OPEN ACCESS
International Journal of Neural Systems, Vol. 31, No. 1 (2021) 2050071 (16 pages)
© The Author(s)
DOI: 10.1142/S0129065720500719

World Scientific
www.worldscientific.com

Dendrite P Systems Toolbox: Representation, Algorithms and Simulators

David Orellana-Martín*, Miguel Á. Martínez-del-Amor†, Luis Valencia-Cabrera‡, Ignacio Pérez-Hurtado§, Agustín Riscos-Núñez¶ and Mario J. Pérez-Jiménez‖
Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avenida Reina Mercedes s/n, 41012 Sevilla, Spain
**dorellana@us.es*
†mdelamor@us.es
‡lvalencia@us.es
§perezh@us.es
¶ariscosn@us.es
‖marper@us.es

Accepted 30 September 2020
Published Online 16 November 2020

Dendrite P systems (DeP systems) are a recently introduced neural-like model of computation. They provide an alternative to the more classical spiking neural (SN) P systems. In this paper, we present the first software simulator for DeP systems, and we investigate the key features of the representation of the syntax and semantics of such systems. First, the conceptual design of a simulation algorithm is discussed. This is helpful in order to shade a light on the differences with simulators for SN P systems, and also to identify potential parallelizable parts. Second, a novel simulator implemented within the P-Lingua simulation framework is presented. Moreover, MeCoSim, a GUI tool for abstract representation of problems based on P system models has been extended to support this model. An experimental validation of this simulator is also covered.

Keywords: Dendrite P systems; spiking neural P systems; simulation; P-Lingua; MeCoSim.

1. Introduction

Membrane Computing is an active field of research with a large variety of applications in many scientific areas, ranging from Biology[9,10,34] to Robotics and Engineering.[35,44–46] For more insights about Membrane Computing, we refer the reader to Ref. 28. Three types of models have been extensively studied in this framework: Cell-like P systems,[27] based on the hierarchical structure of the membranes within a living cell; Tissue-like P systems,[23] based on the inter-communication between the cells in a biological tissue and Spiking Neural (SN) P systems,[19] inspired by the electrical impulses that neurons emit as information.

The latter adapts the third generation of neural networks also known as SN Networks (SNNs).[3,11,13–16,47] Information processing in natural and artificial neural systems has been widely

‡Corresponding author.

studied within the framework of Neural Systems, several modern studies, such as Refs. 1 and 2 among others, provide promising applications in a large variety of fields. Similarly, SN P systems have been also used in many applications,[36] such as in fault diagnosis of electric power systems,[31,42] efficiently solving computationally hard problems,[25,46] and in several theoretical results on computational completeness and universality of different variants.[17,24,26,37] These applications show a great potential that has attracted the attention of the researchers in the area, motivating a deep exploration on the capabilities of the models. Consequently, several variants have been studied by giving attention to different aspects of their definition: changing the number of types of spikes,[18] using a tree-like structure as the topology of the system,[43] adding new connections between neurons,[32] using structural plasticity,[6] with nonlinear spiking rules[30] or even changing the nature of the spikes,[31] among other changes.

As mentioned above, SN P systems are inspired on the functioning of neurons, which are the most important cells of the brain and nervous system, being interconnected by a complex network where direction is essential. They play an important role in receiving sensory stimuli from the external world, in processing the information encoded by the electrical signals, transforming them into chemical signals that allow certain commands to be communicated to other neurons connected to them (by means of a *synapse*), in accordance with the order associated with the network itself.

A neuron consists of three basic elements: *dendrites*, *axon* and *soma*. It can be thought as a singular tree which includes the roots (the axon), the trunk or body (the soma, which contains the nucleus of the cell, depository of the corresponding genome) and the branches and leaves (the dendrites). A (presynaptic) neuron can communicate with another (postsynaptic) neuron by sending an electrical signal, called *action potential*, through the entire axon of the presynaptic neuron, passing from the *synapse* and arriving to the dendrites of the postsynaptic neuron. In the synapse, the action potential is converted into a chemical signal in the form of *neurotransmitter* which can excite or inhibit the postsynaptic neuron from firing its action potential. Specifically, the total sum of dendritic inputs determine whether a neuron will fire an action potential.

Inspired by the role of the dendrites in the process of communication among different neurons, a new kind of formal model, called *dendrite P systems* and abbreviated as DeP systems, was recently incorporated in the membrane computing paradigm.[29] In this new dendrite-centered approach, the role of neurons is simply as storage, since the information processing is translated to the dendrites.[21] For this, an impulse will be transmitted when the neurons connected to the dendrites meet some requirements. It can be seen as a collaboration between various neurons to emit a new impulse. A motivation for such a design is that for some applications, such as computationally hard problems,[40] it is easier to solve problems having cooperating systems. Thus, DeP systems appear as a promising variant of P systems capable of reproducing the computational power of Turing machines, as demonstrated in Ref. 29.

In order to experiment with membrane systems, specially with those having a big amount of information processors, a software simulator of the corresponding variant is needed. Let us recall that simulators are required mainly because, nowadays, nor *in vitro* nor *in silico* implementation exists. P-Lingua[12] is the *de-facto* standard software to simulate P systems. The software consists on a parser that takes an input file written in the P-Lingua language and exports it in a format that a simulator can understand. The standard distribution of P-Lingua includes a generic simulator capable of computing almost every variant of P systems. In Ref. 22 for instance, a simulator for SN P systems was presented, defining the corresponding syntax for this kind of membrane systems in P-Lingua language. Thanks to this kind of software, big experiments with SN P systems can be taken over, since the simulation of these systems by hand would be a hard job. Therefore, it seems straightforward to see that a simulator of DeP systems will make simulations of these systems much easier and, therefore, more complex applications could be assumed.

Two different circumstances can appear while developing a simulator for a variant of P systems: (a) the new kind of P systems is similar to previous variants whose simulator is already implemented, with only syntactical or very simple semantic additions. In this case, the simulator can be implemented as an *extension* of the previous one, just declaring what would happen in the case that the new feature appears in the definition of the model, and including the possible small semantic variants added; and (b) the variant has many different ingredients from the variants with existing simulators. Then, a simulator (and possibly, a parser) *from scratch* should be developed. DeP systems have similarities with SN P systems, but they differ in their treating of information, with different rule types, spiking control and parallel capabilities. Thus, while in SN P each neuron fires its rules independently, according to their inner contents, in DeP systems the potential conflicts caused by the sharing of ingoing synapses by different neurons prevent such independent behavior. These aspects made it inadvisable to use the simulator for SN P systems as a base framework to develop a simulator for DeP systems.

In this work, we present a new methodology to simulate DeP systems, revisiting the formal definition of the model and providing a matrix representation of itself. It also puts special attention in the conceptual simulation algorithm and the integration of the simulator implemented inside MeCoSim[34] and P-Lingua.[12] The experimental validation of the software was also covered within our work, and some traces of the execution of the software are provided, preceded by the specification of the example model in P-Lingua, somehow providing some initial ideas about what can be done with the simulator. For further technical information, check MeCoSim website.[49]

This paper is organized as follows: DeP systems are revisited in Sec. 2. Section 3 introduces a matrix representation and a simulation algorithm for the model, Sec. 4 presents the simulator developed inside P-Lingua and MeCoSim, including an experimental validation of the software. This paper ends with conclusions and future work in Sec. 5.

2. Dendrite P Systems

In this section, this new membrane computing-based model of the neural-like type is depicted. Specifically, it is inspired from the way that neurons communicate with each other through electrical (action potential) and chemical (neurotransmitters) signals, where dendrites play a relevant role as the receiving part of neurons. Both the syntax and the semantics of the model are detailed in the following.

