Extending SNP Systems Asynchronous Simulation Modes in P-Lingua

Luis F. Macías-Ramos[1], Tao Song[2], Linqiang Pan[2,⋆], Mario J. Pérez-Jiménez[1]

[1] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla, Spain
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
`lfmaciasr@us.es, marper@us.es`

[2] Key Laboratory of Image Information Processing and Intelligent Control
School of Automation
Huazhong University of Science and Technology
Wuhan 430074, Hubei, China
`taosong@hust.edu.cn, lqpan@mail.hust.edu.cn`

Summary. Spiking neural P systems (SN P systems for short) is a developing field within the P systems world. Inspired by the neurophysiological structure of the brain, these systems have been subjected to many extensions in recent years, many of them intended to "somewhat" incorporate more and more features inspired by the functioning of the living neural cells. Although when first introduced in [8] SN P systems were considered to work in synchronous mode, it became clear that considering non-synchronized systems would be rather natural both from both from a mathematical and neuro-biological point of view. Asynchronous variants of these systems were introduced in [4], setting up a scenario where even if a neuron had enabled rules ready to fire, such rules non-deterministically could be not applied. Once new theoretical variants are defined, providing simulation software tools enables experimental study and validation of the proposed models. One more than promising developing branch comprises the use of parallel architectures, concretely GPUs, that provide efficient implementations [1, 2, 7, 3]. One drawback of this approach, due to the inherent constraints of the GPUs programming model, is a relatively long development cycle to extend existing variants. At the expense of sacrificing efficiency for expressivity, other alternatives involving sequential approaches can be considered. Within this trend, P–Lingua [5, 6, 15] offers the high flexibility of the Java programming language as well as a general acceptance within the Membrane computing community. P–Lingua affords a standard language for the definition of P systems. Part of the same software project, *pLinguaCore* library provides particular implementations of parsers and simulators for the models specified in P–Lingua. Support for simulating SN P systems in P–Lingua was introduced in [9]. In that version all (synchronous and asynchronous) "working modes" considered in [14] were implemented. Since then, new asynchronous variants have appeared. In this paper we present a brand

⋆ Corresponding author.

new extension of P–Lingua related to asynchronous SN P systems, in order to incorporate simulation capabilities for limited asynchronous SN P systems, introduced in [12], and asynchronous SN P systems with local introduced respectively in [16].

1 Introduction

SN P systems were introduced in [8] 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.

An SN P system consists of a set of neurons placed as nodes of a directed graph (called the *synapse graph*). Each neuron contains a number of copies of a single object type, the *spike*. Rules are assigned to neurons to control the way information flows between them, i.e. rules assigned to a neuron allow it to send spikes to its neighbouring neurons. SN P systems usually work in a synchronous mode, where a global clock is assumed. In each time unit, for each neuron, only one of the applicable rules is non-deterministically selected to be executed. Execution of rules takes place in parallel amongst all neurons of the system. Nevertheless both from a mathematical and neurological point of view, it is rather natural to consider non-synchronized systems, where the use of rules is not obligatory. If a neuron has an enabled rule in a given time unit, the neuron chooses non-deterministically whether to fire (or not) the rule. Of course, new spikes can come into the neuron when the rule is not fired, rendering it non-applicable. If the rule is still applicable through successive time instants, it can fire at any time, independently of how much time has passed since it first became applicable. Asynchronous SN P systems (ASNPS for short) were introduced in [4] and, as described in [14], the validity of a computation in these systems can be restricted by specifying pre-defined values over the number of spikes for the neurons in the system at halting time. If the computation is considered invalid the output of the system is discarded.

In the systems mentioned above any neuron works asynchronously, in a independent way with respect to the others, such that there is no restriction with respect to the number of successive time instants that the neuron can hold firing of an enabled rule. These two characteristics of the "asynchronous mode" can be overridden. In [12], limited asynchronous SN P systems (LASNPS for short) are introduced. In this variant, a global bound $b \geq 2$ (equal for all neurons) is defined and determines the maximum number of computation steps that a neuron can choose not to fire an enabled rule. On the other hand, in [16] asynchronous SN P systems with local synchronization (ASNPSLS for short) are introduced. In this variant, independence amongst neurons is dropped. Local synchronization enables pre-defining a collection of local synchronizing sets contained in the power set of neurons of the system. When a neuron fires (after deciding not to fire an unbounded number of steps) all the neurons placed in the same local synchronizing set fire immediately.

There exists biological motivation for considering these variants, thus also making them a suitable target to be implemented within a simulation framework like P–Lingua. With respect to LASNPS, in a biological system, if a long enough time interval is given, an enabled chemical reaction will conclude within the given time interval. So it is natural to impose a bound on the time interval in which a spiking rule remains unused. With respect to ASNPSLS, in a biological neural system, motifs with 4-5 neurons and communities with 12-15 neurons, associated with some specific functioning are rather common. Neurons from the same motif or community will work synchronously to cooperate with each other. That is, in a biological system, neurons work globally in an asynchronous way, but synchronously at a local level. Said level is represented by the local synchronizing sets.

