A P–Lingua Based Simulator for Spiking Neural P Systems

Luis F. Macías-Ramos, Ignacio Pérez-Hurtado, Manuel García-Quismondo, Luis Valencia-Cabrera, Mario J. Pérez-Jiménez, and Agustín Riscos-Núñez

Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
{lfmaciasr,perezh,mgarciaquismondo,lvalencia,marper,ariscosn}@us.es

Abstract. The research within the field of Spiking Neural P systems (SN P systems, for short) is focusing mainly in the study of the computational completeness (they are equivalent in power to Turing machines) and computational efficiency of this kind of systems. These devices have been shown capable of providing polynomial time solutions to computationally hard problems by making use of an exponential workspace constructed in a natural way. In order to experimentally explore this computational power, it is necessary to develop software that provides simulation tools (simulators) for the existing variety of SN P systems. Such simulators allow us to carry out computations of solutions to NP-complete problems on certain instances. Within this trend, P-Lingua provides a standard language for the definition of P systems. As part of the same project, *pLinguaCore* library provides particular implementations of parsers and simulators for the models specified in P-Lingua. In this paper, an extension of the P-Lingua language to define SN P systems is presented, along with an upgrade of *pLinguaCore* including a parser and a new simulator for the variants of these systems included in the language.

1 Introduction

Spiking neural P systems were introduced in [9] in the framework of membrane computing [13] as a new class of computing devices which are inspired by the neurophysiological behaviour of neurons sending electrical impulses (spikes) along axons to other neurons. Since then, many computational properties of SN P systems have been studied; for example, it has been proven that they are Turing-complete when considered as number computing devices [9], when used as language generators [4,2] and also when computing functions [15].

Investigations related to the possibility of solving computationally hard problems by using SN P systems were first proposed in [3]. The idea was to encode the instances of decision problems in a number of spikes which are placed in an arbitrarily large pre-computed system at the beginning of the computation. It was shown that SN P systems are able to solve the **NP**-complete

problem SAT (the satisfiability of propositional formulas expressed in conjunctive normal form) in a constant time. Slightly different solutions to SAT and 3-SAT by using SN P systems with pre-computed resources were considered in [10]; here the encoding of an instance of the given problem is introduced into the pre-computed resources in a polynomial number of steps, while the truth values are assigned to the Boolean variables of the formula and the satisfiability of the clauses is checked. The answer associated with the instance of the problem is thus computed in a polynomial time. Finally, very simple semi-uniform and uniform solutions to the numerical **NP**-complete problem Subset Sum – by using SN P systems with exponential size pre-computed resources – have been presented in [11]. All the systems constructed above work in a deterministic way.

In [12], neuron division and budding were introduced into the framework of SN P systems in order to enhance the efficiency of these systems. The biological motivation of these features was founded on recent discoveries in neurobiology related to neural stem cells [6]. Neural stem cells persist throughout life within central nervous system in the adult mammalian brain, and this ensures a life-long contribution of new neurons to self-renewing nervous system with about 30000 new neurons being produced every day.

In this paper, a simulator of SN P systems with neuron division and budding is presented based on P-Lingua. P-Lingua is a programming language to define P systems [5,7,22], that comes together with a Java library providing several services; (e.g., parsers for input files and built–in simulators). In this paper we present a new release of P-Lingua. One of the innovations is an extension of the previous syntax in order to define SN P systems with neuron division and budding. Furthermore, the library has been updated to handle P-Lingua input files defining SN P systems. Finally, a new built–in simulator has been added to the library in order to simulate computations of such new models.

The paper is structured as follows. In Section 2 we introduce some definitions about SN P systems with neuron division and budding. Section 3 describes the extensions for P–Lingua programming language in order to support that kind of SN P systems. In Section 4, we introduce the simulator for these P systems presented in this paper, including the simulation algorithm. Section 5 illustrates the use of the simulator through a case study, the SAT problem, including some performance considerations. Finally, conclusions and future work are discussed in Section 6.

2 Preliminaries

This section is based on an extract from [12]. In that article, a formal description of SN P systems with neuron division and budding is outlined. Our simulator is modelled after that article, rather than the one [9] that introduced this variety of P systems, as our simulator supports extended rules.

SN P systems can be considered a variant of P systems, corresponding to a shift from *cell-like* to *neural-like* architectures. In these systems the cells (also called *neurons*) are placed in the nodes of a directed graph, called the *synapse*

graph. The contents of each neuron consist of a number of copies of a single object type, called the *spike*. Every cell may also contain a number of *firing* and *forgetting* rules. Firing rules allow a neuron to send information to other neurons in the form of electrical impulses (also called *spikes*) which are accumulated at the target cell. The applicability of each rule is determined by checking the contents of the neuron against a regular set associated with the rule. In each time unit, if a neuron can use one of its rules, then one of such rules must be used. If two or more rules could be applied, then only one of them is non-deterministically chosen. Thus, the rules are used in a sequential way in each neuron, but neurons function in parallel with each other. Note that, as it usually happens in membrane computing, a global clock is assumed, marking the time for the whole system, and hence the functioning of the system is synchronized. When a cell sends out spikes it becomes "closed" (inactive) for a specified period of time. During this period, the neuron does not accept new inputs and cannot "fire" (that is, cannot emit spikes). Furthermore, SN P systems associate a delay parameter to each rule which occurs in the system. If no firing rule can be applied in a neuron, then there may be the possibility to apply a *forgetting rule*, that removes from the neuron a predefined number of spikes.

The structure of SN P systems (that is, the synapse graph) introduced in [9] is static. Nevertheless, our simulator supports dynamical neural structures. These structures may evolve throughout computations by means of *division* and *budding* rules [12].

Formally, a *spiking neural P system with neuron division and budding* of degree $m \geq 1$ is a construct of the form:

$$\Pi = (O, H, syn, n_1, \ldots, n_m, R, in, out),$$

where:

1. $m \geq 1$ (the initial degree of the system);
2. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
3. H is a finite set of *labels* for neurons;
4. $syn \subseteq H \times H$ is a *synapse dictionary*, with $(i, i) \notin syn$ for $i \in H$;
5. $n_i \geq 0$ is the *initial number of spikes* contained in neuron i, $i \in \{1, 2, \ldots, m\}$;
6. R is a finite set of *developmental rules*, of the following forms:
   (1) *extended firing* (also called *spiking*) rule $[E/a^c \rightarrow a^p; d]_i$, where $i \in H$, E is a regular expression over a, and $c \geq 1$, $p \geq 0$, $d \geq 0$, with the restriction $c \geq p$;
   (2) *neuron division rule* $[E]_i \rightarrow []_j \parallel []_k$, where E is a regular expression and $i, j, k \in H$;
   (3) *neuron budding rule* $[E]_i \rightarrow []_i/[]_j$, where E is a regular expression and $i, j \in H$;
7. $in, out \in H$ indicate the *input* and the *output* neurons of Π.