2.1. *Syntax*

Definition 1. A *dendrite P system* (abbreviated as DeP system), of degree $q \geq 1$, is a tuple of the form $\Pi = (O, \text{syn}, \sigma_1, \ldots, \sigma_q, i_{\text{in}}, i_{\text{out}})$, where

(1) $O = \{a\}$ is the singleton alphabet (a is called *spike*).
(2) $\text{syn} \subseteq \{1, 2, \ldots, q\} \times \{1, 2, \ldots, q\}$ with $(i, i) \notin \text{syn}$ for $1 \leq i \leq q$, is the set of arcs of a directed graph (*synapses graph*). For each node $i, 1 \leq i \leq q$, of such graph the tuple $(i_1, \ldots, i_{s_i})$ denotes the nodes verifying $(i_j, i) \in \text{syn}$ and $i_1 < i_2 < \cdots < i_{s_i}$, that is, s_i is the *indegree* of node i.
(3) Each node $i, 1 \leq i \leq q$, has associated with an object, σ_i, called *neuron*, in such manner that $\sigma_i = (n_i, R_i)$, where
   — n_i is a natural number.
   — R_i is a finite set of rewriting rules of the following form:
$$(E_{i_1}, \ldots, E_{i_{s_i}}) \,/\, a^p \leftarrow (a^{c_1}, \ldots, a^{c_{s_i}}),$$
   where E_{i_j} is a regular expression over O, for $1 \leq j \leq s_i$, and $p, c_1, \ldots, c_{s_i} \in \mathbb{N}$.
(4) $i_{\text{in}}, i_{\text{out}} \in \{1, 2, \ldots, m\}$ indicate the input and output neurons, respectively.

A *DeP system* of degree $q \geq 1$ can be viewed as a set of q *neurons* $\{\sigma_1, \ldots, \sigma_q\}$ placed in the nodes of a directed graph and they are interconnected by arcs (synapses) belonging to the set syn. Communications among neurons are materialized by means of *action potentials*, by sending an object (called *spike*), an abstraction of a quantum of energy, to all neurons linked through a synapse. Neurons from the set $\{\sigma_{i_1}, \ldots, \sigma_{i_{s_i}}\}$ are called *prepositive* (presynaptic or source) neurons of σ_i. Each neuron σ_i initially contains a number, n_i, of spikes and a finite set of rules (called *dendrite rules*) with the following format $(E_{i_1}, \ldots, E_{i_{s_i}}) \,/\, a^p \leftarrow (a^{c_1}, \ldots, a^{c_{s_i}})$. These rules are inspired by the fact that neurons, through dendrites, receive signals (synaptic inputs) from prepositive neuron axons, in such a manner that "processing" the received inputs, it will be determined which potential will be stored in the neuron (such potential will later determine the neuron's ability to fire an action potential at the next step, depending on the dendrites it is connected to). Finally, i_{in} and i_{out} are the indices of the input and output neuron, respectively. We assume that a DeP system may receive an external input on its input neuron before the computation starts, and that the output of the system is defined as the number of spikes emitted by the output neuron.

2.2. *Semantics*

A *configuration* (or *instantaneous description*) at any instant t, denoted by $\mathcal{C}_t$, of a DeP system, $\Pi = (O, \text{syn}, \sigma_1, \ldots, \sigma_q, i_{\text{in}}, i_{\text{out}})$, is described by the number of spikes associated with all neurons in the system, that is, it is a tuple $(n_1(t), \ldots, n_q(t))$ of natural numbers, where $n_i(t)$ is the number of spikes in the neuron σ_i at the instant t. In the case $t = 0$, the *initial configuration* of the system is obtained as follows: $n_i(0) = n_i$, for $1 \leq i \leq q$, being $\sigma_i = (n_i, R_i)$.

A dendrite rule $(E_{i_1}, \ldots, E_{i_{s_i}})/a^p \leftarrow (a^{c_1}, \ldots, a^{c_{s_i}})$ in the neuron σ_i is applicable to a configuration $\mathcal{C}_t$, if for all $j, 1 \leq j \leq s_i$, we have: (a) $n_{i_j}(t) \geq c_j$ and (b) $a^{n_{i_j}(t)} \in L(E_{i_j})$, that is, if the following conjunction holds: $a^{n_{i_1}(t)} \in L(E_{i_1}) \wedge \cdots \wedge a^{n_{s_i}(t)} \in L(E_{s_i}) \wedge n_{i_1}(t) \geq c_1 \wedge \cdots \wedge n_{i_{s_i}}(t) \geq c_{s_i}$. Let us recall that $\sigma_{i_1}, \ldots, \sigma_{i_{s_i}}$ are the prepositive (presynaptic) neurons associated with the neuron σ_i via the dendrites from it.

Since it is possible that two or more dendrite rules associated with a neuron σ_i can be applied to a given configuration $\mathcal{C}_t$, then one of them is nondeterministically chosen. This includes the case when two dendrite rules have a neuron in common in the left-hand side of the rule (i.e. in the set of presynaptic neurons). In each neuron, the associated dendrite rules are applied to a given configuration in a sequential way (in the sense that only one rule is applied). However, dendrite rules associated with different neurons are applied in a simultaneous way, in parallel.

Given a dendrite rule $(E_{i_1}, \ldots, E_{i_{s_i}})\,/\,a^p \leftarrow (a^{c_1}, \ldots, a^{c_{s_i}})$ in the neuron σ_i, the verification of the conditions $a^{n_{i_j}(t)} \in L(E_{i_j}) \wedge n_{i_j}(t) \geq c_j$, for $1 \leq j \leq s_i$, so that the rule is applicable to a configuration $\mathcal{C}_t$, can, informally, be interpreted as follows: all the prepositive neurons σ_{i_j} are ready to be fired, consuming c_{i_j} spikes of the $n_{i_j}(t)$ that were in the configuration $\mathcal{C}_t$. Moreover, by applying the rule, p spikes will be produced and stored in the neuron σ_i.

A configuration is a *halting configuration* if no rule of the system is applicable to it. A *computation* $\mathcal{C}$ is a (finite or infinite) sequence of configurations $\mathcal{C} = \{\mathcal{C}_n \mid n \in \mathbb{N}\}$ such that: (a) the first term, $\mathcal{C}_0$, is an initial configuration of the system; (b) for each natural number n, the configuration $\mathcal{C}_{n+1}$ is obtained from the previous one, $\mathcal{C}_n$, by applying rules of the system as described above and (c) if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration. All the computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the number of spikes received and stored by the output neuron $\sigma_{i_{\text{out}}}$ throughout the computation.

3. Conceptual Simulation Algorithm

In this section, a simulation algorithm for DeP systems is proposed. First, a matrix representation for DeP systems is provided. Later, the simulation algorithm is defined at a conceptual level by using the previously introduced matrix representation of a model. This type of designs has been successfully used by simulators for SN P systems in the literature,[48] proving to be highly parallelizable.[4,5,7] Moreover, this will help to further analyze the main differences with simulators of SN P systems, and point out the challenges for the simulators developed in Sec. 4. Finally, a discussion on how to parallelize some parts of the algorithm is provided.

3.1. *Algorithm based on matrix representation*

Let us assume that the DeP system to be simulated, Π, contains m neurons and n rules in total. Specifically, the number of rules can be also precalculated by $n = \sum_{j=1}^{m} |R_j|$. Moreover, we will assume that the time steps are measured by a variable k. The following vectors and matrices are the key ingredients of the simulation algorithm defined in what follows.

Definition 2. *Configuration vector*: C_k is a vector of size m containing all spikes in every neuron at the kth computation step/configuration, where C_0 is the initial vector containing all spikes in the system at the initial configuration.

Definition 3. *Dendrite transition matrix*: D_Π is a $n \times m$ matrix with the following elements:

$$D_\Pi[i,j] = \begin{cases} c_{ij} & \text{if rule } r_i \in R_{j'}, \text{ with } (j,j') \in \text{Syn}, \\ & \text{is applied consuming } c_{ij} \text{ spikes} \\ & \text{in the presynaptic neuron } j; \\ 0 & \text{otherwise.} \end{cases}$$

Definition 4. *Production transition matrix*: P_Π is a $n \times m$ matrix with the following elements:

$$P_\Pi[i,j] = \begin{cases} p_i & \text{if rule } r_i \in R_j \text{ and it is applied} \\ & \text{producing } p_i \text{ spikes in total;} \\ 0 & \text{otherwise.} \end{cases}$$

Definition 5. *Expression matrix*: E_Π is a $n \times m$ matrix with the following elements:

$$E_\Pi[i,j] = \begin{cases} E_{ij} & \text{if rule } r_i \in R_{j'}, \text{ with } (j,j') \in \text{Syn}, \\ & \text{and it has associated the regular} \\ & \text{expression } E_{ij} \text{ for the presynaptic} \\ & \text{neuron } j; \\ a^* & \text{if rule } r_i \in R_{j'}, \text{ with } (j,j') \notin \text{Syn}. \end{cases}$$

The reason for assigning the regular expression a^* in nonpresynaptic neurons in Definition 5 will enable the application of column-wide operations, as will be explained later in this section.