In this paper a new extension of SN P systems simulator included in the P–Lingua framework is presented which enables simulation for LASNPS and ASNPSLS. The paper is structured as follows. Section 2 is devoted to introducing background concepts: specifications of SN P systems working in synchronous and asynchronous (normal, limited and local) modes are introduced in an informal way. Section 3 covers the P–Lingua syntax for the new introduced modes. Section 4 presents some P–Lingua files which exemplifies how to define the different models in P–Lingua specification language. Finally, Section 5 shows the corresponding simulation algorithms for the aforementioned models. Section 6 covers conclusions and future work.

2 Preliminaries

In this section we introduce, in an informal way, SN P systems model in their original synchronous form and, subsequently, asynchronous extensions corresponding to limited asynchronous SN P systems, introduced in [12], and asynchronous SN P systems with local synchronization, introduced in [16].

2.1 Spiking neural P systems

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, cells (also called *neurons*) are placed in the nodes of a directed graph, called the *synapse graph.* Contents of each neuron consist of a number of copies of a single object type, called the *spike.* Every neuron may also contain a number of *firing rules* and *forgetting rules.* Firing rules allow a neuron to send information to other neurons in the form of electrical impulses, *spikes*, which are accumulated at the target neuron, consuming some of their own spikes (in a quantity at least equal to the number of spikes fired). Forgetting rules imply only consuming spikes, while no one is sent to the neighbouring neurons.

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 can 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 respect to each other. Let us notice 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. Also asynchronous scenarios can be considered as shown in [14] and discussed below.

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). The lapse of time required for the rule to be fired is called the "delay" of the rule, which can be any natural number $d \geq 0$. Only firing rules can have a non-negative delay, while forgetting rules have always a delay zero. Let us notice that when $d = 0$, the neuron immediately becomes "open" (active) after firing, being able to send and receive spikes again, thus never being "really closed".

The *configuration* of the system is described by its topological structure (which is constant along computations when not considering division or budding rules) and the number of spikes associated with each neuron. Using the rules as described above, it is possible to 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.

Further reading about this model, along with a formal specification, can be found in [8].

2.2 Asynchronous SN P systems

In asynchronous SN P systems even if a neuron has a rule enabled in a given time unit, this rule is not obligatorily used. The neuron may choose to remain unfired, maybe receiving spikes from the neighbouring neurons. The unused rule may be used later, as long as it stays enabled, without any restriction on the interval during which it has remained unused. If the new spikes made the rule non-applicable, then the computation continues in the new circumstances (maybe other rules are enabled now).

Further reading about these systems, along with a discussion on their power can be found in [4].

2.3 Limited asynchronous SN P systems

In asynchronous SN P systems an enabled rule not used at a certain instant may be used later, as long as it stays enabled, without any restriction on the interval during which it has remained unused. Nevertheless, from the biological point of view it is convenient to consider a boundary on the number of time units that such rule remains unfired, since in nature given a long enough time interval, an enabled chemical reaction will conclude within such interval. Following this, in limited asynchronous SN P systems, a single global upper bound $b \geq 2$ (equal for all neurons) is defined on time intervals. If a rule in neuron σ_i is enabled at step t and neuron σ_i receives no spike from step t to step $t+b-2$, then this rule can and must be applied at a step in the next time interval b (that is, at a non-deterministically chosen step from t to $t+b-1$). If the enabled rule in neuron σ_i is not applied, and neuron σ_i receives new spikes, making the rule non-applicable, then computation continues in the new circumstance (maybe other rules are enabled now).

Further reading about these systems, along with a discussion on their universality can be found in [12].

2.4 Asynchronous SN P systems with local synchronization

In asynchronous SN P systems an unused rule may be used later, without any restriction with respect to the functioning (also asynchronous) of the other neurons. Nevertheless, from the biological point of view it is convenient to consider interrelation between neurons in terms of synchronicity. In a biological neural system, small groups of neurons, associated with some specific functioning, are rather common. Neurons within these communities work synchronously to cooperate with each other, while globally working in an asynchronous way with respect to "unrelated" neurons in the system. To model this behaviour in neural-like systems, asynchronous SN P systems with local synchronization are introduced. In these systems, a family of sets (called *ls-sets*) of locally synchronous neurons $Loc = \{loc_1, loc_2, \ldots, loc_l\} \subseteq \mathcal{P}(\{\sigma_1, \sigma_2, \ldots, \sigma_m\})$ is defined ($\mathcal{P}(\{\sigma_1, \sigma_2, \ldots, \sigma_m\})$ being the power set of $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$).