Note that the above definition is slightly different from the usual one found in the literature, where the neurons initially present in the system are explicitly listed as $\sigma_i = (n_i, R_i)$, where $1 \leq i \leq m$ and R_i is the set of the rules associated

with neuron with label i. In what follows we will refer to neuron with label $i \in H$ also denoting it with σ_i.

If an extended firing rule $[E/a^c \to a^p; d]_i$ has $E = a^c$, then we will write it in the simplified form $[a^c \to a^p; d]_i$; similarly, if a rule $[E/a^c \to a^p; d]_i$ has $d = 0$, then we can simply write it as $[E/a^c \to a^p]_i$; hence, if a rule $[E/a^c \to a^p; d]_i$ has $E = a^c$ and $d = 0$, then we can write $[a^c \to a^p]_i$. A rule $[E/a^c \to a^p]_i$ with $p = 0$ is written in the form $[E/a^c \to \lambda]_i$ and is called *extended forgetting* rule. Rules of the types $[E/a^c \to a; d]_i$ and $[a^c \to \lambda]_i$ are said to be *standard*.

In addition to spiking rules, our simulator supports neuron *division* and neuron *budding* rules. Basically, division rules create a pair of new membranes out of a previously existing one (that disappears). These membranes are placed in parallel. Membranes created by means of division rules inherit the synapses going into and out of the original membrane. On the other hand, budding rules create a new membrane while preserving the original one. This new membrane is placed serially following the original one. Furthermore, the new membrane created by means of budding rules inherits the outgoing synapses of the parent membrane. These synapses are transferred from the parent to the child membrane. In addition, a synapse from the original membrane to the newly created membrane is established. Thus, after the application of division rules two new membranes with (potentially) new labels are created. The original one disappears. On the other hand, after the application of budding rules only a new membrane with a new (potentially) label is created, preserving the original membrane. In both cases additional synapses are added connecting the child membranes with pre-existent membranes according to the rules dictated by the synapse dictionary. As a result of the simulator supporting both division and budding rules, the membrane structure of the system is able to evolve dynamically along the computation steps. A more thorough description of the semantics of SN P systems can be found in [12] and in [18].

The *configuration* of the system is described by the topological structure of the system, the number of spikes associated with each neuron, and the *state* of each neuron (open or closed). Using the rules as described above, one can define *transitions* among configurations. Any (maximal) sequence of transitions starting in the initial configuration is called a *computation*. A computation *halts* if it reaches a configuration where all neurons are open and no rule can be used.

Traditionally, the input of an SN P system used in the accepting mode is provided in the form of a spike train, a sequence of steps when one spike or no spike enters the input neuron. Thus, we need several spikes at a time to come into the system via the input neuron, i. e., we consider "generalized spike trains" instead, written in the form $a^{i_1} \cdot a^{i_2} \cdot \ldots \cdot a^{i_r}$, where $r \geq 1, i_j \geq 0$ for each $1 \leq j \leq r$. The meaning is that i_j spikes are introduced in neuron σ_{in} in step j (all these i_j spikes are provided at the same time).

3 P-Lingua Syntax for SN P Systems

This section is devoted to show the extended P-Lingua syntax that covers the SN P systems considered in the proposed P-Lingua language revision. The following paragraphs describe the new features incorporated in order to support SN P systems. These features include different simulation modes, an initial membrane structure (which consists of the initial set of membranes and a collection of synapses), a simulation input spike train and the explicit (and optional) specification of input and output membranes. These features also include language-related items, such as regular expressions and new reserved words.

3.1 Regular Expressions

Regular expressions have been introduced. These regular expressions are a subset of those defined according to the Java package *java.util.regex* formally specified at [20]. This subset is defined by combining the following symbols as established in the syntax just mentioned:

```
'a', '(', ')', '[', ']', '{', '}', ',', '^', '*', '+', '?', '|'
```

The purpose of the inclusion of regular expression in this version of P-Lingua is to specify the regular expressions associated to the rules of SN P systems.

Additional information related to the Java language can be found at [21]. We remark that these regular expressions are not checked by the parser. Instead, they are piped directly into the simulator. Then, the simulator performs the parsing. Regular expressions E are written double-quoted: *"E"*. Also, in this version the membrane labels can accept parameters built on integer expressions. In addition, no new arithmetic operator has been introduced in this revision.

3.2 Model Specification

In this version of P-Lingua, a SN P system specification must define an initial membrane structure and a set of rules. The membrane structure is composed of a set of membranes joined by synapses. These synapses are specified as a set of connections. If the SN P system considers division and budding rules, then a synapse dictionary must be also specified.

Apart from the initial configuration and the set of rules, in this version of P-Lingua the user can also define an input membrane and a set of output membranes. In addition, the user can also define simulation parameters. These parameters include simulation modes and spike trains.

P-Lingua files defining SN P systems must begin with the following sentence:

```
@model<spiking_psystems>
```

Additionally, `@model<spiking_psystems>` can be followed by a pair of sentences in order to specify the simulation modes to be used. These modes determine the semantics in which the simulation is performed as described in [8]. The sentences are:

- `@masynch = v1;`
  where $v1 \in \{0, 1, 2\}$. If this sentence is not present, then *v1* defaults to 0. These values denote the following modes:
  0: *Synchronous (standard)* mode.
  1: *Asynchronous unbounded* mode.
  2: *Asynchronous bounded* mode.

  Let us consider a membrane structure with N membranes. If `@masynch` is set to 2 then the next sentence can be used to express the valid halting configuration:

  ```
  @mvalid = (m1,n1), (m2,n2),..., (mN,nN);
  ```

  where, for each integer $i \in [1, \ldots, N]$:
  - m_i is a membrane label in the SN P system.
  - n_i is an integer expression which specifies the number of spikes contained in m_i at the end of the computation.

  If the sentence is not used then every halting configuration is considered valid. Also if the sentence is used when `@masynch` is not set to 2, it would be ignored.

- `@mseq = v2;`
  where $v2 \in \{0, \ldots, 5\}$. If this sentence is not present, then *v2* defaults to 0. These values denote the following modes:
  0: *parallel (standard)* mode.
  1: *standard sequential* mode.
  2: *max pseudo-sequential* mode.
  3: *max sequential* mode.
  4: *min pseudo-sequential* mode.
  5: *min sequential* mode.

Let us consider an initial membrane structure of a SN P system with N membranes and M synapses. In this version of P-Lingua, in order to define that initial membrane structure, the following sentence must be written:

```
@mu = m1, m2, ..., mN;
```

where, for each integer $i \in [1, \ldots, N]$, m_i is the label of membrane i. The label *environment* cannot be used. Given an initial membrane structure, in order to define the connections between the membranes, the following sentence must be written:

```
@marcs = arc1, arc2,..., arcM;
```

where, for each integer $i \in [1, \ldots, M]$, $arc_i = (m_k, m_l)$, m_k and m_l being two membrane labels of an SN P system configuration and $m_k \neq m_l$.