Definition 6. *Rule-neuron indexing vector*: R_Π is a vector of size n that says, for each rule i, to which rule set R_j belongs to; or in other words, the neuron where such rule is associated. Thus, if rule $r_i \in R_j$, then the value at the corresponding position for such rule is $R_\Pi[i] = j$.

Definition 7. *Applicability vector*: A_k is a vector of size n that shows, at a given configuration C_k, if a rule is applicable (having value 1) or not (having value 0 instead). For simplicity, the values 1 and 0 are considered equivalent to the Boolean values *TRUE* and *FALSE*, respectively. This vector can be updated during the simulation of one transition step, so that its contents can be changed after checking the regular expressions (when knowing which rules are applicable) and also during spike consumption (when knowing which rules will be finally applied). For simplicity, we will explicitly denote the applicability vector after checking the regular expressions as A'_k.

The above definitions can be combined with the following equation. It can be used to calculate **a** configuration vector for the $(k+1)$th step, given a configuration vector (C_k) at the kth step, and P_Π:

$$C_{k+1} = C'_k + A_k \cdot P_\Pi, \tag{1}$$

where

- $A'_k \leftarrow \text{EXPR}(C_k, E_\Pi)$;
- $A_k,\ C'_k \leftarrow \text{DEND}(A'_k, C_k, D_\Pi, R_\Pi)$.

The main procedure for simulating just one computation is given in Algorithm 3.1. There are three auxiliary functions, INIT, EXPR (Algorithm 3.2) and DEND (Algorithm 3.3). INIT function is used to initialize the matrix representation described above. No further details are given for this function, since the initialization can be easily inferred from the corresponding definitions. The main procedure is just a loop over the time steps, with a limit of L steps. This parameter is set by the user, and by default is $L = \infty$. The loop simulates a computation of the system, and each iteration computes a transition step. First, rule applicability is checked by only looking to the associated regular expressions. This is a very first step that must be performed before consuming spikes. Second, the final selection of rules is done by consuming spikes, determining in this way which rules can be applied simultaneously provided that there are enough spikes in the neurons. Third, the final step after consuming all spikes is to produce the spikes in the corresponding neuron. This can be done easily by a vector–matrix multiplication (applicability vector and production matrix).

EXPR function computes the applicability of rules in a given configuration, looking only at the associated regular expressions at the expression matrix E_Π. This is a first step in the selection process where we check which rules are candidates to be executed. This can be easily done by looping over the columns of the expression matrix (i.e. the rules), and determining, for each row (i.e. neuron), if the numbers of spikes in the current configuration C_k belong to the language generated by the corresponding regular expression. In order to implement this operation as a column-wide conjunction, those rows representing neurons that are not presynaptic contain the regular expression a^*. This way, the number of spikes in these neurons will always satisfy the regular expression, therefore not affecting the column-wide conjunction. The output of this function is an applicability vector represented by A'_k; that is, it is not a valid applicability vector yet, but a first approximation.

The calculated applicability vector after EXPR is then used in the DEND auxiliary function. Although

Algorithm 3.1. MAIN PROCEDURE: simulating one computation

```
Require: A DeP system $\Pi$ of degree $m$, with $n$ rules, and a limit $L$ of time steps.
Ensure: A computation of the system
1: $(D_\Pi, E_\Pi, P_\Pi, R_\Pi, C_0) \leftarrow$ INIT($\Pi$)
2: $k \leftarrow 0$
3: repeat
4:     $A'_k \leftarrow$ EXPR($C_k, E_\Pi$)                    ▷ Algo. 3.2
5:     $(A_k, C'_k) \leftarrow$ DEND($A'_k, C_k, D_\Pi, R_\Pi$)   ▷ Algo. 3.3
6:     $C_{k+1} \leftarrow C'_k + A_k \cdot P_\Pi$              ▷ Spike production
7:     $k \leftarrow k+1$
8: until $k \geq L \vee \bigwedge_{i=1}^{n} \neg A_k[i]$        ▷ Stop condition
9: return $C_0 \ldots C_{k-1}$          ▷ Returns the simulated computation
```

Algorithm 3.2. EXPR: checking applicability by regular expressions.

```
Require: The expression matrix $E_\Pi$, and the configuration $C_k$.
Ensure: Applicability vector $A'_k$.
1: $n \leftarrow numrows(E_\Pi)$                     ▷ Number of rules
2: $m \leftarrow numcols(E_\Pi)$                     ▷ Number of neurons
3: for $i \leftarrow 1$ to $n$ do                    ▷ For each rule
4:     $A'_k[i] \leftarrow \bigwedge_{j=1}^{m} C_k[j] \in L(E_\Pi[i,j])$   ▷ All Regex
5: end for
6: return $A'_k$
```

this function loops over the rules in random order, let us recall that this is different from nondeterminism. In fact, the algorithm uses this approach to mimic nondeterministic behavior, albeit by accepting the limitations of technology to fully capture the theoretical underlying concept. When visiting a rule, it tries to consume the corresponding spikes from all presynaptic neurons, using in this way a winner-takes-all strategy. If this is not possible, then the rule is disabled. To do this, the function uses two steps: (1) checking if a rule can consume all the corresponding spikes and (2) if it is possible, then it effectively consumes them, otherwise it disables the rule to avoid further conflicts with other rules. Moreover, following the DeP definition, the function also disables the applicability of other rules within the same neuron. Therefore, the random order to loop the rules is key for the algorithm, and there is a dependency between iterations, given that a rule can disable others.

Algorithm 3.3. DEND: checking applicability by spike consumption.

```
Require: The dendrite matrix $D_\Pi$, the rule-neuron indexing vector $R_\Pi$, an applicability vector $A'_k$ and the configuration $C_k$.
Ensure: Updated applicability vector $A_k$ and configuration vector $C'_k$.
1: $n \leftarrow numrows(D_\Pi)$                       ▷ Number of rules
2: $m \leftarrow numcols(D_\Pi)$                       ▷ Number of neurons
3: $A_k \leftarrow A'_k$                               ▷ New copy
4: $C'_k \leftarrow C_k$                               ▷ New copy
5: for $i \leftarrow 1$ to $n$ in shuffle order do     ▷ For each rule
6:     if $A_k[i]$ then                                ▷ Step 1. Test if applicable
7:         for $j \leftarrow 1$ to $m$ do              ▷ For each neuron
8:             if $C'_k[j] < D_\Pi[i,j]$ then          ▷ Rule conflict
9:                 $A_k[i] \leftarrow FALSE$           ▷ Disable rule
10:            end if
11:        end for
12:    end if
13:    if $A_k[i]$ then                                ▷ Step 2. Rule is applied
14:        for $j \leftarrow 1$ to $m$ do              ▷ For each neuron
15:            $C'_k[j] \leftarrow C'_k[j] - D_\Pi[i,j]$   ▷ Consume spikes
16:        end for
17:        $j \leftarrow R_\Pi[i]$                     ▷ Get neuron of the rule
18:        for $z \leftarrow 1$ to $n$ do              ▷ For each rule in the neuron
19:            if $z \neq i \wedge R_\Pi[z] = j$ then
20:                $A_k[z] \leftarrow FALSE$           ▷ Disable rule
21:            end if
22:        end for
23:    end if
24: end for
25: return $A_k, C'_k$
```

3.2. *Comparison with spiking neural P systems*

Simulating a transition step requires two *phases*, as generally done when simulating other P systems variants: selection and execution. However, it is also possible to identify four *sub-phases*, as shown below. Some of them can overlap as we will see later, depending on the semantics requirements of the model to simulate.

(1) Check-in rule applicability.
(2) Selection of rules to be applied.
(3) Rules' left-hand side consumption.
(4) Rules' right-hand side production.