Given neurons in the same ls-set loc_j, if one of these neurons fires, then all neurons in loc_j that have enabled rules should fire. Of course, it is possible that all neurons from loc_j remain unfired even if they have enabled rules. That is, all neurons from loc_j may remain still, or all neurons from loc_j with enabled rules fire at a same step (of course, neurons without enabled rules cannot fire). Hence, neurons work asynchronously at the global level, while working synchronously within each ls-set.

Further reading about these systems, along with a discussion on their universality can be found in [16].

3 P–Lingua Syntax for LASNPS and ASNPSLS

In [6], a Java library called *pLinguaCore* was presented under GPL license. The library provides parsers to handle input files, built–in simulators to generate P System computations and it is able to export several output file formats that represent P systems. *pLinguaCore* is not a closed product because developers with knowledge of Java can add new components to the library, thus extending it. Milestone releases can be downloaded from `http://www.p-lingua.org` while releases containing latest developments can be found within the distribution of MeCoSim (`http://www.p-lingua.org/mecosim/`). At the time of this writing, the extensions related to the present paper have not been included in a milestone release, so interested readers may refer to the MeCoSim distribution.

This section introduces several recently developed P–Lingua simulators for SN P systems. Support for SN P systems in P–Lingua was introduced in [9], covering the basic model along with neuron division and budding rules as well as asynchronous mode, as originally introduced. Following this, in [10], partial simulation of SN P systems with functional astrocytes (also defined in [10]) was introduced. Finally, simulators for SN P systems with "hybrid" (excitatory and inhibitory) astrocytes and SN P systems with anti-spikes were presented in [11].

In what follows, P–Lingua syntax for defining both LASNPS and ASNPLS is shown. P–Lingua syntax for the other SN P system variants is not covered here, but can be found in the cited papers.

3.1 P–Lingua syntax for limited asynchronous SN P systems

In LASNPS, a global upper bound $b \geq 2$ is defined for all rules. Consequently, a new instruction has been included into P–Lingua to define such upper bound, extending the existing model specification framework for Spiking Neural P systems. Thus, that instruction can be used only when the source P–Lingua files defining the corresponding models begin with the following sentence:

```
@model<spiking_psystems>
```

while also requiring the right asynchronous mode to be set to with the following sentence:

```
@masynch = 3;
```

The instruction to define the global upper bound is:

```
@mboundall = b;
```

where:

- b is the global upper bound, with $b \geq 2$.

3.2 P–Lingua syntax for asynchronous SN P systems with local synchronization

In ASNPLS, a local synchronizing set is defined, consisting of a collection of sets (called ls-sets) that determines which neurons should fire synchronously. Consequently, a new instruction has been included into P–Lingua to define such set, extending the existing model specification framework for Spiking Neural P systems. Thus, that instruction can be used only when the source P–Lingua files defining the corresponding models begin with the following sentence:

```
@model<spiking_psystems>
```

while also requiring the right asynchronous mode to be set to with the following sentence:

```
@masynch = 4;
```

The instruction to define the local synchronizing set is:

```
@mlocset = {ls-1, ls-2, ..., ls-h, ..., ls-m};
```

where:

- ls-h $= \{\sigma_{h,1}, \ldots, \sigma_{h,u_h}\}$ is each one of the ls-sets containing a non-empty collection of membrane labels.

4 Examples

This section is devoted to consider examples dealing with both LASNPS and ASNPSLS. Two sets of examples are presented, one per variant. For each set, a formal specification is presented, followed by a link to the corresponding P–Lingua code. To conclude, an analysis of the functioning of the systems is shown, along with statistical data referred to the output of the systems after several simulations within MeCoSim are performed.

4.1 Asynchronous SN P systems with local synchronization

Consider an asynchronous SN P system with local synchronization Π shown in Figure 1.

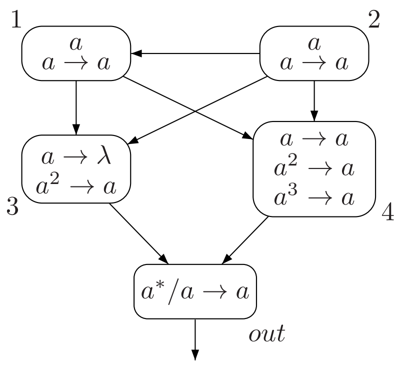

Fig. 1. An example of an asynchronous SN P system with local synchronization Π

The system Π consists of five neurons with labels 1, 2, 3, 4 and *out*. Initially, neurons σ_1 and σ_2 have one spike inside, and other neurons contain no spike. The formal definition of system Π is as follows:

$$\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_3, \sigma_4, \sigma_{out}, Loc, syn), \text{ where}$$

- $\sigma_1 = (1, R_1)$ with $R_1 = \{a \to a\}$;
- $\sigma_2 = (1, R_2)$ with $R_2 = \{a \to a\}$;
- $\sigma_3 = (0, R_3)$ with $R_3 = \{a \to \lambda, a^2 \to a\}$;
- $\sigma_4 = (0, R_3)$ with $R_4 = \{a \to a, a^2 \to a, a^3 \to a\}$;
- $\sigma_{out} = (0, R_{out})$ with $R_{out} = \{a^*/a \to a\}$;
- Loc is the family of sets of locally synchronous neurons;
- $syn = \{(1,3), (1,4), (2,1), (2,3), (2,4), (3, out), (4, out)\}$;
- *out* indicates the output neuron.