Let us consider a dictionary. In this version of P-Lingua, in order to specify that synapse dictionary the following optional sentence must be written:

```
@mdict = e1, e2, ..., eD;
```

where D is the number of entries of the dictionary and, for each integer $p \in [1, \ldots, D]$, $e_p = (l_i, l_j)$, l_i and l_j are two different labels and $l_i \neq l_j$.

If the SN P system specification contains division or budding rules the dictionary is mandatory. Besides, an implicit dictionary is built from the parameter `@marcs`. This dictionary is extended by the "explicit" dictionary defined by the sentence `@mdict`.

In order to specify the (optional) input membrane of the SN P system, the following sentence may be written:

```
@min = m;
```

where m is the label of a membrane existing in the initial SN P system membrane structure. SN P systems without input membrane can also be specified in P-Lingua, by just omitting this sentence.

Let us consider an input spike train consisting of S steps for a SN P system with an input membrane. In this version of P-Lingua, in order to specify this input spike train, the following sentence must be written:

```
@minst = r1, r2, ..., rS;
```

where, for each integer $i \in [1, \ldots, S]$, $r_i = (i, a_i)$, i and a_i being two integer expressions, $i > 1$ and $a_i \geq 0$. The parameter i denotes the step of the computation calculated by the simulator while the parameter a_i denotes the number of spikes that are introduced in the input membrane at step i. When the pair $r_j = (j, a_j)$ is undefined for some step j then the simulator assumes that zero spikes are introduced in the system at that step.

In order to specify the (optional) output membranes of the SN P system, the following sentence may be written:

```
@mout = o1, o2, ..., oL;
```

where, for each integer $i \in [1, \ldots, L]$, o_i denotes a membrane label of the SN P system. SN P systems without output membranes can also be specified in P-Lingua, by just omitting this sentence.

3.3 Sample of the First Few Lines of a P-Lingua SN P System Definition File

The following paragraph shows the first few lines of a P-Lingua SN P system definition file. These lines cover examples of the previously defined syntax.

```
@model<spiking_psystems>

@masynch = 2;
@mvalid = (1, 3), (2, 6), (3, 4);

@mseq = 2;

@mu = 1,2,3;
@marcs = (1,2), (1,3);
@mdict = (1,d),(2,f);
@min = 1;
@minst = (1,3), (5,4), (8,2);
@mout = 1,2;
```

3.4 Definition of Multisets

Initial multisets of objects for neurons can be defined in the same way as initial multisets of objects for cell-like membranes, with the restriction that only the object a may be used.

For instance:

```
@ms(0) = a*15;
```

3.5 Definition of Rules

If a P-Lingua file begins with the `@model<spiking_psystems>` sentence, then four types of rules can be defined:

(1) *Firing rules*, that can be specified in the following ways:
   - `[a*c]'h --> [a*p]'h "e" :: d;`
   - `[a*c    -->  a*p]'h "e" :: d;`

(2) *Forgetting rules*, that can be specified in the following ways:
   - `[a*c]'h --> [#]'h "e" :: d;`
   - `[a*c    -->  #]'h "e" :: d;`

(3) *Neuron division rules*, that can be specified in the following way:

```
[]'i --> [#]'j || [#]'k "e";
```

(4) *Neuron budding rules*, that can be specified in the following way:

```
[]'i --> [#]'i / [#]'j "e";
```

where h, i, j and k are membrane labels of the SN P system described, c, p and d are integer expressions which satisfy $c \geq 1$, $c \geq p$ and $d \geq 0$, and e is a regular expression over $\{a\}$.

For firing and forgetting rules, both d and *"e"* are optional with d defaulting to 0. In forgetting rules d is always set to 0. When e is not present in the rule, then e defaults to the left hand side of the rule.

Division and budding rules have no delays, so d is not used, but the regular expression e is mandatory.

For instance, the following rules are valid spiking rules in P-Lingua:

```
– [a*3]'1 --> [a*2]'1 "a^3" :: 3;
– [a*3    -->  a*2]'1 "a?" :: 6;
– [a*3    -->    #]'1 "a+a^3" :: 3;
– [a*3    -->    #]'1 "a*" :: 5;
```

Also, the following rules are valid division and budding rules in P-Lingua:

```
– []'1 --> []'2 || []'3 "a*";
– []'1 --> []'1  / []'2 "(a^3)|a";
```

Recall that in P-Lingua, the symbol # is optional (it can be omitted), for instance the following rules:

```
– []'1 --> [#]'2 || [#]'3 "a*";
– []'1 --> [#]'1 / [#]'2 "(a^3)|a";
– [a*3 --> #]'1 "a+a^3" :: 3;
– [a*3 --> #]'1 "a*" :: 5;
```

can be written equivalently as:

```
– []'1 --> []'2 || []'3 "a*";
– []'1 --> []'1 / []'2 "(a^3)|a";
– [a*3 --> ]'1 "a+a^3" :: 3;
– [a*3 --> ]'1 "a*" :: 5;
```

3.6 Definition of the Output

In order to specify additional output results to be shown to the user after the computation halts (when it halts), the following sentences may be included:

```
@moutres_binary;
@moutres_natural(k,strong,alternate);
@moutres_summatories;
```

where all of them are optional.

When parameter `@moutres_binary` is specified then the output is shown as a binary spike train. If more than one output membrane was defined then a binary output spike train is shown for each one of them. The binary spike train for a given output membrane o_j is a binary sequence $b_1, b_2, ..., b_N$ with $b_i = 0$ if and only if o_j sends no spikes to the environment at computation step i, and 1 otherwise.

When parameter `@moutres_natural` is specified then a natural output spike train is shown to the user. If more than one output membrane was defined then a

train is shown for each output membrane. For a thorough explanation please refer to [14]. If *natural(k,strong,alternate)* is specified, then k is an integer expression with $k \geq 2$ and *strong* and *alternate* take boolean values (*false* or *true*).

When parameter `@moutres_summatories` is specified then the sum of the spikes sent to the environment for each output membrane, (i.e., the contents of the environment) is shown as output.

3.7 Reserved Words

The set of reserved words has been updated in the current syntax of the P-Lingua language by adding the following text strings:

```
@masynch, @mseq, @mvalid, @marcs, @mdict, @min, @minst,
@mout, |, @moutres_binary, @moutres_natural, @moutres_summatories
```

The inclusion of these reserved words is necessary in order to include the new features of this version of P-Lingua. These new features are explained above.

4 A Simulator Software for SN P Systems

In [7], a Java library called *pLinguaCore* was presented under GPL license. It includes parsers to handle input files and built–in simulators to generate P system computations. It can export several output file formats to represent P systems. It is not a closed product because developers with knowledge of Java can add new components to the library. In this paper, *pLinguaCore* has been upgraded to support SN P systems. Now, the library is able to handle input P–Lingua files which define SN P systems and it includes a new built–in simulator in order to simulate SN P system computations. The current version of the library can be downloaded from `http://www.p-lingua.org`.