In 2011, a matrix representation of SN P systems without delays was introduced.[48] This representation has been later on extended to different neural-like variants.[8,20] A simulation algorithm with such a representation uses selection phase with sub-phases 1 and 2, and execution phase with sub-phases 3 and 4. Let us recall the main concepts presented in this work. A transition step can be simulated by computing the following equation, which is based only on linear algebra operations:

$$C_{k+1} = S_k^z \cdot M_{\text{SNP}} + C_k, \tag{2}$$

where

- S_k^z is a *spiking vector*, which is a Boolean vector of size n. It says, for each rule, whether it is selected at step k. Super index z indicates that it is possible to have more than one spiking vector (given by nondeterminism), so the zth vector is employed.
- M_{SNP} is the *transition matrix*, which is an integer matrix of size $n \times m$. Each row is associated with a rule, and each rule with a neuron. For each rule of the form $a^c \to a^p$, the matrix contains $-c$ in the column corresponding with the neuron it belongs to, and $+p$ in the columns of neurons connected with it.
- C_k is a *configuration vector*, which is an integer vector of size m, denoting the amount of spikes within each neuron at a certain step k.

It can be seen that the matrix-based simulation algorithm for SN P systems can be easily parallelized in architectures such as GPUs.[4,5,7,8] This is empowered by a key property: the four sub-phases can be applied one after the other, without overlapping. Thus, each phase can be parallelized independently. Moreover, the second sub-phase is the most complex one, but still can be parallelized.[4] The third and fourth sub-phases can be merged into a simple vector–matrix multiplication. On the contrary, the simulation algorithm for DeP requires an overlapping of sub-phases 2 (selection) and 3 (subtraction). The reason for this is that it cannot select one rule until being certain that it will consume all the corresponding spikes. This makes the selection phase more complex, while the execution phase consists in just sub-phase 4 (production).

3.3. *A parallel design*

Although the simulation algorithm of DeP systems is less parallelizable, it is still possible to design parallel implementations from some parts as discussed next. First all, we can identify a high degree of parallelism in sub-phase 1, that is, in EXPR function. Each column can be processed by a parallel map operation, i.e. the iterations of the `for` loop in line 3 of Algorithm 3.2 can be executed independently. The column-wide conjunction in line 4 can be performed by a parallel reduction using the logical `and` operator, after applying a map in parallel to check if the numbers of spikes belong to the languages.

Moreover, sub-phase 4 (line 6 in Algorithm 3.1) can be easily parallelized, as in the case of SN P systems, because it is just a vector–matrix multiplication. Many CPU and GPU algebra libraries already provide optimized implementations of this operation. Moreover, in case that production matrix contains a majority of zero values, it would be possible to use sparse matrix libraries.

However, for sub-phases 2 and 3 (in DEND function), we need a highly sequential implementation. The `for` loop in line 5 at Algorithm 3.3 cannot be fully parallelized, unless we precompute partitions of dependencies between neurons. In other words, if the implementation knows in advance which neurons will compete one with each other, then we only need to perform the random order between those competing neurons. Neurons without dependency can be run in parallel, but their rules must be still processed sequentially (only one rule can be executed within each neuron). Moreover, the traversing of elements within the columns (`for` loops at lines 7 and 14) could be also parallelized using map and reduction operations.

4. Software Simulator

The design of new models of computation is a challenging task, with a number of aspects to consider and properly define in terms of syntactic, semantic and dynamic elements involved. The initial description of such computing devices requires a proper exhaustive formalization, usually clarified with illustrative examples. However, despite significant efforts devoted in the descriptions (to prepare correct definitions, proofs of computational results and sound examples), it is nearly impossible to guarantee the absence of inaccuracies or ambiguities in the natural language used. Occasionally, also some errors might be present in examples and simulation traces, due to the subtleties of the newly created models.

In this context, it is highly advisable for researchers to have at their disposal some support in the form of software tools to aid them in conceiving and defining new models of computation. As it might be expected, DeP systems are not an exception to this general need. As a matter of fact, during the current work certain aspects to improve were found in the foundational paper. Some of them have been revised in previous sections (e.g. reinforcing the formal definition of DeP systems), while others will be described in this section.

However, far from being taken as some type of criticism over the initial work, these aspects mentioned intend to provide a constructive feedback, highlighting the relevance of software assistants to help researchers in the tough task of the design of computing models. Therefore, a significant effort within this work was devoted to the development of sound tools to design, debug and simulate DeP systems. In what follows the approach taken to satisfy these needs is described, highlighting the main contributions in terms of the software tools provided.

4.1. *Approach*

When addressing the solution to certain software needs, one might consider different strategies depending on the aspects involved. Apart from other hybrid initiatives, these main approaches might be followed:

- Facing a completely new project, developing the software *from scratch*.
- Adapting existing solutions to the particular conditions of the new context.

Both alternatives have their strengths and weaknesses. Thus, developing a new software product from the ground implies focusing on the specific requirements for the problem under study, making the best decisions in terms of design principles, technologies applied and internal structures used. On the other hand, extending available solutions to new needs enables the developers to take advantage of certain elements, albeit at the cost of accepting the restrictions imposed by the existing software.

Along the work presented in this paper, the challenge was to handle DeP systems, a new model of computation within membrane computing. This implied considering a way to *write* these P systems, algorithms to capture the dynamics of the systems and certain tools to debug and perform virtual experiments through the simulation of such DeP systems.

As it has been pointed out in previous sections, there are similarities between SN P systems and DeP systems. This fact impulsed the bet for the second approach mentioned, so that a reasonable approach would be trying to adapt existing software working with SN P systems.

There are several interesting software products[41] to deal with different types of P systems, some of them providing open source solutions. Among all of them, a widely used mature software is P-Lingua framework, well known in membrane computing community, and chosen by the authors of this paper. This decision was based on the open source nature of the project, the availability of general-purpose tools to use in the framework and the familiarity of some of the authors of this paper as developers involved in such project.

In what follows, some of the main features of the framework provided by P-Lingua and MeCoSim are described, before detailing the main contributions made for the specification and simulation of DeP systems.

4.2. *Framework*

The growth of membrane computing theoretical achievements came along with the evolution of the related software, from didactic and specific purpose tools to general-purpose software for a variety of P systems. Such products provided solutions to support both the analysis of theoretical aspects and the modeling and experimentation with real world problems. In this context, P-Lingua[12,33] meant a significant milestone, providing a uniform framework for the specification, debugging and simulation of different types of P systems. P-Lingua defines a standard language to define P systems. Additionally, it provides a number of parsers and simulators for many variants of these computational devices.

Several products extended the capabilities of the framework, including features as invariants detection and model checking tools. As the theoretical studies by P system experts gave rise to the study of real world problems, it became clear the need of higher-level tools abstracting the users from internal details of these devices. As a consequence, Membrane Computing Simulator (`MeCoSim`[34,39]) emerged as such new higher-level layer, built on top of P-Lingua. It provided a user-friendly visual interface for P systems designers but, more importantly, a mechanism for the delivery of end-user applications for people outside membrane computing community. Thus, solutions based on P systems for certain relevant problems could be defined by the designers, who would deliver custom apps for experts in the domain of such problems (ecologists, economists, etc.) These end-users could use such apps as black boxes, to perform their visual experiments for their scenarios of interest, in order for them to make better decisions based on the predictions of the models.

In terms of the types of P systems supported, MeCoSim environment uses the parsing and simulation engines provided by P-Lingua framework. Initially, it inherited P-Lingua core capabilities in its version 4.0,[50] and later developed a fork for such engines, with many additional variants of membrane systems. For an updated list of supported models, the reader can visit MeCoSim website.[49] Besides, for an overall view of the approach, integrating P-Lingua, MeCoSim and the connection with high-performance tools, is provided in Ref. 38.