To complete the definition of the system, specifying Loc is required. Four cases are considered:

a) $Loc = \emptyset$
b) $Loc = \{\{\sigma_1, \sigma_2\}\}$
c) $Loc = \{\{\sigma_1, \sigma_3\}\}$
d) $Loc = \{\{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4\}\}$

The P–Lingua code corresponding to this system can be found at: `http://www.p-lingua.org/examples/SNPSLocalSynch.pli`.

This code is parameterised, allowing execution of each one of the four Π system variants depending on parameter c.

Next, we are going to analyse functioning of system Π for each one of the aforementioned cases.

Case a) $Loc = \emptyset$.

In this case, neurons can fire at any time when having enable rules. Output of the system is $\{1, 2, 3, 4\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

spikes	ratio
1	39%
2	20%
3	36%
4	5%

Table 1. Ratio of output spikes for case a)

Case b) $Loc = \{\{\sigma_1, \sigma_2\}\}$.

In this case, neurons σ_1, σ_2 fire at the same moment due to local synchronization. After any of the former neurons fires, neuron σ_1 contains 1 spike, while neurons σ_3, σ_4 contain 2 spikes each. From this point, we have the following cases:

1. Neuron σ_1 fires before neurons σ_3, σ_4. After this, neuron σ_3 contains 3 spikes, so cannot fire, while neuron σ_4 also containing 3 spikes will eventually send out 1 spike. Thus, the output of the system in this case is $\{1\}$.
2. Neuron σ_3 fires before neurons σ_1, σ_4. Spike from neuron σ_3 reaches σ_{out}. From here, any spike passing through σ_3 will be lost, with either 1 or 2 spikes getting to σ_{out} from neuron σ_4. Consequently, the output of the system will be $\{2, 3\}$.
3. Neuron σ_4 fires before neurons σ_1, σ_3. Thus, 1 spike reaches neuron σ_{out} from σ_4. If σ_1 fires before σ_3, this neuron becomes blocked and cannot fire any more, with 1 spike more coming to neuron σ_{out} from σ_4. In any other case, σ_3 sends another spike to σ_{out} and loses the next spike that receives, while σ_4 getting 1 spike that will be sent to σ_{out} eventually. Consequently, the output of the system will be $\{2, 3\}$.
4. Neurons σ_1, σ_3 fire together before σ_4. In this case, σ_3 sends 1 spike to σ_{out}, while the next spike coming from σ_1 will be lost. Finally, σ_4 consumes 3 spikes and sends 1 spike to σ_{out}. Consequently, the output of the system will be $\{2\}$.
5. Neurons σ_1, σ_4 fire together before σ_3. In this case, 1 spike reaches σ_{out} from σ_4, while σ_3, containing 3 spikes, gets blocked. Finally, σ_4 sends 1 spike to σ_{out}. Consequently, the output of the system will be $\{2\}$.
6. Neurons σ_3, σ_4 fire together before σ_1. In this case, 2 spikes reach σ_{out} from σ_3, σ_4 each. Following this σ_1 fires 1 spike that reaches σ_3, σ_4. Spike in σ_3 will be lost while the one in σ_4 will reach σ_{out}. Consequently, the output of the system will be $\{3\}$.
7. Neurons $\sigma_1, \sigma_3, \sigma_4$ fire together. This case is similar to the previous one. Consequently, the output of the system will be $\{3\}$.

As a result of all of this, output of the system in this case is $\{1, 2, 3\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

spikes	ratio
1	17%
2	38%
3	25%
4	0%

Table 2. Ratio of output spikes for case b)

Case c) $Loc = \{\{\sigma_1, \sigma_3\}\}$.