The simulation algorithm described below generates one possible computation for a SN P system with an initial configuration C_0 containing n membranes $m_1, \ldots, m_n$. Recall that when working with recognizer P systems all computations yield the same answer (confluence).

The simulation algorithm is structured in six steps:

I. Initialization
In this step the data structures needed to conduct the simulation are initialized.

II. Selection of rules
In this step the set of rules to be executed in the current step is calculated.

III. Build execution sets
In this step the rules to be executed are split into different sets, according to their kind.

IV. Execute division and budding rules
In this step division and budding rules are executed. The execution is performed in two phases: In the first one, new neurons are calculated out of existing

neurons by applying budding and division rules. In the second one additional synapses are introduced according to the synapse dictionary.

V. Execute spiking rules
In this step execution of spiking rules is performed.

VI. Ending
In this step the current configuration is updated with the configuration newly calculated and the halting condition is checked (no rules are applicable).

In what follows, the simulation algorithm is described.

I. Initialization

1. Let C_t be the current configuration
2. Let $M_{sel} \equiv \emptyset$ be a set of membranes who are susceptible of executing a rule in the current computation step
3. Let m_0 be a virtual membrane (with label 0) representing the environment

II. Selection of rules

1. Each membrane m_i stores the following elements:
   - last rule r_i selected to be executed in a previous step for that membrane (initially none)
   - an integer decreasing-only counter d_i, that stores the number of steps left for the membrane to open and fire in case r_i is a firing rule (initially zero).
   For each membrane m_i, do
   (a) If m_i is closed as a result of being involved in the execution of a budding or division rule, then open m_i (let $d_i = 0$) and clear its rule r_i
   (b) If m_i is closed as a result of being involved in the execution of a firing rule (thus r_i is a firing rule) then
      i. Decrease the counter d_i
      ii. Add m_i to M_{sel}
      iii. Go to process the next membrane
   (c) Let $S_i \equiv \emptyset$ be the set of possible rules to be executed over m_i
   (d) For each rule r_j with label j do
      i. If r_j is active and can be executed over m_i then add r_j to S_i
   (e) If S_i is empty then go to process the next membrane
   (f) Select non deterministically a rule r_k from S_i
   (g) Set r_k as the new selected rule for m_i
   (h) If r_k is a firing rule, update the counter d accordingly
   (i) Add m_i to M_{sel}
   (j) Clear S_i
2. If M_{sel} is not empty and the simulator operates in sequential mode then
   (a) Select a membrane m_s from M_{sel} according to the sequential mode
   (b) Clear M_{sel}
   (c) Add m_s to M_{sel}

III. Build execution sets

1. Let $Division \equiv \emptyset$ be the set that stores the membranes having a division rule selected to be executed in the current step
2. Let $Budding \equiv \emptyset$ be the set that stores the membranes having a budding rule selected to be executed in the current step
3. Let $Spiking \equiv \emptyset$ be the set that stores the membranes having a spiking rule selected to be executed in the current step (or susceptible to be executed in the case of firing rules with delays)
4. For each membrane m_i from M_{sel} do
   (a) Let r_i be the selected rule for m_i
   (b) If r_i is a division rule then add m_i to $Division$
   (c) If r_i is a budding rule then add m_i to $Budding$
   (d) If r_i is a spiking rule then add m_i to $Spiking$

IV. Execute division and budding rules

1. Let $Div \equiv \emptyset$ be the set that stores the membranes that are generated as a result of applying a division rule in the current step
2. Let $Bud \equiv \emptyset$ be the set that stores the membranes that are generated as a result of applying a budding rule in the current step
3. For each membrane m_i from $Division$ do
   (a) If the simulator operates in asynchronous mode then
      i. Determine non deterministically if the rule has to be executed
      ii. If the rule does not have to be executed then go to process the next membrane
   (b) Let r_i be the selected rule for m_i: $[E]_i \rightarrow []_j || []_k$
   (c) Relabel m_i with the j label, thus from now on we refer to m_j
   (d) Create a new membrane m_k and close it
   (e) For each incoming edge from some membrane m_p to m_j create a new edge from m_p to m_k
   (f) For each outgoing edge from m_j to some membrane m_p create a new edge from m_k to m_p
   (g) Add m_j and m_k to Div
4. For each membrane m_i from $Budding$ do
   (a) If the simulator operates in asynchronous mode then
      i. Determine non deterministically if the rule has to be executed
      ii. If the rule has not to be executed then go to process the next membrane
   (b) Let r_i the selected rule for m_i: $[E]_i \rightarrow []_i / []_j$
   (c) Create a new membrane m_j and close it
   (d) For each outgoing edge from m_i to some membrane m_p do
      i. Create a new edge from m_j to m_p
      ii. Remove the edge from m_i to m_p
   (e) Create a new edge from m_i to m_j
   (f) Add m_j to Bud
5. For each membrane m_i from Div create new edges involving m_i according to the synapse dictionary if necessary

6. For each membrane m_i from Bud create new edges involving m_i according to the synapse dictionary if necessary

V. Execute spiking rules

1. For each membrane m_i from $Spiking$ do
   (a) If m_i is closed then go to process the next membrane
   (b) If the simulator operates in asynchronous mode then
      i. Determine non deterministically if the rule has to be executed
      ii. If the rule does not have to be executed then go to process the next membrane
   (c) Let r_i be the selected rule for m_i
   (d) If r_i is a firing rule of the form $[E/a^c \rightarrow a^p; d]_i$ then
      i. Remove c spikes from the multiset of m_i
      ii. For each membrane m_j connected to m_i by an edge going from m_i to m_j, add p spikes to the multiset of m_j if and only if m_j is open
   (e) If r_i is a forgetting rule of the form $[E/a^c \rightarrow \lambda]_i$ then remove c spikes from the multiset of m_i

VI. Ending

1. Let $C_{t+1} = C_t$
2. If M_{sel} is not empty then goto **I**

Before going on, it is important to note that we assume the defined P system to be free of syntax errors that could lead to an incorrect computation, since the P–Lingua parser checks for any possible programming errors.

5 An Example: A Family of SN P Systems Solving SAT

In order to illustrate the operation of the simulator, a uniform and efficient solution to `SAT` by means of a family of SN P systems with spiking, neuron division and budding rules is described in this section.

We will start considering the description of the family in terms of its theoretical definition. Then a P–Lingua source code describing the SN P system for a particular instance of the problem will be shown. Subsequently, a generalized version of the P–Lingua previous code for any given instance of the `SAT` problem will be presented. This is achieved by means of the inclusion of parameters in the P–Lingua code and the integration with MeCoSim [16].

Finally the simulation results will be presented.