For those not familiar with this framework, it is worth recalling the basic elements provided for any type of P system covered:

- A *language* to define the P system or family of P systems under study. Given a certain problem, a simple text file in P-Lingua language would include the ingredients of the P system designed (membrane structures, rules, etc.), possibly parameterized (e.g. to cover P system families).
- A *parser* for the language dialect in order to control that the specification provided for a given P system matches the syntactic and semantic constraints imposed by the type of membrane system used.
- At least one *simulator* for the variant of P system under study. Thus, for a correct model written in P-Lingua, properly debugged using the parser for the specified model, the simulator should be able to simulate the behavior of the system (step-by-step or until a given condition is reached — typically, running for a certain number of steps, or getting to a halting configuration where no applicable rules are present).

4.3. *Contributions*

Once the needs identified were clear, the authors decided to provide the tools required by the approach described, developing them under the umbrella of the environment provided by P-Lingua and MeCoSim. Then, some components could be re-used, getting attached to certain general structures, while other aspects could be handled separately.

To summarize, the following are the main contributions to handle DeP systems with software tools:

- A language to define DeP systems through a new type of model defined in P-Lingua language, with its syntax and semantics allowing a special type of rules.
- Parsing tools for DeP systems providing a mechanism to analyze the systems introduced in P-Lingua language, inform about possible errors (if any), and build the initial configuration of the system (based on the structure and rules specified).
- A simulation engine to simulate the possible computations of the DeP system.
- The integration within P-Lingua and MeCoSim allowing the use of the command-line and visual tools for the design, debug, virtual experimentation and custom apps definition.

In what follows, some aspects about the tools provided are pointed out, while omitting certain technical details probably not too interesting for a broad audience.

Extensions of P-Lingua language

First of all, it is worth recalling that P-Lingua language presents general mechanisms for the definition of P system elements (structure, objects in multisets and rules), parameters, iterators and mechanisms to define functions following a structured programming paradigm. Such elements of the language have been preserved for the definition of DeP systems, thus enabling the P systems designers the possibility to use widely extended elements present in other types of P systems.

Second, new elements were introduced in the language in order to capture the specific ingredients of

DeP systems. In this sense, the following elements have been included:

- Model specification. A new type of model has been introduced, `dendrite`. Thus, any DeP system defined must start with the declaration:

  ```
  @model<dendrite>
  ```

  From this line onwards, the parser will be conscious of the type of system introduced, and all the elements defined along the system should match the valid structures and rules existing in DeP systems.
- Neurons present in the system. The general syntax of P-Lingua for the definition of membrane structures has been adopted, in this case, to specify the neurons assigned to `@mu` element:

  ```
  @mu = 1, 2, 3, 4;
  ```

  However, when a DeP system is being created with this syntax, different types of structures are created (in comparison with those of SN P systems), taking into account the nature of the neurons involved in this new type of model.
  As it can be seen, neurons 1–4 have been defined for the given DeP system. Some of these neurons can be marked as input or output neurons:

  ```
  @min = 1;
  @mout = 4;
  ```

- Initial multisets. Only objects of type `a` can be introduced in the membranes, as it happens in the case of SN P systems, and the general syntax for the definition of multisets in different types of P systems is adopted, with the same restriction imposed to the definition of SN P systems.

  ```
  @ms(1) = a*4;
  @ms(2) = a*2;
  ```

  In this example, four spikes are assigned to neuron 1, while two spikes are initially placed in neuron 2, and no spikes are present in the remaining neurons (3 and 4).
- Arcs definition. As formally defined in Sec. 2, DeP systems present a synapses graph, with its set of arcs representing the connections between source and target neurons. In the language defined, this follows the usual syntax present in SN P systems.

  ```
  @marcs = (1,3);
  @marcs += (2,3);
  @marcs += (2,4);
  @marcs += (3,4);
  ```

- Dendrite rules specification. This element is completely different from other types of rules defined in P-Lingua for the definition of rules for other types of P systems. The syntax used aims to be as close as possible to the definitions given in the seminal paper of DeP systems, keeping the same syntactic elements and order of the rules. Thus, a rule present in neuron 3 like $(a^4, a^2) \,/\, a^2 \leftarrow (a^2, a^2)$ can be expressed in the new dialect of P-Lingua as follows:

  ```
  ("a{4}","a{2}") [a*2 <-- (a*2, a*2)]'3;
  ```

  As those familiar with P-Lingua can observe, new syntactic elements as the reverse arrow or the parenthesis for the ordered sets of regular expressions and source spikes to be consumed are included in the language. Additionally, it is worth noting the inversion of the roles of left and right sides of the rules; in this case, the left side is devoted to the target neuron (identified by 3 in the example), with right-hand side acting as the source neurons (with their labels omitted, deduced from the synapses graph). With respect to the writing of regular expressions in P-Lingua, as it can be seen, they are surrounded by parentheses, separated by commas, each one of them in double quotes, and following the syntax accepted in Java language.[51]

Simulation of DeP systems with P-Lingua and MeCoSim

Once a P system is specified in P-Lingua with the syntax described above, and following the general mechanisms of the language, the debugging process can start. Through the parsing tools, this process checks if the definition is correct, informing about possible errors or warnings and finally generates the initial representation of the system. This is done in P-Lingua project with many for types of P systems, but a new variant imposes certain restrictions, with different ingredients, rule types, etc.

In the case of DeP systems, a significant change was made in the core of P-Lingua fork within MeCoSim. Before the existence of these systems, a number of packages were available to deal with many variants of SN P systems, all preserving certain crucial features. However, the nature of DeP systems involves a different approach where the control of the sending of spikes is not on the source neurons but on the target ones. Additionally, no rules in SN P systems can be translated into dendrite rules, and the opposite also does not hold, as the reader can see in Ref. 29. Consequently, a relevant decision was made: building a whole separated structure for all the representation and simulation of DeP systems, despite the similarities that could be externally observed for those familiar with SN P systems.

The previous facts led to the creation of new software packages children of certain `neuralLike` ones, as a superstructure for those subpackages now devoted to deal with `spiking` and the newly created `dendrite`. This applies to different packages devoted to P system definition, rules representation and simulators.

Regarding the structure of dendrite rules, their structure is far from *traditional* P system rules, both in SN P and other types of cell-like or tissue-like systems, because the left- and right-hand sides are normally devoted to express the multisets in source and target regions, with their consumed and produced objects (spikes in this case), respectively. However, in DeP systems those roles are inverted, leading to internal structures unable to inherit certain general mechanisms representing those sides in P-Lingua. Additionally, the fact that the right-hand side of the rules in this case presents an ordered set of multisets also differs from SN P systems, as it does the presence of another ordered set for the regular expressions for the corresponding respective source neurons.

All in all, the necessary structures were created to represent DeP systems in P-Lingua, and a new simulator was also built to capture the semantic and dynamic aspects of their representation. While internally this simulator differs significantly from others in the platform, it still follows the same general scheme:

(1) Initialization
(2) For each computation step, while there are applicable rules:
    (a) Selection of rules
    (b) Execution of rules.

In this case, while the initialization and execution stages do not present too many interesting aspects to highlight, the selection phase differs significantly from others, specially if compared with SN P systems. Thus, SN P system simulators in P-Lingua share a common idea: from a given configuration, the next computation step can select the rules present inside each neuron based on the contents of such neuron, and the different neurons can operate in parallel, independently. All neurons can select rules based in the same current configuration, each neuron selecting at most one single rule for this step. Thus, sequential simulators for such systems usually iterate over each neuron in order to select the one to be applied in this step, reserving the spikes to consume and produce but differing the actual application of the rules to the execution stage.

However, in the case of DeP systems, the iteration over each neuron independently cannot guarantee the correct application of the semantic and dynamic aspects of this model of computation. In contrast, every neuron where certain rules are present must be careful of other rules in different neurons but sharing common source neurons sending spikes; this is derived from the fact that if different neurons sharing the same presynaptic neuron have applicable rules at a given step, such neurons can have a potential *conflict* in the case that there are not enough spikes for all of them, and some of such neurons (and their corresponding rules to apply) must be selected nondeterministically.

Thus, in the initialization of the system, the simulator for DeP systems includes the calculation of the static dependencies of the neurons:

- `potentialConflictingTargetNeurons`: the set of all the neurons sharing some source neurons connected to them.
- `potentialConflicts`: a map storing, for each target neuron (neuron receiving an arc) with potential conflicts, the set of neurons that share some source neurons with it.