In this scenario, we can consider the following cases:

a) Neuron σ_2 fires before σ_1. In this case, after neuron σ_2 fires, σ_1 contains 2 spikes, being unable to continue working, neuron σ_3 contains 1 spike, that will be lost, and σ_4 contains 1 spike, that will eventually reach σ_{out}. Thus, in this case the output of the system is $\{1\}$.

b) Neurons σ_1, σ_2 fire at the same time. In this case, after the firing, σ_1 contains 1 spike, while σ_3, σ_4 contain 2 spikes each. From here, we have the following cases:
   1. Neurons σ_1, σ_3 fire before σ_4. In this case, neuron σ_{out} gets 1 spike from σ_3, while neuron σ_1 sends 1 spike to σ_3, that will be lost, and σ_4. Having 3 spikes, σ_4 sends 1 spike to σ_{out}. Consequently, the output of the system will be $\{2\}$.
   2. Neurons σ_1, σ_3 fire at the same time as σ_4. This case is similar to the previous one, with σ_{out} getting 2 spikes from σ_4. Consequently, the output of the system will be $\{3\}$.
   3. Neurons σ_1, σ_3 fire after σ_4. This case is similar to the previous one. Consequently, the output of the system will be $\{3\}$.

c) Neuron σ_1 fires before σ_2. In this case, after neuron σ_1 fires, $\sigma_2, \sigma_3, \sigma_4$ contain only 1 spike each. We have the following cases from here:
   1. Neuron σ_2 fires before any other neuron. In this case, neuron σ_1 gets 1 spike, while neurons σ_3, σ_4 contain 2 spikes each. As we showed before, the output of the system will be $\{2, 3\}$.
   2. Neuron σ_3 fires before any other neuron. This spike is lost. From this point, all the spikes passing through σ_3 will be lost, with σ_4 being able to fire up to 3 times. Consequently, the output of the system will be $\{1, 3\}$.
   3. Neuron σ_4 fires before any other neuron. This spike reaches σ_{out}. At this point, σ_2, σ_3 contain 1 spike each. From here:
      – Neuron σ_2 fires first. At this point, σ_1, σ_4 contain 1 spike each, while σ_3 contains 2 spikes. The following cases are possible:
         * Neurons σ_1, σ_3 fire before σ_4. In this case, σ_3 only sends 1 spike to σ_{out} (others are lost), while σ_4 consuming 2 spikes and sending 1 spike to σ_{out}. Consequently, the output of the system will be $\{3\}$.
         * Neurons σ_1, σ_3 fire at the same time as σ_4. This case is similar to the previous one with σ_4 sending an additional spike to σ_{out}. Consequently, the output of the system will be $\{4\}$.
         * Neurons σ_1, σ_3 fire after σ_4. This case is similar to the previous one. Consequently, the output of the system will be $\{4\}$.
      – Neuron σ_3 fires first, resulting in 1 spike being lost. After neuron σ_2 fires, $\sigma_1, \sigma_3, \sigma_4$ contain 1 spike each. The following cases are possible:
         * Neurons σ_1, σ_3 fire before σ_4. In this case, all spikes involving σ_3 are lost, while σ_4 ends consuming 2 spikes and sending 1 spike to σ_{out}. Consequently, the output of the system will be $\{2\}$.
         * Neurons σ_1, σ_3 fire at the same time as σ_4. This case is similar to the previous one with σ_4 sending an additional spike to σ_{out}. Consequently, the output of the system will be $\{3\}$.
         * Neurons σ_1, σ_3 fire after σ_4. This case is similar to the previous one. Consequently, the output of the system will be $\{3\}$.

  - Neuron σ_2, σ_3 fire at the same time. This case similar to the previous one. Consequently, the output of the system will be $\{2, 3\}$.
4. Neurons σ_2, σ_3 fire together and before σ_4. Spike in σ_3 is lost, while σ_2 sends 1 spike to $\sigma_1, \sigma_3, \sigma_4$, resulting in σ_1, σ_3 containing 1 spike each while σ_4 contains 2 spikes. The spike in σ_3 will be lost in any case. If σ_1, σ_3 fire before σ_4, 2 spikes are stored in σ_4, thus, only 1 spike is sent to σ_{out}. In other case, 2 spikes reach σ_{out}. Consequently, the output of the system will be $\{1, 2\}$.
5. Neurons σ_2, σ_4 fire together and before σ_3. In this case, 1 spike reaches σ_{out} from σ_4, while the spike sent by σ_2 results in σ_1, σ_4 containing 1 spike each while σ_3 contains 2 spikes. This case is analogous to the previous one taking into account that the first rule applied in σ_3 is the firing rule. Consequently, the output of the system will be $\{3, 4\}$.
6. Neurons σ_3, σ_4 fire together and before σ_2. Spike in σ_3 is lost, while spike in σ_4 reaches σ_{out}. After σ_2 fires, $\sigma_1, \sigma_3, \sigma_4$ contain 1 spike each. Spikes passing through σ_3 will be lost and either another 1 or 2 spikes can get to σ_{out} from σ_4. Consequently, the output of the system will be $\{2, 3\}$.
7. Neurons $\sigma_2, \sigma_3, \sigma_4$ fire together. This case is analogous to the previous one. Consequently, the output of the system will be $\{2, 3\}$.