5.1 Description of the Family

As mentioned above, a uniform and efficient solution to `SAT` by means of a family of SN P systems with spiking, neuron division and budding rules is described in what follows. This solution has already been presented in [12].

Let us consider a propositional formula $SAT(n,m) = C_1 \wedge \ldots \wedge C_m$ over n variables $x_1 \ldots x_n$, consisting of m clauses $C_j = y_{j,1} \vee \ldots \vee y_{j,k_j}$, $1 \leq j \leq m$, where $y_{j,i} \in \{x_l, \neg x_l \mid 1 \leq l \leq n\}$, $1 \leq i \leq k_j$. Without loss of generality, we may assume that no clause contains two occurrences of some x_i or two occurrences of some $\neg x_i$ (the formula is not redundant at the level of clauses), or both x_i and $\neg x_i$ (otherwise such a clause is trivially satisfiable, hence can be removed).

Because the construction is uniform, we need a way to encode any given instance $\gamma_{n,m}$ of C. Each clause C_i of $\gamma_{n,m}$ can be seen as a disjunction of at most n literals, and thus for each $j \in \{1, 2, \ldots, n\}$ either x_j occurs in C_i, or $\neg x_j$ occurs, or none of them occurs. In order to distinguish these three situations we define the *spike variables* $\alpha_{i,j}$, for $1 \leq i \leq m$ and $1 \leq j \leq n$, as variables whose values are amounts of spikes; we assign to them the following values:

$$\alpha_{i,j} = \begin{cases} a, & \text{if } x_j \text{ occurs in } C_i; \\ a^2, & \text{if } \neg x_j \text{ occurs in } C_i; \\ a^0, & \text{otherwise.} \end{cases}$$

In this way, clause C_i will be represented by the sequence $\alpha_{i,1} \cdot \alpha_{i,2} \cdot \ldots \cdot \alpha_{i,n}$ of spike variables; in order to represent the entire formula $\gamma_{n,m}$ we just concatenate the representations of the single clauses, thus obtaining the generalized spike train $\alpha_{1,1} \cdot \alpha_{1,2} \cdot \ldots \cdot \alpha_{1,n} . \alpha_{2,1} \cdot \alpha_{2,2} \cdot \ldots \cdot \alpha_{2,n} \cdot \ldots \cdot \alpha_{m,1} \cdot \alpha_{m,2} \cdot \ldots \cdot \alpha_{m,n}$. As an example, the representation of $\gamma_{3,2} = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_3)$ is the sequence $a \cdot a^2 \cdot a^0 \cdot a \cdot a^0 \cdot a$. In order to let the systems have enough time to generate necessary workspace before computing the instances of $SAT(n,m)$, a spiking train $(a^0)^{2n}$ is added at the beginning of the input. In general, for any given instance $\gamma_{n,m}$ of $SAT(n,m)$, the encoding sequence is $cod(\gamma_{n,m}) = (a^0)^{2n} \cdot \alpha_{1,1} \cdot \alpha_{1,2} \cdot \ldots \cdot \alpha_{1,n} \cdot \alpha_{2,1} \cdot \alpha_{2,2} \cdot \ldots \cdot \alpha_{2,n} \cdot \ldots \cdot \alpha_{m,1} \cdot \alpha_{m,2} \cdot \ldots \cdot \alpha_{m,n}$.

For each $n, m \in \mathbb{N}$, we construct

$$\Pi(\langle n, m \rangle) = (O, H, syn, n_1, \ldots, n_q, R, in, out),$$

with the following components:

The initial degree of the system is $q = 4n + 7$;

$O = \{a\}$;

$H = \{in, out, cl\} \cup \{d_i \mid i = 0, 1, \ldots, n\}$
$\cup \{Cx_i \mid i = 1, 2, \ldots, n\} \cup \{Cx_i 0 \mid i = 1, 2, \ldots, n\}$
$\cup \{Cx_i 1 \mid i = 1, 2, \ldots, n\} \cup \{t_i \mid i = 1, 2, \ldots, n\}$
$\cup \{f_i \mid i = 1, 2, \ldots, n\} \cup \{0, 1, 2, 3\}$;

$syn = \{(d_i, d_{i+1}) \mid i = 0, 1, \ldots, n-1\} \cup \{(d_n, d_1)\}$
$\cup \{(in, Cx_i) \mid i = 1, 2, \ldots, n\} \cup \{(d_i, Cx_i) \mid i = 1, 2, \ldots, n\}$
$\cup \{(Cx_i, Cx_i 0) \mid i = 1, 2, \ldots, n\} \cup \{(Cx_i, Cx_i 1) \mid i = 1, 2, \ldots, n\}$
$\cup \{(i+1, i) \mid i = 0, 1, 2\} \cup \{(1, 2), (0, out)\}$
$\cup \{(Cx_i 1, t_i) \mid i = 1, 2, \ldots, n\} \cup \{(Cx_i 0, f_i) \mid i = 1, 2, \ldots, n\}$;

$n_{d_0} = n_0 = n_2 = n_3 = 1$, $n_{d_1} = 6$, and there is no spike in the other neurons;

R is the following set of rules:

(1) **spiking rules:**

$[a \to a]_{in}$,
$[a^2 \to a^2]_{in}$,
$[a \to a; 2n + nm]_{d_0}$,
$[a^4 \to a^4]_i$, $i = d_1, \ldots, d_n$,
$[a^5 \to \lambda]_{d_1}$,
$[a^6 \to a^4; 2n + 1]_{d_1}$,
$[a \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^2 \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^4 \to \lambda]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^5 \to a^5; n - i]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^6 \to a^6; n - i]_{Cx_i}$, $i = 1, 2, \ldots, n$,
$[a^5 \to a^4]_{Cx_i 1}$, $i = 1, 2, \ldots, n$,
$[a^6 \to \lambda]_{Cx_i 1}$, $i = 1, 2, \ldots, n$,
$[a^5 \to \lambda]_{Cx_i 0}$, $i = 1, 2, \ldots, n$,
$[a^6 \to a^4]_{Cx_i 0}$, $i = 1, 2, \ldots, n$,
$[(a^4)^+ \to a]_{t_i}$, $i = 1, 2, \ldots, n$,
$[(a^4)^+ \to a]_{f_i}$, $i = 1, 2, \ldots, n$,
$[a^{4k-1} \to \lambda]_{t_i}$, $k = 1, 2, \ldots, n$, $i = 1, 2, \ldots, n$,
$[a^{4k-1} \to \lambda]_{f_i}$, $k = 1, 2, \ldots, n$, $i = 1, 2, \ldots, n$,
$[a^m \to a^2]_{cl}$,
$[(a^2)^+ / a \to a]_{out}$,
$[a \to a]_i$, $i = 1, 2$,
$[a^2 \to \lambda]_2$,
$[a \to a; 2n - 1]_3$;

(2) **neuron division rules:**

$[a]_0 \to []_{t_1} \parallel [\,]_{f_1}$,
$[a]_{t_i} \to []_{t_{i+1}} \parallel [\,]_{f_{i+1}}$, $i = 1, 2, \ldots, n - 1$,
$[a]_{f_i} \to []_{t_{i+1}} \parallel [\,]_{f_{i+1}}$, $i = 1, 2, \ldots, n - 1$;

(3) **neuron budding rules:**

$[a]_{t_n} \to [\,]t_n / [\,]cl$,
$[a]_{f_n} \to [\,]f_n / [\,]cl$.