Thus, in every step of the computation, for each neuron over which we are iterating (let us say n_1), the selection phase takes into account different aspects:

- If n_1 is not a potential conflicting neuron, then the applicability of its rules can be independently handled, so it will check the matching of the corresponding regular expressions and source spikes availability, and one of the possible applicable rules is selected to be applied in this step, pseudorandomly chosen, in order to simulate a nondeterministic choice.
- Otherwise, if n_1 presents potential conflicts, then the applicability of the rules in such neuron will

be evaluated along with all the neurons potentially conflicting with it. Let us note the word "potentially", because before making any decision, the general applicability conditions (source multisets and regular expressions) will need to be checked. For these neurons potentially in conflict (n_1 and the ones possibly in conflict with it), only one of them will be nondeterministically chosen (let us say n_2), among those with applicable rules. A specific rule will be selected to be applied with such target neuron n_2. The spikes required by such rule will be removed from a temporary configuration of the system, in order to avoid the use of the same source spikes by other rules. Additionally, n_2 will be blocked in this step, given that a rule has already been selected for it. Then, if $n_2 \neq n_1$ (i.e. the neuron selected is not the one we were iterating over), then it might be the case that a rule could be still applicable. Then, the process would continue (in an internal loop) with the temporary configuration excluding the spikes reserved by the previous rule and blocking the previously selected neuron. If more neurons still were in conflict with n_1, it could still lead to other neurons getting selected, so this internal loop would continue until n_1 was selected among the currently conflicting neurons with applicable rules or no neuron was selected, meaning that none of the conflicting neurons would have applicable rules over the temporary configuration. Then, the next iteration of the external loop would visit the next neurons, ignoring those selected and blocked for this step.

Resulting from this selection phase, a set of rules will be selected. Then, the *execution phase* will apply the actual change to the current configuration, passing from C_t to C_{t+1}, removing the objects consumed in the right-hand side of the rule of each rule selected, and adding the objects produced in the left-hand side, in the target neuron, with the semantics specified in Sec. 2.

4.4. *Experimentation*

The elements described in the previous sections have implied significant changes in P-Lingua fork of MeCoSim project, and minor changes in MeCoSim itself. In reward, all the general mechanisms of these software packages are available for people interested

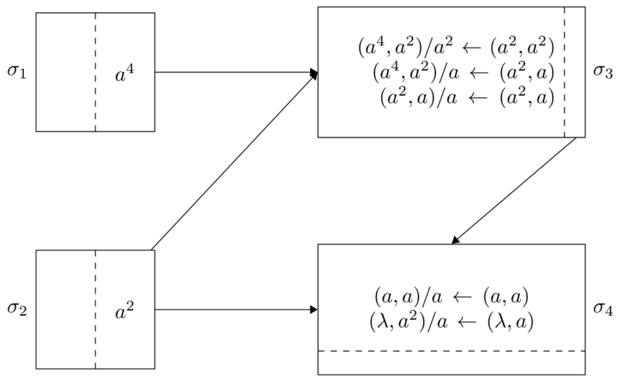

σ_1
a^4
σ_2
a^2
$(a^4, a^2)/a^2 \leftarrow (a^2, a^2)$
$(a^4, a^2)/a \leftarrow (a^2, a)$
$(a^2, a)/a \leftarrow (a^2, a)$
σ_3
$(a, a)/a \leftarrow (a, a)$
$(\lambda, a^2)/a \leftarrow (\lambda, a)$
σ_4

Fig. 1. The example in DeP systems seminal paper.

in DeP systems, along with the specific aspects highlighted in the previous section.

In order to check the correctness of the behavior of the new tools developed, the seminal paper of DeP systems[29] was taken as a reference. In such work, the authors explain in their Sec. 2.2 an example with its structure and rules (as illustrated by Fig. 1), and with the possible traces for all the possible computations of the system.

The translation of the design provided in the paper to the corresponding specification in P-Lingua code is the following:

```
@model<dendrite>

def main()
{
call dendrite_init_conf();
call dendrite_rules();
}
def dendrite_init_conf()
{
@mu = 1,2,3,4;
@ms(1) = a*4;
@ms(2) = a*2;
@marcs = (1,3);
@marcs += (2,3);
@marcs += (2,4);
@marcs += (3,4);
@min = 1;
@mout = 4;
}
def dendrite_rules()
{
("a{4}","a{2}") [a*2 <-- (a*2, a*2)]'3;
("a{4}","a{2}") [a <-- (a*2, a)]'3 ;
("a{2}","a") [a <-- (a*2, a)]'3 ;
("a","a") [a <-- (a, a)]'4;
("a{0}","a{2}") [a <-- (a*0, a)]'4;
}
```

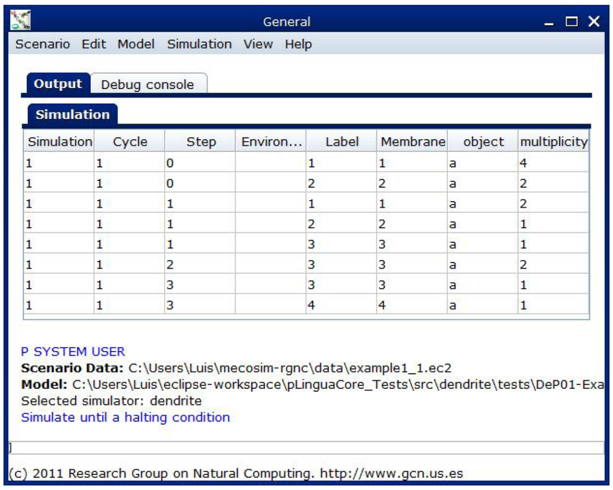
Let us note that the last rule includes a regular expression "a{0}" to represent the empty word. Similarly, in order to capture the multiplicity of objects sent from the first presynaptic neuron in such rule, the term a*0 is the multiset where no objects a are present.

In the given example, as described in the reference paper, different conflicts appear, both inside the same neuron and with different neurons involved. For each possible computation, the authors provide the trace of such computation. In this case, starting with the configuration $C_0 = (4, 2, 0, 0)$, according to the paper, three computations are possible, succinctly shown here:

- Case (1): $C_1 = (2, 0, 2, 0)$, $C_2 = (2, 0, 1, 1)$;
- Case (2a): $C_1 = (2, 1, 1, 0)$, $C_2 = (0, 0, 3, 0)$;
- Case (2b): $C_1 = (2, 1, 1, 0)$, $C_2 = (2, 0, 0, 1)$.

When checking a new simulator for a new model of computation, it might be the case that some errors can be found in the formal definition of the model, in the manual traces for a given example or in the simulation tools. Not surprisingly, in this case, one error was found in the case (2a), after experimenting intensively with the model in MeCoSim for the example provided, as illustrated in Fig. 2.

In such figure, column *multiplicity* shows the number of spikes present in the neuron labeled by column *Label*, for each given *Step*. The absence of information about a certain number of spikes in a neuron at a given step implies that no spikes are present at that moment in the given region. As it can be observed in the trace, the last step is 3. However, all the possible computations listed above presented 2 steps only and then the system halted with no rules being applicable after such two steps. On the contrary, in the trace of the figure, corresponding to case (2a), it can be noted that 2 spikes (instead of 3) are present in neuron 3, $C_2 = (0, 0, 2, 0)$. This implies that some rule becomes applicable, finally leading to a third step in the computation, and the final configuration $C_3 = (0, 0, 1, 1)$.



Simulation	Cycle	Step	Environ...	Label	Membrane	object	multiplicity
1	1	0		1	1	a	4
1	1	0		2	2	a	2
1	1	1		1	1	a	2
1	1	1		2	2	a	1
1	1	1		3	3	a	1
1	1	2		3	3	a	2
1	1	3		3	3	a	1
1	1	3		4	4	a	1

```
P SYSTEM USER
Scenario Data: C:\Users\Luis\mecosim-rgnc\data\example1_1.ec2
Model: C:\Users\Luis\eclipse-workspace\pLinguaCore_Tests\src\dendrite\tests\DeP01-Exa
Selected simulator: dendrite
Simulate until a halting condition
```

(c) 2011 Research Group on Natural Computing. http://www.gcn.us.es

Fig. 2. Snapshot of a useful trace in MeCoSim's dendrite simulator.

