Simulating FRSN P Systems with Real Numbers in P-Lingua on sequential and CUDA platforms

Luis F. Macías-Ramos, Miguel A. Martínez-del-Amor, and Mario J. Pérez-Jiménez

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

Abstract. Fuzzy Reasoning Spiking Neural P systems (FRSN P systems, for short) is a variant of Spiking Neural P systems incorporating fuzzy logic elements that make it suitable to model fuzzy diagnosis knowledge and reasoning required for fault diagnosis applications. In this sense, several FRSN P system variants have been proposed, dealing with real numbers, trapezoidal numbers, weights, etc. The model incorporating real numbers was the first introduced [13], presenting promising applications in the field of fault diagnosis of electrical systems. For this variant, a matrix-based algorithm was provided which, when executed on parallel computing platforms, fully exploits the model maximally parallel capacities. In this paper we introduce a P-Lingua framework extension to parse and simulate FRSN P systems with real numbers. Two simulators, implementing a variant of the original matrix-based simulation algorithm, are provided: a sequential one (written in Java), intended to run on traditional CPUs, and a parallel one, intended to run on CUDA-enabled devices.

Keywords: Membrane Computing · P systems · Spiking Neural P systems · Fuzzy Reasoning Spiking Neural P systems · Fault diagnosis · Fuzzy knowledge · Fuzzy reasoning · P-Lingua · Java · CUDA

1 Introduction

Membrane computing is a branch of natural computing, which takes inspiration from the structure and functioning of living cells to provide parallel and distributed computational models, called membrane systems or P systems.

P systems were first introduced in [15], and many variants were subsequently developed, which can be divided into three categories: cell-like systems, inspired by the hierarchical membrane structure of eukaryotic cells [15]; tissue-like systems, inspired by the way in which cells organize and communicate within a net-like structure in tissues [8]; and neural-like systems, inspired by the way in which the neurons in the brain exchange information by means of the propagation of spikes [5]. Models belonging to this last variant are collectively called Spiking Neural P systems (SN P systems, for short).

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 connected neurons. Two kinds of rules are considered: firing/spiking rules and forgetting rules. By applying a firing/spiking rule, some spikes are consumed and new spikes are produced. Produced spikes are sent to all neurons linked to the neuron executing the rule. By applying a forgetting rule, spikes are removed from neurons. SN P systems usually work in 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.

SN P systems have become really popular within the Membrane Computing community and extensive work has been conducted to study their properties and produce new variants. For instance, it has been proved that these systems are computational complete (equivalent in power to Turing machines) when considered as number computing devices [5], used as language generators [2,3], or to compute functions [14]. Different kinds of asynchronous "working modes" have been also addressed [12,16,17]. In what concerns to produce new variants of the model, this has involved incorporating new elements such as weights [19], anti-spikes [9], extended rules [16], budding and division rules [10] and astrocytes [1,7,11], among other examples.

In [13] a new SN P systems variant, called FRSN P systems, was introduced, incorporating fuzzy logic elements. The motivation of this variant was to bring together desirable features (understandable, dynamical, synchronized, non-linear, non-deterministic, able to handle incomplete and uncertain information) to model diagnosis knowledge and reasoning in the field of fault diagnosis. To accomplish this, new ingredients were added to extend original SN P systems: three types of neurons (proposition neurons, AND-type and OR-type rule neurons), fuzzy truth values (modelled by means of real numbers) and a new firing mechanism. Also, a matrix-based algorithm was provided, suitable to be executed on parallel computing platforms, and thus able to fully exploit the model maximally parallel capacities. Applications of this new model have been related to fault diagnosis on electrical systems so far [13,21]. Variants have also appeared since the model introduction, for instance dealing with trapezoidal numbers [22,23] and weights [20], with applications to power systems fault diagnosis.

Due to the promising applications of FRSN P systems, it becomes interesting to provide the corresponding simulators, thus favouring research on this model within Membrane Computing community as well as in applied fields. In this paper we introduce support for FRSN P systems with real numbers into `P-Lingua` [4,28] framework. `P-Lingua` consist of a general programming language for P systems called `P-Lingua` itself and a Java [26] based open source library called `pLinguaCore`. In particular, `P-Lingua` language provides a common syntax for specifying P systems variants, with `pLinguaCore` provides both parsers and simulators for such variants. The notable variety of supported models (see [28] for a list of related publications) contributed to make `P-Lingua` widely used among members of the Membrane Computing community, turning its specification language into *a sort of standard*.