5.2 A P-Lingua Source Code. Particular Instance Solution of SAT Problem

Code description. This section is devoted to the P–Lingua source code that defines a SN P system belonging to the family specified above. In particular, we consider the following formula as the instance of `SAT` that we try to solve:

$$\varphi = (x_1 \vee x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge \\ \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4) \wedge (x_1 \vee x_4) \wedge (\neg x_1 \vee x_2 \vee x_3 \vee x_4)$$

Thus we are resolving `SAT(n,m)` with `n = 4` and `m = 8`. Of course, the `main` module can be easily modified in order to define any other P system of the family.

The source code is structured as follows:

1. Module `main()` that defines a SN P system solving the `SAT` problem for the formula described above with 4 variables and 8 clauses. Firstly, it calls the module `spiking_init_conf(n)` for $n \equiv 4$. Secondly, it calls the module `spiking_rules(n,m)` for $(n, m) \equiv (4, 8)$.
2. Module `spiking_init_conf(n)` that defines the initial configuration of a SN P system solving the `SAT` problem for any instance with n variables. Furthermore, this module defines the train of spikes which encode the φ formula. This train is defined according to what we stated above.
3. Module `spiking_rules(n,m)` that defines the spiking rules of the SN P system for any instance with n variables and m clauses.
4. Module `neuron_division_rules(n)` that defines the division rules of the SN P system for any instance with n variables.
5. Module `neuron_budding_rules(n)` that defines the budding rules of the SN P system for any instance with n variables.

The source code for the SN P system defined above can be found in the URL: `http://www.p-lingua.org/examples/sat_SNPSystem.pli`. In what follows we will show its most significant part.

```
@model<spiking_psystems>

// Encoding module main()

def main()
{
call spiking_init_conf(4);
call spiking_rules(4,8);
call neuron_division_rules(4);
call neuron_budding_rules(4);
}

// Encoding module spiking_init_conf()

def spiking_init_conf(n)
{

// Encoding initial membranes
@mu = in, out;
@mu += 0,1,2,3;
@mu += d{i} : 0<=i<=n;
@mu += Cx{i} : 1<=i<=n;
```

```
@mu += Cx{i,0} : 1<=i<=n;
@mu += Cx{i,1} : 1<=i<=n;

// Encoding initial membrane spikes
@ms(d{0}) = a;
@ms(0) = a;
@ms(2) = a;
@ms(3) = a;
@ms(d{1}) = a*6;

// Encoding initial synapse graph (also updating synapse dictionary)

@marcs  = (d{i},d{i+1}):0<=i<=n-1;
@marcs += (d{n},d{1});
@marcs += (in,Cx{i}):1<=i<=n;
@marcs += (d{i},Cx{i}):1<=i<=n;
@marcs += (Cx{i},Cx{i,0}):1<=i<=n;
@marcs += (Cx{i},Cx{i,1}):1<=i<=n;
@marcs += ({i+1},{i}):0<=i<=2;
@marcs += (1,2);
@marcs += (0,out);

// Encoding additional synapse dictionary updating

@mdict = (Cx{i,1},t{i}):1<=i<=n;
@mdict+= (Cx{i,0},f{i}):1<=i<=n;

// Encoding input neuron

@min = in;

// Encoding input formula spike train

// Encoding first clause

@minst = (9,1);
@minst+= (10,1);
@minst+= (11,2);
@minst+= (12,1);

...

// Encoding last clause

@minst+= (37,2);
```

```
@minst+= (38,1);
@minst+= (39,1);
@minst+= (40,1);

// Encoding output neuron

@mout = out;
}
// Encoding module spiking_rules()

def spiking_rules(n,m)
{
[a --> a]'in;
[a*2 --> a*2]'in;
[a --> a]'d{0} :: 2*n+n*m;
[a*4 --> a*4]'d{i} : 1<=i<=n;
[a*5 --> #]'d{1};
[a*6 --> a*4]'d{1} :: 2*n+1;
[a --> #]'Cx{i} : 1<=i<=n;
[a*2 --> #]'Cx{i} : 1<=i<=n;
[a*4 --> #]'Cx{i} : 1<=i<=n;
[a*5 --> a*5]'Cx{i} :: n-i : 1<=i<=n;
[a*6 --> a*6]'Cx{i} :: n-i : 1<=i<=n;
[a*5 --> a*4]'Cx{i,1} : 1<=i<=n;
[a*6 --> #]'Cx{i,1} : 1<=i<=n;
[a*5 --> #]'Cx{i,0} : 1<=i<=n;
[a*6 --> a*4]'Cx{i,0} : 1<=i<=n;
[a --> a]'t{i} "(a{4})+" : 1<=i<=n;
[a --> a]'f{i} "(a{4})+" : 1<=i<=n;
[a*(4*k-1) --> #]'t{i} : 1<=k<=n,1<=i<=n;
[a*(4*k-1) --> #]'f{i} : 1<=k<=n,1<=i<=n;
[a*m --> a*2]'cl;
[a --> a]'out "(a{2})+";
[a --> a]'{i} : 1<=i<=2;
[a*2 --> #]'2;
[a --> a]'3 :: 2*n-1;
}
// Encoding module neuron_division_rules()

def neuron_division_rules(n)
{
[]'0 --> []'t{1} || []'f{1} "a";
[]'t{i} --> []'t{i+1} || []'f{i+1} "a" : 1<=i<=n-1;
[]'f{i} --> []'t{i+1} || []'f{i+1} "a" : 1<=i<=n-1;
}
```

```
// Encoding module neuron_budding_rules()

def neuron_budding_rules(n)
{
[]'t{n} --> []'t{n} / []'cl "a";
[]'f{n} --> []'f{n} / []'cl "a";
}
```

5.3 Generalized P-Lingua Source Code for the Family of SN P Systems Solving SAT

Code description. This section describes and shows the P–Lingua source code that defines a family of SN P systems including parameters to allow the definition of different particular scenarios, problems, initial conditions without changing the code. Unlike the previous example, we define the `SAT` problem for any `CNF` formula φ.

Thus we are resolving `SAT(n,m)` with n and m depending on the values introduced by the user in MeCoSim. We can now define any SN P system of the family without changing the P-Lingua code, only by entering different initial data in the graphical user interface of MeCoSim.

