pi^2 \rightarrow a^2$ or $\pi \rightarrow a$ and remains with $(2n - 3) \geq 1$ or zero anti-spikes respectively. In the same step $\sigma_1$ fires and sends an anti-spike to $\sigma_X$ while $\sigma_3$ fires and sends a spike to $\sigma_y$ and $\sigma_n$. If $\sigma_X$ uses its rules $\pi(\pi a)^+ / \pi^2 \rightarrow a^2$ in step 2 (when $X \leq -2$), then it sends two spikes to $\sigma_2$, $\sigma_y$ and $\sigma_n$. After the second step, neuron $\sigma_X$ has even number $(2n - 2 \geq 2)$ of anti-spikes and do not spike in the next step. Neurons $\sigma_2$, $\sigma_y$ and $\sigma_n$ use their rules $\pi^2 \rightarrow a^2$, $a^3 \rightarrow \lambda$ and $a^3 \rightarrow a$ respectively in the third step to indicate that the condition is false. If $\sigma_X$ uses its rules $\pi \rightarrow a$ in step 2, then it sends a spike to $\sigma_2$, $\sigma_y$ and $\sigma_n$. After the second step, neurons $\sigma_2$ has a spike and both $\sigma_y$ and $\sigma_n$ will have two spikes each and $\sigma_X$ has an anti-spike. In the third step $\sigma_X$ fires using its rule $\pi^2 \rightarrow a$ and sends a spike to $\sigma_2$, $\sigma_y$ and $\sigma_n$. Neurons $\sigma_2$, $\sigma_y$ and $\sigma_n$ use their rules $\pi^2 \rightarrow a^2$, $a^3 \rightarrow \lambda$ and $a^3 \rightarrow a$ respectively in the fourth step. In both cases, at the end, $\sigma_X$ is left with $2n$ anti-spikes and $\sigma_n$ spikes by using its rule $a^3 \rightarrow a$, which means that the test $X = 0$ is false.

In all the three cases, after reading the contents of $\sigma_X$, it is restored to its original value.

To simulate the test $X \neq 0$, we simply swap the neurons $\sigma_y$ and $\sigma_n$ in Fig. 2(a). The SN PA system for the test $X > 0$ is shown in Fig. 2(b). We can observe from case 2 above that when $X > 0$, after the step 2, $\sigma_y$ and $\sigma_n$ are left with an anti-spike.
In step 3, \( \sigma_3 \) forgets the anti-spike whereas \( \sigma_y \) fires using its rule \( \pi \rightarrow a \), which means that the test \( X > 0 \) is true.

### 4.3. Implementation of Assignment Statements

The implementation of the assignment statement \( Y := X \) is shown in Fig. 3(a). The neurons \( \sigma_X \) and \( \sigma_Y \) correspond to the variables \( X \) and \( Y \) respectively. The contents of the neuron \( \sigma_Y \) are cleared before the first step of the implementation. In step one, the input neuron sends a spike to the neuron \( \sigma_X \). The number of spikes/anti-spikes in \( \sigma_X \) becomes odd. Now we again consider all the three cases.