In order to properly follow the computation trace, the software provides a debug console to run the DeP system step-by-step, showing information about the rules applied in each computation step, along with the corresponding configurations obtained.

```
*****************************************
CONFIGURATION: 0
NEURON ID: 1, Label: 1   Multiset: a*4
NEURON ID: 2, Label: 2   Multiset: a*2
NEURON ID: 3, Label: 3   Multiset: #
NEURON ID: 4, Label: 4   Multiset: #
*****************************************
STEP: 1
Rules selected for NEURON ID: 3, Label: 3
1 * #2 (a{4}, a{2})/ [a <-- (a*2,a)]'3

CONFIGURATION: 1
NEURON ID: 1, Label: 1   Multiset: a*2
NEURON ID: 2, Label: 2   Multiset: a
NEURON ID: 3, Label: 3   Multiset: a
NEURON ID: 4, Label: 4   Multiset: #
*****************************************
STEP: 2
Rules selected for NEURON ID: 3, Label: 3
1 * #3 (a{2}, a)/ [a <-- (a*2,a)]'3

CONFIGURATION: 2
NEURON ID: 1, Label: 1   Multiset: #
NEURON ID: 2, Label: 2   Multiset: #
NEURON ID: 3, Label: 3   Multiset: a*2
NEURON ID: 4, Label: 4   Multiset: #
*****************************************
STEP: 3
Rules selected for NEURON ID: 4, Label: 4
1 * #5 (a{0}, a{2})/ [a <-- (#,a)]'4

CONFIGURATION: 3
NEURON ID: 1, Label: 1   Multiset: #
NEURON ID: 2, Label: 2   Multiset: #
NEURON ID: 3, Label: 3   Multiset: a
NEURON ID: 4, Label: 4   Multiset: a
*****************************************
Halting configuration
```

In the humble opinion of the authors of this paper, the error detected in the example of the seminal work does not respond to an unfortunate event, it is instead due to the normal impossibility for the human beings to guarantee the absence of mistakes; of course it may have occurred that no error was present for this particular case, but such faults are natural when new models are being created manually. The point of bringing this case is no other than justify the need of simulation tools to aid in the debugging process, specially when dealing with novel models where the number of aspects taken into consideration are at least worth receiving some kind of external help.

5. Conclusions

In this paper, we have presented a new simulator of a new variant of membrane systems: DeP systems. This framework evokes the information transport and processing of dendrites in the brain. In Ref. 29, this model was demonstrated to be a Turing-complete model; that is, this type of membrane systems is capable of solving all the problems that can be solved by a Turing machine. In this paper, we give an extended formalization of DeP systems (with respect to the seminal paper), as well as a matrix-based representation of such systems. The simulation algorithm is defined in such a way if the system is nondeterministic; that is, if it has various possible computational paths, so that each execution of the algorithm can lead to a different result. The complication of the applicability of rules, to perform a computation step, is the fact that rules of the same neuron are sequential, and that the information of a neuron can travel to several neurons while the number of spikes is sufficient. Because of it, rules must be, in some sense, deactivated when a conflicting rule is applied and not enough spikes are present for both. This is reflected in the simulation algorithm. The software has been validated with the example shown in the seminal paper.[29] It is worth highlighting that the purpose of this example is to test the reliability of the simulator developed, whose aim was precisely that one: to provide a first set of tools devoted to capture the syntax and semantics of the newly created model of computation of DeP systems. This simulator could be taken as a reference in order for other high-performing tools to compare again in terms of efficiency, while preserving the same behavior in terms of results obtained for each given DeP system.

This implementation opens various research lines. On the one hand, it gives researchers the opportunity to apply DeP systems in other fields, such as robotics or fault diagnosis. The integration with MeCoSim is crucial when dealing with large systems, since experts from other fields can configure themselves the scenario they want to test. On the other hand, it gives a base framework to implement new variants of DeP systems, taking into account the novelty of the model and the fact that a totally new simulator has been implemented here. Therefore, it will be easier to extend it for new variants of the model. Apart from these research lines, a matrix-representation and simulation of these extended systems would be very interesting, since there are several software platforms that could improve the performance of the computation time. In fact, GPUs are good at accelerating the simulation of P systems by means of massive parallelism. However, the more complex selection phase makes this parallelization a challenge.

Acknowledgments

This work was supported by the research project TIN2017-89842-P (MABICAP), co-financed by *Ministerio de Economía, Industria y Competitividad* (*MINECO*) of Spain, through the *Agencia Estatal de Investigación* (*AEI*) and by *Fondo Europeo de Desarrollo Regional* (*FEDER*) of the European Union.

References

1. U. R. Acharya, S. L. Oh, Y. Hagiwara, J. H. Tan and H. Adeli, Deep convolutional neural network for the automated detection and diagnosis of seizure using EEG signals, *Comput. Biol. Med.* **100** (2018) 270–278.
2. U. R. Acharya, S. L. Oh, Y. Hagiwara, J. H. Tan, H. Adeli and D. P. Subha, Automated EEG-based screening of depression using deep convolutional neural networks, *Comput. Methods Programs Biomed.* **161** (2018) 103–113.
3. M. Bernet and B. Yvert, An attention-based spiking neural network for unsupervised spike-sorting, *Int. J. Neural Syst.* **29**(8) (2019) 1850059:1–1850059:19.
4. F. G. Cabarle, H. N. Adorna and M. A. Martínez-del-Amor, Simulating spiking neural P systems without delays using GPUs, *Int. J. Nat. Comput.* **2**(2) (2011) 19–31.