The structure of the source code does not change from the previous version. The main difference is the presence of parameters instead of particular values for introducing the number of variables (`n`), the number of clauses (`m`) and the description of the formula in the form of an input spike train (given by the values of the parameters `val(i,j)`. The source code for the SN P system defined above can be found in the URL: `http://www.p-lingua.org/examples/sat_SNPSystem_2.pli`. In what follows we will show the changes introduced respect to the ad-hoc versions.

```
def main()
{
call spiking_init_conf(n,m);
call spiking_rules(n,m);
call neuron_division_rules(n);
call neuron_budding_rules(n);
}

def spiking_init_conf(n,m)
{
...
// Encoding input formula spike train.
// It depends on the parameter values of m, n and val{i,j}

@minst+= ((2*n+j)+(n*(i-1)),val{i,j}):1<=i<=m, 1<=j<=n;
...
}
```

Integration with MeCoSim. This section enumerates the main aspects of the integration of the previous P-Lingua code with MeCoSim by means of a set of captures.

1. Configuration file
   First of all, the MeCoSim configuration file to build the visual simulator adapted to the family of SN P systems solving SAT is filled. The simulation parameters tab to generate the parameters used in the P-Lingua code is shown below.

Param Name	Param Value	Index 1	Index 2
n	<1,1,1>		
m	<1,1,2>		
val	<2,1,2>	[1..m]	[1..n]

Fig. 1. MeCoSim Config File. Simulation Params

2. Input data
   The end user is provided with an interface to enter the needed input data to instantiate the initial SN P system to compute the satisfiability for a given formula.

   – General parameters (number of variables and clauses)

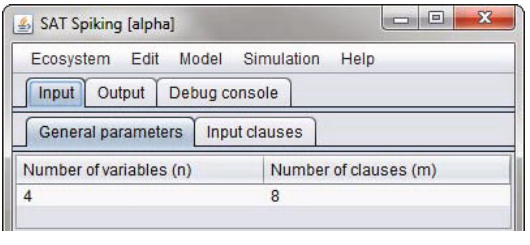

Fig. 2. MeCoSim GUI. General parameters tab

   – Input clauses

   When the user starts the simulation, the configured parameters for P-Lingua are instantiated from the input tables of MeCoSim. Then the computation runs until a halting configuration is reached. Then the output tables and charts are shown.

3. Input spikes train
   A graphical output has been defined to visualize the input spikes train. As mentioned before, these spikes comes into the system in certain steps and quantities as we see in the chart.

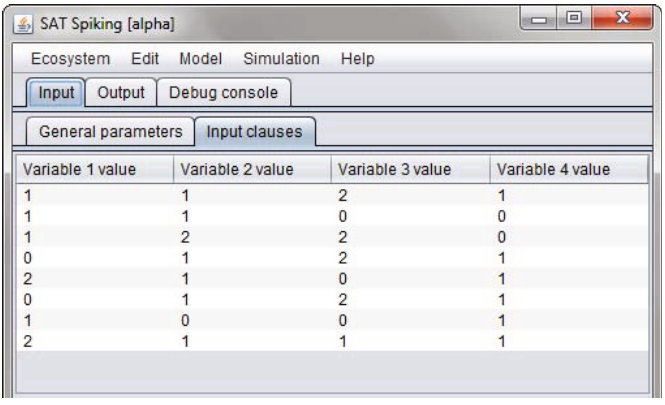

Fig. 3. MeCoSim GUI. Input clauses tab.

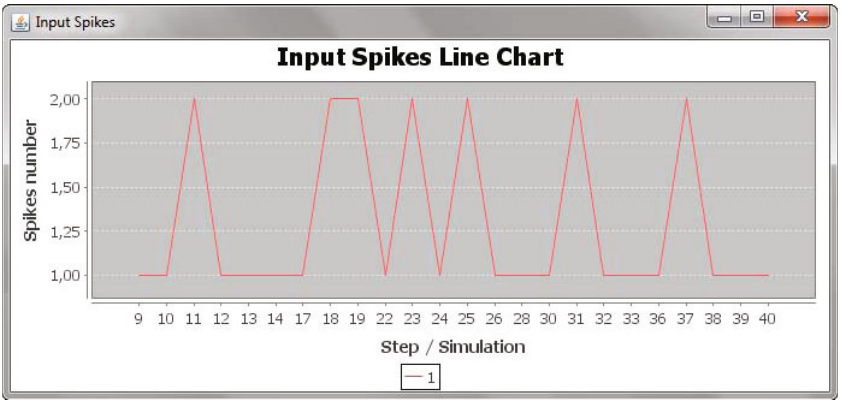

Fig. 4. MeCoSim GUI. Input spikes train chart.

4. Output Results
   Once the simulation halts, the result is shown in the form of an output table and a chart, denoting "Yes" if the formula is satisfiable and "No" otherwise.

   – Satisfiability result table
   – Satisfiability chart

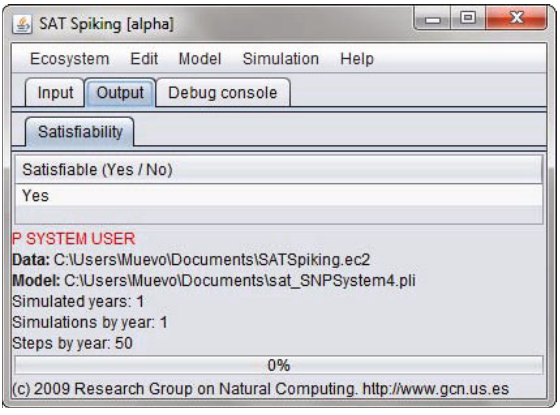

Fig. 5. MeCoSim GUI. Satisfiability table.

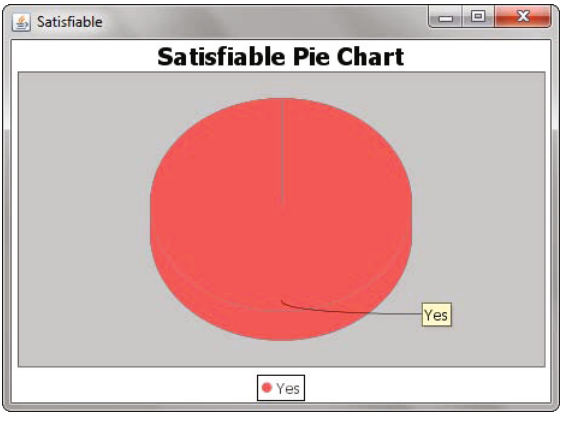

Fig. 6. MeCoSim GUI. Satisfiability chart.