As a result of all of this, output of the system in this case is $\{1, 2, 3, 4\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

spikes	ratio
1	35%
2	29%
3	33%
4	3%

Table 3. Ratio of output spikes for case c)

Case d) $Loc = \{\{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4\}\}$.

Neurons σ_1, σ_2 will fire at the same moment (due to the local-synchronization) sending 2 spikes to neurons σ_3, σ_4 respectively. At that moment, neuron σ_1 receives 1 spike from neuron σ_2. With 2 spikes inside, neurons σ_3, σ_4 will fire at the same moment. If they fire before neuron σ_1, then σ_{out} receives 2 spikes first, and then 1 spike more from σ_4 (the spike in σ_3 is lost), thus receiving in total 3 spikes. The case in which all neurons fire at the same time is similar to the previous one. To conclude, if neuron σ_1 fires before neurons σ_3, σ_4, then neuron σ_3 cannot fire, and σ_4 will consume 3 spikes and send 1 spike to σ_{out}.

Consequently, output of the system is $\{1, 3\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

spikes	ratio
1	15%
2	0%
3	85%
4	0%

Table 4. Ratio of output spikes for case d)

Conclusion
The computation result of Π is $\{1, 2, 3, 4\}$.

4.2 Limited asynchronous SN P systems

Consider a limited asynchronous SN P system Π' shown in Figure 2.

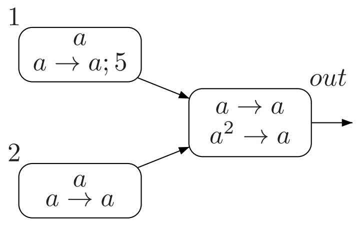

Fig. 2. An example of a limited asynchronous SN P system Π'

The system Π consists of three neurons with labels 1, 2 and *out*. Initially, neurons σ_1 and σ_2 have one spike inside with σ_{out} containing no spike. The formal definition of system Π' is as follows:

$$\Pi = (\{a\}, \sigma_1, \sigma_2, \sigma_{out}, b, syn), \text{ where}$$

- $\sigma_1 = (1, R_1)$ with $R_1 = \{a \to a; 5\}$;
- $\sigma_2 = (1, R_2)$ with $R_2 = \{a \to a\}$;
- $\sigma_{out} = (0, R_{out})$ with $R_{out} = \{a \to a, a^2 \to a\}$;
- b is a single upper bound on time intervals, valid for all rules;
- $syn = \{(1, out), (2, out)\}$;
- *out* indicates the output neuron.

To complete the definition of the system, specifying upper bound $b \geq 2$ is required. Two cases are considered:

a) $b = 2$
a) $b = 4$

The P–Lingua code corresponding to this system can be found at: `http://www.p-lingua.org/examples/SNPSLimited.pli`.

This code is parameterised, allowing execution of each one of the two Π' system variants depending on parameter b.

Next, we are going to analyse functioning of system Π' for each one of the aforementioned cases.

Case a) $b = 2$.

Initially, neurons σ_1, σ_2 have 1 spike inside. Neuron σ_2 will fire no later than step b = 2, sending 1 spike to neuron σ_{out}. With 1 spike inside, σ_{out} will fire before step 2b = 4, sending 1 spike to the environment. Neuron σ_1 can fire at step 1 or 2 (due to the fact that the interval bound b is 2), and sends 1 spike to neuron σ_{out} at step 6 or 7. In this way, neuron σ_{out} will emit 2 spikes into the environment.

Consequently, output of the system is $\{2\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

spikes	ratio
1	0%
2	100%
3	0%
4	0%

Table 5. Ratio of output spikes for case a)

Case b) $b = 4$.

Initially, neurons σ_1, σ_2 have 1 spike and will fire at any step no later than step b = 4. Following this, if neuron σ_1 fires at step 1, it sends 1 spike to neuron σ_{out} at step 6 and so on. On the other hand, if neuron σ_2 fires at step 4, it sends 1 to neuron σ_{out} at such step. In this case, after neuron σ_{out} receives the first spike at step 4, it will fire at any step no later than step 8 (due to the fact that the upper bound b is 4). As a result of all of this, we have to cases:

a) If σ_{out} stays inactive later than step 6, it will accumulate 2 spikes inside, and send 1 spike into the environment.
b) If σ_{out} fires before step 6, it will send 2 spikes to the environment.

In this condition, The system can generate numbers 1 and 2. In a similar way other computations of the system can be checked.

Consequently, output of the system is $\{1, 2\}$.

A table showing the results of the execution of the system after 100 simulations with MeCoSim can be found below.