Developing FRSN P systems support has involved designing a specific parser (since with respect to `P-Lingua`, FRSN P systems are considered a "separated" variant of SN P systems), a simulation algorithm (which is a variation of the one introduced in [13]) and the corresponding simulators. The provided simulation algorithm is a matrix-based one. As such is susceptible to being executed on parallel platforms, specially intended to work in simultaneously with hundreds to millions of data stored in matrices. In this way, it can take advantage of the corresponding execution speedup. Indeed, GPUs have been successfully used to accelerate well-known linear algebra libraries, such as `MKL BLAS` and `LAPACK`. Specifically, `NVIDIA` GPUs are able to execute scientific applications through `CUDA` [6], harnessing the highly parallel architecture within them (featuring up to 3000 computing cores). In this sense, `CUDA` offers special linear algebra libraries such as `cuBLAS` and `CULA` tools, delivering up to 17$\times$ of speedups for some applications [24]. Therefore, along with a Java sequential simulator, a parallel one has been developed intended to be able to run on the majority of `CUDA`-compatible [24] devices. This last simulator works by means of a `JAVA-CUDA` binding provided by the open source `JCUDA` [27] library, available for Windows, Linux, MacOS and other operating systems.

This paper is structured as follows. Section 2 is devoted to recall the basic ingredients of FRSN P systems with real numbers. In Sect. 3, a `P-Lingua` syntax for such variant is introduced. Section 4 is devoted to simulation aspects: the new matrix-based simulation algorithm is introduced, and invoking the sequential and parallel simulators is discussed. Also, compatibility and performance of the parallel simulator is addressed. Section 5 covers conclusions and future work.

2 Fuzzy Reasoning Spiking Neural P Systems with Real Numbers

In what follows, we recall FRSN P systems with real numbers, which constitute an extension of SN P systems. As new ingredients, three types of neurons (proposition neurons, AND-type and OR-type rule neurons), and elements from the fuzzy logic such as fuzzy truth values are incorporated, as well as a new firing mechanism defined after such fuzzy logic elements. FRSN P systems with real numbers can model and visualize fuzzy production rules in a diagnosis knowledge base due to their graphical nature. Combination of neuron's new firing mechanism and fuzzy logic ensures to automatically accomplish dynamic fuzzy reasoning. FRSN P systems with real numbers can be defined as follows (an extensive description of this model can be found at [13]):

Definition 1. *A FRSN P system Π with real numbers of degree (l, q, n, k), with $l, k \geq 1$, $q \geq 0$ and $n \geq l+q+1$, is a tuple of the form $(A, \sigma_1, \ldots, \sigma_{n+k}, syn, I, O)$, where*

(1) $A=\{a\}$ *is the singleton alphabet (the object a is called spike);*
(2) $\sigma_1, \ldots, \sigma_{n+k}$ *are neurons, of the form* $\sigma_i=(\alpha_i, \tau_i, r_i)$, $1 \leq i \leq n+k$, *where*
  - $(\star)$ $\alpha_i \in [0,1]$ *and it is called the (potential) value of spike contained in neuron* σ_i *(also called pulse value);*
  - $(\star)$ $\tau_i \in [0,1]$ *is the truth value associated with neuron* σ_i*;*
  - $(\star)$ r_i *is a firing/spiking rule contained in neuron* σ_i*, of the form* $E/a^{\alpha} \rightarrow a^{\beta}$*, where* $\alpha, \beta \in [0,1]$.

(3) $syn \subseteq \{1, \ldots, n+k\} \times \{1, \ldots, n+k\}$ *with* $i \neq j$ *for all* $(i,j) \in syn$, $1 \leq i, j \leq n+k$ (synapses *between neurons);*
(4) $I = \{\sigma_1, \ldots, \sigma_l\}$ *is the set of the input neurons that verifies the following: for each* $\sigma \in I$, $indegree(\sigma) = 0$.
(5) $O = \{\sigma_{l+q+1}, \ldots, \sigma_n\}$ *is the set of the output neurons that verifies the following: for each* $\sigma \in O$, $outdegree(\sigma) = 0$.
(6) Neurons $\sigma_{l+1}, \ldots, \sigma_{l+q}$ *are called internal neurons.*

FRSN P systems with real numbers constitute an extension of SN P systems in the following way (we refer to [13] for more details):