When \( X = 0 \), \( \sigma_X \) is empty. When a spike arrives from the input neuron, there will be a spike in \( \sigma_X \), so it uses the rule \( a \rightarrow \pi \) and sends an anti-spike to \( \sigma_1 \), \( \sigma_2 \) and the output neuron \( \sigma_{out} \). The anti-spike is converted into spike inside the neurons \( \sigma_1 \) and \( \sigma_{out} \). The anti-spike is ignored in \( \sigma_2 \). The spike from \( \sigma_1 \) is sent to \( \sigma_3 \). Neuron \( \sigma_3 \) converts the spike into anti-spike and sends it to \( \sigma_4 \). The output neuron sends a spike to \( \sigma_4 \) and the next construct. In \( \sigma_4 \), the spike and anti-spike are annihilated. At the end of simulation, both the neurons \( \sigma_X \) and \( \sigma_Y \) remain empty, representing that \( X := 0 \) and \( Y := 0 \).
When \( X < 0 \), say \( X = -n \), then \( \sigma_X \) has \( 2n \) anti-spikes. When a spike arrives to \( \sigma_X \), it will have an odd number \((2n-1)\) of anti-spikes. It uses the rule \( \bar{a}(\bar{a}a) + \bar{a}^2 \rightarrow a^2 \) and sends two spikes to neurons \( \sigma_1, \sigma_2 \) and \( \sigma_{out} \). The two spikes are ignored in \( \sigma_1 \), as two spikes are stored in \( \sigma_2 \) and \( \sigma_{out} \). The process continues until \( \sigma_X \) is left with an anti-spike (which means \( 2n - 2 \) anti-spikes are sent), then it uses \( \bar{a} \rightarrow a \) and sends to \( \sigma_3 \). Neuron \( \sigma_3 \) converts anti-spike to a spike and sends this spike to neuron \( \sigma_4 \). Here \( \sigma_2 \) and \( \sigma_{out} \) will have an odd number of spikes. In the next \( n - 1 \) steps, \( \sigma_2 \) converts two of its spikes into anti-spikes and sends them to \( \sigma_X \) and \( \sigma_Y \), where as \( \sigma_{out} \) ignores the two spikes. After these \( n - 1 \) steps, \( \sigma_2 \) and \( \sigma_{out} \) are left with a spike. \( \sigma_4 \) will have two spikes and use its rule \( a^2 \rightarrow \pi^2 \) sending two anti-spikes to both \( \sigma_X \) and \( \sigma_Y \). We can observe that at the end, \( \sigma_X \) and \( \sigma_Y \) will have \( 2n \) anti-spikes which means that the contents of \( X \) are copied into \( Y \).

When \( X > 0 \), say \( X = n \), when a spike arrives to \( \sigma_X \), it will have an odd number \((2n+1)\) of spikes. The system works in the same way as above to copy the contents of \( \sigma_X \) into \( \sigma_Y \), and the extra spike in \( \sigma_X \) gets annihilated in neuron \( \sigma_4 \).

The implementation of the statement \( X := -Y \) is similar to \( X := Y \) with an extra neuron \( \sigma_5 \) to complement the output of \( \sigma_2 \) and \( \sigma_4 \) before copying it into \( \sigma_Y \).

5. \textit{if}, \textit{while}, \textit{seq}, \textit{par} and \textit{alt} Constructions

We have the implementations for variables, assignment statements and relational statements. Using the combinations of these implementations, it is possible to implement the basic constructions of the programming language described above.

5.1. The \textit{if} constructor

The statement \( \text{if} \ T \ \text{then} \ P \ \text{else} \ Q \) is implemented through the SN PA system \( \Pi_{if} \). It contains three subsystems \( \Pi_T, \Pi_P \) and \( \Pi_Q \). \( \Pi_P \) and \( \Pi_Q \) are the subsystems corresponding to the statements \( P \) and \( Q \) respectively.

\( \Pi_T \) is a structure of spiking neurons which represents the condition \( T \) of the \textit{if} statement which must be satisfied if \( P \) is to be activated. When the output neuron of the previous construct fires it sends an input spike to the \( \sigma_n \) of \( \Pi_T \). Whether or not the condition is satisfied will determine which output neuron of \( \Pi_T \) fires. If \( T \) is satisfied neuron \( \sigma_y \) is activated, and it will send a spike to the initiating neuron of the subsystem \( \Pi_P \) that, upon completion will activate the output neuron of the constructor \textit{if}. If \( T \) is not satisfied then neuron \( \sigma_n \) is activated and it will send a spike to the initiating neuron of the subsystem \( \Pi_Q \). Again upon completion of the subsystem \( \Pi_Q \), the output neuron of the constructor \textit{if} will be activated. In either case, \( \Pi_{if} \) halts after firing of its output neuron. The implementation of \textit{if} statement is shown in Fig. 4.
5.2. The while Constructor

The while statement, \( \text{while} \ (T) \ P \) is implemented in a similar way as the if statement. \( \Pi_w \) is the SN PA system for the while statement. The while constructor must first check if a condition \( T \) is satisfied and then, if it is, allows a process \( P \) to be activated. The cycle continues until the condition represented by \( T \) is no longer met. If at any point \( T \) is not met when tested, then the while constructor passes control to the next process. The implementation of while statement is shown in Fig. 5.