5.4 Simulation Results

The *pLinguaCore* Java library includes a command–line interface in order to parse P–Lingua input files and simulate the defined P systems. The SN P system defined above can be simulated by writing the next command[1] in a system console:

```
java -jar plinguacore.jar plingua_sim -pli sat_SNPSystem.pli -o
                                output.txt
```

A complete explanation of commands for *pLinguaCore* can be found in [7]. In this case, the P–Lingua input file is called `sat_SNPSystem.pli` and it contains

[1] A Java runtime environment 1.6.0 or better must be installed. It can be downloaded from `http://www.java.com`

the source code referenced before. The file `output.txt` is a text file where information about the parser process and the generated computation is stored:

1. Initial cells
2. Initial multisets
3. Rules set
4. For each configuration:
   (a) Multiset of objects in the environment
   (b) Multiset of objects for each cell
   (c) Rules selected to be executed in the next step.

The simulator runs until reaching a halting configuration, where no rule can be selected to be executed in the next step. At this stage, the environment contains either a single object a or no object at all.

For the P system defined in the file `sat_SNPSystem.pli`, an object a is sent to the environment after 46 steps of computation and it halts. That is, it is an *accepting* computation.

5.5 Performance

In order to exemplify the simulation algorithm performance, a few execution examples are presented. All of them are referred to the family of SN P System solving SAT shown above, concretely `3-SAT` instances. The execution environment is a Dual Core AMD Athlon II X2 250 3 GHz Speed 3 GB RAM computer running Windows XP Service Pack 3.

Table 1. Performance results

variables (n)	clauses (m)	execution time
3	3	0.734 s.
3	4	0.938 s.
3	5	0.969 s.
4	4	1.578 s.
4	5	1.937 s.
5	5	4.016 s.

6 Conclusions and Future Work

In this paper we have presented a new release of P-Lingua, that significantly extends the previous version by incorporating the ability to work with Spiking Neural P systems (SN P systems). Besides, a new simulation algorithm has been designed and implemented, taking into account features of SN P systems such as neuron division and budding. This new simulator has been included into the library *pLinguaCore*, and it has been checked by simulating a family of SN

P systems taken from the literature, for solving the well-known **NP**-complete problem `SAT`. A brief description of the solution to this problem using SN P systems has been included in the paper, along with the corresponding P-Lingua source code.

In addition, an adaptation of the P-Lingua code to integrate with MeCoSim, providing the end user of the new simulator the capability for entering different input formula and visualizing the results of the computation in a graphical and structured way.

A possible course of future work is to include weights and thresholds, as described in Wang et al. [19], in P-Lingua and pLinguaCore simulator for SN P systems. The referred paper propose using weighted synapses, potentials in neurons, and rules which handle these potentials under the control of given firing thresholds.

Finally, we are working along with other research groups to join forces to combine the expressive richness and flexibility of P-Lingua and MeCoSim with the efficiency of parallel simulators based on CUDA [1].

Acknowledgements. The authors acknowledge the support of the project TIN2009–13192 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds, and the support of the Project of Excellence with *Investigador de Reconocida Valía* of the Junta de Andalucía, grant P08-TIC-04200.

References

1. Cabarle, F., Adorna, H., Martínez-del-Amor, M.A.: Spiking neural P system without delay simulator implementation using GPGPUs. In: Eleventh Philippine Computing Science Congress, pp. 35–43 (2011)
2. Chen, H., Freund, R., Ionescu, M., Păun, G., Pérez-Jiménez, M.J.: On string languages generated by spiking neural P systems. Fundamenta Informaticae 75, 141–162 (2007)
3. Chen, H., Ionescu, M., Ishdorj, T.-O.: On the efficiency of spiking neural P systems. In: Proceedings of the 8th International Conference on Electronics, Information, and Communication, pp. 49–52 (2006)
4. Chen, H., Ionescu, M., Ishdorj, T.-O., Păun, A., Păun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with extended rules: universality and languages. Natural Computing 7(2), 147–166 (2008)
5. Díaz-Pernil, D., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A P-Lingua Programming Environment for Membrane Computing. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) WMC 2008. LNCS, vol. 5391, pp. 187–203. Springer, Heidelberg (2009)
6. Galli, R., Gritti, A., Bonfanti, L., Vescovi, A.L.: Neural stem cells: an overview. Circulation Research 92, 598–608 (2003)
7. García-Quismondo, M., Gutiérrez-Escudero, R., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Riscos-Núñez, A.: An Overview of P-Lingua 2.0. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) WMC 2009. LNCS, vol. 5957, pp. 264–288. Springer, Heidelberg (2010)

8. Ibarra, O.H., Leporati, A., Păun, A., Woodworth, S.: Spiking Neural P Systems. In: Paun, G., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, ch. 13, pp. 337–362. Oxford University Press, Oxford (2009)
9. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundamenta Informaticae 71(2-3), 279–308 (2006)
10. Ishdorj, T.-O., Leporati, A.: Uniform solutions to SAT and 3-SAT by spiking neural P systems with pre-computed resources. Natural Computing 7(4), 519–534 (2008)
11. Leporati, A., Gutiérrez-Naranjo, M.A.: Solving Subset Sum by spiking neural P systems with pre-computed resources. Fundamenta Informaticae 87(1), 61–77 (2008)
12. Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking Neural P systems with neuron division and budding. In: Proceedings of the Seventh Brainstorming Week on Membrane Computing, vol. 2, pp. 151–168 (2009)
13. Păun, G.: Membrane Computing. An Introduction. Springer, Berlin (2002)
14. Păun, G., Pérez-Jiménez, M.J., Rozenberg, G.: Spike trains in spiking neural P systems. International Journal of Foundations of Computer Science 17(4), 975–1002 (2006)
15. Păun, A., Păun, G.: Small universal spiking neural P systems. BioSystems 90(1), 48–60 (2007)
16. Pérez-Hurtado, I., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Colomer, M.A., Riscos-Núñez, A.: MeCoSim: A general purpose software tool for simulating biological phenomena by means of P Systems. In: IEEE Fifth International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA 2010), vol. 1, pp. 637–643 (2010)
17. Rozenberg, G.: DNA processing in ciliates. The wonders of DNA computing in vivo. In: Unconventional models of Computation (UMC 2K), pp. 116–118 (2010)
18. Wang, J., Hoogeboom, H.-J., Pan, L.: Spiking Neural P Systems with Neuron Division. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) CMC 2010. LNCS, vol. 6501, pp. 361–376. Springer, Heidelberg (2010)
19. Wang, J., Hoogeboom, H.-J., Pan, L., Păun, G.: Spiking neural P systems with weights and thresholds. In: Proceedings of Tenth Workshop on Membrane Computing, pp. 514–533 (2009)
20. Java's regular expressions specification, `http://download.oracle.com/javase/1.4.2/docs/api/java/util/regex/Pattern.html`
21. The Sun's Java Web Site, `http://www.oracle.com/us/technologies/java/index.html`
22. The P–Lingua Web Site, `http://www.p-lingua.org/`
23. The SATLIB library, `http://www.cs.ubc.ca/~hoos/SATLIB/index-ubc.html`