5. F. G. Cabarle, H. N. Adorna, M. A. Martínez-del-Amor and M. J. Pérez-Jiménez, Spiking neural P system simulations on a high performance GPU platform, in *11th Int. Conf. Algorithms and Architectures for Parallel Processing* (Melbourne, 2011), Lecture Notes in Computer Science, Vol. 7017 (Springer, Berlin, Heidelberg, 2011), pp. 99–108.
6. F. G. C. Cabarle, H. N. Adorna, M. J. Pérez-Jiménez and T. Song, Spiking neural P systems with structural plasticity, *Neural Comput. Appl.* **26**(8) (2015) 1905–1917.
7. J. P. Carandang, J. M. B. Villaflores, F. G. C. Cabarle, H. N. Adorna and M. A. Martínez-del-Amor, CuSNP: Spiking neural P systems simulators in CUDA, *Rom. J. Inf. Sci. Technol.* **20**(1) (2017) 57–70.
8. J. P. Carandang, F. G. C. Cabarle, H. N. Adorna, N. H. S. Hernandez and M. A. Martínez-del-Amor, Handling non-determinism in spiking neural P systems: Algorithms and simulations, *Fundam. Inform.* **164**(2–3) (2019) 139–155.
9. M. A. Colomer, A. Margalida and M. J. Pérez-Jiménez, Population dynamics P system (PDP) models: A standardized protocol for describing and applying novel bio-inspired computing tools, *PLoS ONE* **8**(4) (2013) e60698.
10. P. Frisco, M. Gheorghe and M. J. Pérez-Jiménez (eds.), *Applications of Membrane Computing in Systems and Synthetic Biology* (Springer, 2014).
11. F. Galán-Prado, A Morán, J. Font, M. Roca and J. L. Roselló, Compact hardware synthesis of stochastic spiking neural networks, *Int. J. Neural Syst.* **29**(8) (2019) 1950004:1–1950004:13.
12. M. García-Quismondo, R. Gutiérrez-Escudero, M. A. Martínez-del-Amor, E. Orejuela-Pinedo and I. Pérez-Hurtado, P-Lingua 2.0: A software framework for cell-like P systems, *Int. J. Comput. Commun. Control* **4**(3) (2009) 234–243.
13. S. Ghosh-Dastidar and H. Adeli, Improved spiking neural networks for EEG classification and epilepsy and seizure detection, *Integr. Comput.-Aided Eng.* **14**(3) (2007) 187–212.
14. S. Ghosh-Dastidar and H. Adeli, Spiking neural networks, *Int. J. Neural Syst.* **19**(4) (2009) 295–308.
15. S. Ghosh-Dastidar and H. Adeli, A new supervised learning algorithm for multiple spiking neural networks with application in epilepsy and seizure detections, *Neural Netw.* **22**(10) (2009) 1419–1431.
16. R. Hu, Q. Huang, H. Wang, J. He and S. Chang, Monitor-based spiking recurrent network for the representation of complex dynamic patterns, *Int. J. Neural Syst.* **29**(8) (2019) 1950006:1–1950006:22.
17. M. Ionescu, G. Păun, M. J. Pérez-Jiménez and T. Yokomori, Spiking neural dP systems, *Fundam. Inform.* **111**(4) (2011) 423–436.
18. M. Ionescu, G. Păun, M. J. Pérez-Jiménez and A. Rodríguez-Patón, Spiking neural P systems with several types of spikes, *Int. J. Comput. Commun. Control* **6**(4) (2011) 647–655.
19. M. Ionescu, G. Păun and T. Yokomori, Spiking neural P systems, *Fundam. Inform.* **71**(2) (2006) 279–308.
20. Z. B. Jimenez, F. G. C. Cabarle, R. T. A. de la Cruz, K. C. Buño, H. N. Adorna, N. H. S. Hernandez and X. Zeng, Matrix representation and simulation algorithm of spiking neural P systems with structural plasticity, *J. Membr. Comput.* **1** (2019) 145–160.
21. R. Llinás, C. Nicholson, J. A. Freeman and D. E. Hillman, Dendritic spikes and their inhibition in alligator Purkinje cells, *Science* **160**(3832) (1968) 1132–1135.
22. L. F. Macías-Ramos, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia-Cabrera, M. J. Pérez-Jiménez and A. Riscos-Núñez, A P-Lingua based simulator for spiking neural P systems, in *12th Int. Conf. Membrane Computing* (Fontainebleau, France, 2011), Lecture Notes in Computer Science, Vol. 7184 (Springer, Berlin, Heidelberg, 2012) 257–281.
23. C. Martín-Vide, J. Pazos, G. Păun and A. Rodríguez-Patón, A new class of symbolic abstract neural nets: Tissue P systems. *Lect. Notes Comput. Sci.* **2387** (2002) 290–299.
24. L. Pan, G. Păun, G. Zhang and F. Neri, Spiking neural P systems with communication on request. *Int. J. Neural Syst.* **27**(8) (2017) 1750042.
25. L. Pan, G. Păun and M. J. Pérez-Jiménez, Spiking neural P systems with neuron division and budding, *Sci. China Inf. Sci.* **54**(8) (2011) 1596–1607.
26. L. Pan, J. Wang and H. J. Hoogeboom, Spiking neural P systems with astrocytes, *Neural Comput.* **24**(3) (2012) 805–825.
27. G. Păun, Computing with membranes, *J. Comput. Syst. Sci.* **61**(1) (2000) 108–143. Turku Center for Computer Science-TUCS Report 208, November 1998.
28. G. Păun, G. Rozenberg and A. Salomaa (eds.), *The Oxford Handbook of Membrane Computing* (Oxford University Press, Oxford, 2010).
29. H. Peng, T. Bao, X. Luo, J. Wang, X. Song, A. Riscos-Núñez and M. J. Pérez-Jiménez, Dendrite P systems, *Neural Netw.* **127** (2020) 110–120.
30. H. Peng, Z. Lv, B. Li, X. Luo, J. Wang, X. Song, T. Wang, M. J. Pérez-Jiménez and A. Riscos-Núñez, Non-linear spiking neural P systems, *Int. J. Neural Syst.* **30**(10) (2020) 2050008.
31. H. Peng, J. Wang, M. J. Pérez-Jiménez, H. Wang, J. Shao and T. Wang, Fuzzy reasoning spiking neural P system for fault diagnosis, *Inf. Sci.* **235** (2013) 106–116.
32. H. Peng, J. Yang, J. Wang, T. Wang, Z. Sun, X. Song, X. Luo and X. Huanga, Spiking neural P systems with multiple channels, *Neural Netw.* **95** (2017) 66–71.

33. I. Pérez-Hurtado, Desarrollo y aplicaciones de un entorno de programació n para Computació n Celular: P-Lingua, Ph.D. thesis, Universidad de Sevilla, Seville (2010).
34. I. Pérez-Hurtado, L. Valencia-Cabrera, M. J. Pérez-Jiménez, M. A. Colomer and A. Riscos-Núñez, MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems, in *Proc. 2010 IEEE Fifth Int. Conf. Bio-inspired Computing: Theories and Applications*, Changsha, China, 2010, pp. 637–643.
35. I. Pérez-Hurtado, M. A. Martínez-del-Amor, G. Zhang, F. Neri and M. J. Pérez-Jiménez, A membrane parallel rapidly-exploring random tree algorithm for robotic motion planning, *Integr. Comput.-Aided Eng.* **27**(2) (2020) 121–138.
36. H. Rong, T. Wu, L. Pan and G. Zhang, Spiking neural P systems: Theoretical results and applications, in *Enjoying Natural Computing, Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday* (Springer, 2018), pp. 256–268.
37. X. Song, J. Wang, H. Peng, G. Ning, Z. Sun, T. Wang and F. Yang, Spiking neural P systems with multiple channels and anti-spikes, *Biosystems* **169–170** (2018) 13–19.
38. L. Valencia-Cabrera, M.Á. Martínez-del-Amor and I. Pérez-Hurtado, A simulation workflow for membrane computing: From MeCoSim to PMCGPU through P-Lingua, in *Enjoying Natural Computing, Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday* (Springer, 2018), pp. 237–255.
39. L. Valencia-Cabrera, An environment for virtual experimentation with computational models based on P systems, Ph.D. thesis, University of Seville (2015).
40. L. Valencia-Cabrera, D. Orellana-Martín, M.Á. Martínez-del-Amor, A. Riscos-Núñez and M. J. Pérez-Jiménez, Reaching efficiency through collaboration in membrane systems: Dissolution, polarization and cooperation, *Theor. Comput. Sci.* **701** (2017) 226–234.
41. L. Valencia-Cabrera, D. Orellana-Martín, M. A. Martínez-del-Amor and M. J. Pérez-Jiménez, An interactive timeline of simulators in membrane computing, *J. Membr. Comput.* **1**(3) (2019) 209–222.
42. T. Wang, X. Wei, J. Wang, T. Huang, H. Peng, X. Song, L. Valencia-Cabrera and M. J. Pérez-Jiménez, A weighted corrective fuzzy reasoning spiking neural P system for fault diagnosis in power systems with variable topologies, *Eng. Appl. Artif. Intell.* **92** (2020) 103680.
43. T. Wu, Z. Zhang, G. Păun and L. Pan, Cell-like spiking neural P systems, *Theor. Comput. Sci.* **623** (2016) 180–189.
44. T. Wu, F. D. Bilbie, A. Paun, L. Pan and F. Neri, Simplified and yet Turing universal spiking neural P systems with communication on request, *Int. J. Neural Syst.* **28**(8) (2018) 1850013:1–1850013:19.
45. W. Xuayuan *et al.*, Multi-behaviors coordination controller design with enzymatic numerical P systems for robots, *Integr. Comput.-Aided Eng.* (2020) 1–22.
46. G. Zhang, H. Rong, F. Neri and M. J. Pérez-Jiménez, An optimization spiking neural P system for approximately solving combinatorial optimization problems, *Int. J. Neural Syst.* **24**(5) (2014) 1440006.
47. X. Zhang, G. Foderaro, C. Henriquez and S. Ferrari, A scalable weight-free learning algorithm for regulatory control of cell activity in spiking neuronal networks, *Int. J. Neural Syst.* **28**(2) (2018) 1750015:1–1750015:20.
48. X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan and M. J. Pérez-Jiménez, Matrix representation of spiking neural P systems, in *11th Int. Conf. Membrane Computing* (Jena, 2010), Lecture Notes in Computer Science, Vol. 6501 (Springer, Berlin, Heidelberg, 2011), pp. 377–392.
49. MeCoSim Website, http://www.p-lingua.org/mecosim/.
50. P-Lingua Website, http://www.p-lingua.org/.
51. Regular expressions in Java language, https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html.