Again \( T \) is the condition of the while statement, \( P \) is the process to be executed while this condition is satisfied. Neuron \( \sigma_{in} \) is a neuron that triggers the execution of a process. A spike emitted by \( \sigma_{in} \) will activate \( \sigma_w \). Upon completion \( \Pi_P \) emits a spike to the input neuron of \( \Pi_w \). If the condition \( T \) is met then neuron \( \sigma_y \) is activated, otherwise neuron \( \sigma_n \) is activated.

Neuron \( \sigma_y \) is connected to the input neuron of the process \( P \). If neuron \( \sigma_n \) is triggered it will activate the output neuron of \( \sigma_w \), passing the control to the next constructor.

5.3. The seq Constructor

The function of the seq constructor is to allow a series of processes to activate sequentially, with the next process in the list only being activated once the current process is completed.

Neuron \( \sigma_{in} \) is the activation neuron of the constructor. \( P_1, P_2, \ldots, P_n \) is the sequence of processes to be activated and are implemented by disjoint subsystems; one
subsystem for each process. Only after each process is completed, then it will send an activation spike to the next process in the sequence. As Fig. 6(a) shows, \( \sigma_{in} \) sends an activation spike to \( \Pi_{P_1} \), which in turn sends an activation spike to \( \Pi_{P_2} \), which sends an activation spike to \( \Pi_{P_3} \), which upon its completion will send a spike that activates the next constructor.

![Diagram of subsystems](image)

**Fig. 5.** Implementation of the \texttt{while} Statement.

![Diagram of constructors](image)

**Fig. 6.** Implementation of \texttt{seq}, \texttt{par} and \texttt{alt} constructors.
5.4. The \textit{par} Constructor

The \textit{par} constructor activates a list of processes concurrently. As in Section 5.3, these are implemented by disjoint subsystems; one subsystem for each process. Control only passes from a \textit{par} constructor once all of the listed processes have been completed.

Once again $\sigma_{in}$ is the activation neuron of the constructor. When $\sigma_{in}$ spikes, the spike gets transmitted to the input of each of the processes to be activated. As each process completes it will send a spike to $\sigma_{\text{out}}$, see Fig. 6(b). When all processes are completed, neuron $\sigma_{\text{out}}$ accumulates $n$ spikes and fires using the rule $a^n \rightarrow a$, which passes the control to the next constructor.

5.5. The \textit{alt} Constructor

The \textit{alt} constructor (essentially non-deterministic choice) is given a set of processes implemented on disjoint subsystems, runs them concurrently, and terminates when one of the processes terminates. So the initiating neuron of the \textit{alt} sends an initiating spike to all of the initiating neurons of its constituent processes, and a spike from output neuron of any of the constituent processes is sufficient to trigger a spike from the terminating neuron of the \textit{alt}. The implementation is shown in Fig. 6(c).

6. Conclusion

In this paper we have represented variables, several operations and different programming language constructs using spiking neural P systems with anti-spikes. The discussion above and the implementations we have done with spiking neural P systems with anti-spikes suggest the possibility of a compiler which takes a statement in a simple parallel programming language and constructs an SN PA system which can execute the computational meaning of the statement. One limitation of our implementation is that the neuron for a variable is to be replicated everywhere it is used with new outgoing synapses and new post-synaptic neurons. One important open problem is to find the implementation of the constructs in such a way that neurons representing the variables can have the same pre and post synaptic neurons and any kind of operation can be performed with those neurons without replicating them. The use of other variants of SN P systems to simulate the parallel programming constructs can also be a scope of further research.

References