Therefore, the computation result of Π' is $\{1, 2\}$.

spikes	ratio
1	2%
2	98%
3	0%
4	0%

Table 6. Ratio of output spikes for case b)

5 Simulation Algorithm

In what follows a P–Lingua based simulation algorithm for LASNPS and ASNPSLS is shown in pseudo-code form. This algorithm is a revision from the one included in the foundational paper on simulating SN P systems in P–Lingua [9]. Consequently, it generates one possible computation for a SN P system with an initial configuration C_0 containing n neurons $m_1, \ldots, m_n$. Let us recall that when working with recognizer P systems all computations yield the same answer (confluence).

The simulation algorithm is structured in six stages:

I. Initialization
In this stage the data structures needed to perform the simulation are initialized.

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

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

IV. Execute division and budding rules
In this stage 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 stage execution of spiking rules is performed.

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

The simulation algorithm follows.

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 the void rule)
   - 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).
   - an integer decreasing-only counter $b_i$, that stores the number of steps left for the membrane to decide if its current selected rule can be freely chosen to fire or not (initially zero; if zero then it should fire in this step).
   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 the simulator is working on limited asynchronous mode and each one of the following statements is true
        i. $b_i > 0$
       ii. $r_i$ is not void
      iii. $r_i$ is active
       iv. $r_i$ can be applied over $m_i$
      Then
        i. Decrease the counter $b_i$
       ii. Add $m_i$ to $M_{sel}$
      iii. Go to process the next membrane
   c) 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
   d) Let $S_i \equiv \emptyset$ be the set of possible rules to be executed over $m_i$
   e) 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 $S_i$
   f) If $S_i$ is empty then
        i. Set $b_i$ to zero
       ii. Go to process the next membrane
   g) Select non deterministically a rule $r_k$ from $S_i$
   h) Set $r_k$ as the new selected rule for $m_i$
   i) If $r_k$ is a firing rule, update the counter $d_i$ accordingly
   j) Restart the counter $b_i$ to the global upper bound $b$
   k) Add $m_i$ to $M_{sel}$
   l) 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. Let $Available \equiv \emptyset$ be the set that stores the membranes having a rule selected to
   be executed in the current step
5. Let $toFire \equiv \emptyset$ be the set that stores the membranes having a rule that will be
   fired in the current step
6. Let $toFireAux \equiv \emptyset$ be an auxiliary set
7. Let $locProc \equiv \emptyset$ be the set that stores the membranes that have been processed
   in terms of local synchronizing (that is, each one of their neighbours have been
   marked to fire immediately and have been processed also)
8. 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 $Spiking$
   e) Add $m_i$ to $toFireAux$ if and only if the call to procedure $DecideToFire(m_i, r_i)$
      yields true.
   f) If the simulator is operating in asynchronous mode with local synchronization
      then add $m_i$ to $Available$
9. Add all membranes from $toFireAux$ into $toFire$
10. If the simulator is operating in asynchronous mode with local synchronization then
    for each membrane $m_i$ from $toFireAux$ do
    a) Call the recursive procedure $Process(m_i, Available, locProc, toFire)$
```

Procedure $DecideToFire(m_i, r_i)$

```
1. Let $m_i, r_i$ be input/output arguments declared consistently as specified above
2. If $r_i$ is a budding rule or a division rule, then return the result of the call to
   procedure $DecideAsynch(m_i, r_i)$
3. If $r_i$ is a firing rule or a forgetting rule, then
   a) Let $d$ be the delay associated to rule $r_i$
   b) Let $s$ be the number of steps left for membrane $m_i$ to become open
   c) If $d = 0$ then return the result of the call to procedure $DecideAsynch(m_i, r_i)$
   d) Else
      i. If $d = s$ return the result of the call to procedure $DecideAsynch(m_i, r_i)$
      ii. Else return true
```

Procedure $DecideAsynch(m_i)$

```
1. Let $m_i$ be input/output argument declared consistently as specified above
2. Return true if and only if one of the following statements is true
```

```
a) the simulator is operating in synchronous mode
b) the simulator is operating in "Standard" asynchronous mode and the truth
   value "true" is obtained with a probability equal to 0.5
c) the simulator is operating in limited asynchronous mode and either
   c.1) the counter $b_i$ associated to $m_i$ is equal to zero
   c.2) otherwise the truth value "true" is obtained with a probability equal to 0.5
        and, subsequently, the counter $b_i$ associated to $m_i$ is set to zero
d) the simulator is operating in asynchronous mode with local synchronization
   and the truth value "true" is obtained with a probability equal to 0.5
```

Recursive procedure $Process(m_i, Available, locProc, toFire)$

```
1. Let $m_i, Available, locProc, toFire$ be input/output arguments declared consis-
   tently as specified above
2. If $m_i \in locProc$ then exit
3. Else
   a) Add $m_i$ into $locProc$
   b) If $m_i \in Available$ then
      i. Add $m_i$ into $toFire$
      ii. Let $Affected$ be the set containing all the membranes that should
          immediately fire in case $m_i$ fires
      iii. For each membrane $m_j \in Affected$, do
           A. Call the recursive procedure $Process(m_j, Available, locProc, toFire)$
```

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 $m_i \notin toFire$ 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 $m_i \notin toFire$ 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 $m_i \notin toFire$ then go to process the next membrane
        b) If $m_i$ is closed 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 $[E/a^c \to a^p; d]_i$ 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 $[E/a^c \to \lambda]_i$ 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
```

6 Conclusions and Future Work

In this paper we have shown very recent developments in the field of simulators for SN P systems, concretely P–Lingua based ones. New variants are presented, integrating limited and locally synchronized asynchronous modes. In this sense, a new release of P–Lingua, that extends the previous SN P systems simulator has been developed, incorporating the ability to work with the new implemented models. This new simulator has been included into the library *pLinguaCore* and tested by simulating selected examples provided by experts and referred in Section 4.

At the moment, an extension to incorporate fuzzy reasoning SN P systems is in development. Once this work is done, a desirable feature would be to provide a mechanism for defining arbitrary computable functions, thus fully simulating SNPSFA. Additional elements such as weights might also be incorporated. Also connecting *pLinguaCore* with existing CUDA-based simulators in being considered at present.

Acknowledgements

The authors acknowledge the support of the project TIN2012–37434 of the Ministerio de Ciencia e Innovación of Spain, cofinanced by FEDER funds. T. Song an L. Pan were supported by National Natural Science Foundation of China (61033003, 91130034, and 61320106005).

References

1. Cabarle, F.G., Adorna, H.N., Martínez-Del-Amor, M.A.: Simulating spiking neural P Systems without delays using GPUs. IJNCR 2(2), 19–31 (2011)
2. Cabarle, F.G., Adorna, H.N., Martínez-Del-Amor, M.A., Pérez-Jiménez, M.J.: Spiking neural P System simulations on a high performance GPU platform. Lecture Notes in Computer Science 7017, 99–108 (10/2011 2011), `http://www.springerlink.com/content/f490qnv027884g27/`, algorithms and Architectures for Parallel Processing ICA3PP Workshops, (ADCN 2011)
3. Cabarle, F.G., Adorna, H.N., Martínez-Del-Amor, M.A., Pérez-Jiménez, M.J.: Improving GPU simulations of spiking neural P Systems. Romanian Journal of Information Science and Technology 15, 5–20 (06/2012 2012), `http://www.imt.ro/romjist/Volum15/Number15_1/cuprins15_1.htm`
4. Cavaliere, M., Egecioglu, m., Ibarra, O.H., Ionescu, M., Paun, G., Woodworth, S.: Asynchronous spiking neural p systems: Decidability and undecidability. In: Garzon, M.H., Yan, H. (eds.) DNA. Lecture Notes in Computer Science, vol. 4848, pp. 246–255. Springer (2007), `http://dblp.uni-trier.de/db/conf/dna/dna2007.html#CavaliereEIIPW07`
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.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5391, pp. 187–203. Springer (2008)
6. 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.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5957, pp. 264–288. Springer (2009)
7. Gheorghe, M., Paun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.): Membrane Computing - 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7184. Springer (2012)
8. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P Systems. Fundam. Inform. 71(2-3), 279–308 (2006)
9. Macías-Ramos, L.F., Pérez-Hurtado, I., García-Quismondo, M., Valencia-Cabrera, L., Pérez-Jiménez, M.J., Riscos-Núñez, A.: A P-Lingua based simulator for spiking neural P Systems. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Int. Conf. on Membrane Computing. Lecture Notes in Computer Science, vol. 7184, pp. 257–281. Springer (2011)
10. Macías-Ramos, L.F., Pérez-Jiménez, M.J.: Spiking Neural P Systems with Functional Astrocytes, 12th international conference on Membrane Computing, accepted paper

11. Macías-Ramos, L.F., Pérez-Jiménez, M.J.: On recent developments in p-lingua based simulators for spiking neural p systems. Asian Conference on Membrane Computing pp. 14–29 (10/2012 2012)
12. Pan, L., Wang, J., Hoogeboom, H.J.: Limited asynchronous spiking neural p systems. Fundam. Inf. 110(1-4), 271–293 (Jan 2011), `http://dl.acm.org/citation.cfm?id=2362097.2362116`
13. Păun, G.: Computing with membranes (P Systems): An introduction. In: Current Trends in Theoretical Computer Science, pp. 845–866 (2001)
14. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
15. Research Group on Natural Computing, University of Seville: The P–Lingua website. `http://www.p-lingua.org`
16. Song, T., Pan, L., Păun, G.: Asynchronous spiking neural p systems with local synchronization. Inf. Sci. 219, 197–207 (Jan 2013), `http://dx.doi.org/10.1016/j.ins.2012.07.023`