- There are two types of neurons: proposition neurons (associated with propositions in a fuzzy knowledge base) and rule neurons (associated with fuzzy production rules with AND/OR-type antecedent part). Specifically, system Π has n proposition neurons $\sigma_1, \ldots, \sigma_n$ and k rule neurons $\sigma_{n+1}, \ldots, \sigma_{n+k}$. Rule neurons are classified into two classes: AND-type rule neuron and OR-type rule neuron.

$$\overbrace{\underbrace{\sigma_1, \ldots, \sigma_l}_{\textit{input neurons}}, \underbrace{\sigma_{l+1}, \ldots, \sigma_{l+q}}_{\textit{internal neurons}}, \underbrace{\sigma_{l+q+1}, \ldots, \sigma_n}_{\textit{output neurons}}}^{\textit{proposition neurons}}, \overbrace{\sigma_{n+1}, \ldots, \sigma_{n+k}}^{\textit{rule neurons}}$$

- Content of neuron σ_i is denoted by a fuzzy truth value $\alpha_i \in [0,1]$ which can be interpreted as the (potential) value of spike from the view point of biological neurons. For a neuron σ_i, if $\alpha_i > 0$, we say the neuron contains a spike with (potential) value α_i; otherwise, the neuron contains no spike.
- Given that each neuron is associated with either a fuzzy proposition or a fuzzy production rule, the value $\tau_i \in [0,1]$ will be used to express the truth value of the fuzzy proposition or confidence factor of the fuzzy production rule.
- Each neuron σ_i contains only one firing/spiking rule r_i, which has the form $E/a^{\alpha} \rightarrow a^{\beta}$, where $E = a^n$ and $n \in \mathbb{N}$ is the number of input synapses from other neurons to the neuron. The condition $E = a^n$ indicates that if σ_i receives n spikes the firing/spiking rule can be applied; otherwise the rule is not enabled. When the number of spikes received by a neuron is less than n, value of the spikes received will be updated according to logical AND or OR operations.

- The firing mechanism of neurons can be described as follows. For neuron σ_i, if its firing rule $E/a^\alpha \to a^\beta$ can be applied, this means that its pulse value $\alpha > 0$ is consumed, the neuron fires, and then it produces a spike with value β: all neurons σ_j with $(i, j) \in syn$ will immediately receive the spike. Each kind of neurons use different ways to handle both α and β.

It is worth pointing out that fuzzy production rules of a fuzzy diagnosis knowledge base can be mapped into a FRSN P system model (again, we refer the reader to [13] for more details).

3 P–Lingua Syntax for FRSN P Systems with Real Numbers

In what follows we discuss an extension of the `P-Lingua` syntax to specify FRSN P systems with real numbers. Let us stress the fact that, with respect to `P-Lingua`, this variant is considered as separate model from SN P systems.

Definition of P system model

In order to define a FRSN P systems with real numbers, the first line of the `P-Lingua` file should be as follows:

```
@model<fuzzy_psystems>
```
.

Main module specification

In `P-Lingua`, instructions are organized into modules, except for global variables definitions, that are placed outside any module. At least a module is required, which is called `main`, at is the entry point to the `P-Lingua` model specification. The syntax to define this module is the following:

```
def main { /* instructions are placed here */ }
```
,

Specification of the fuzzy variant

In order to specify the kind of FRSNPS, the following sentence must be written (it has to be the first sentence in the model specification):

```
@fvariant = v;
```
,

where `v` is a positive integer specifying the variant. In the case of FRSN P systems with real numbers, `v` must set to `1`, hence `@fvariant = 1;`.

Specification of the sequential/parallel execution (experimental)

If the model is to be simulated on a `CUDA` parallel platform, the following sentence must be written below the `@fvariant` sentence:

```
@parallel;
```
.

If this sentence is not included, a sequential simulation is performed. This way of specifying the sequential/parallel execution is experimental, and may change in future versions of `P-Lingua`.

Specification of proposition neurons

In order to specify the proposition neurons present in the system, the following sentence must be written:

`@mu = p1,...,pi,...,pn;`,

where `pi` is the label of the ith proposition neuron.

Specification of input proposition neurons
In order to specify the input proposition neurons present in the system, the following sentence must be written:

`@min = pi1,...,piq,...,pis;`,

where `piq` is the label of the qth input proposition neuron, and must correspond to a proposition neuron defined in the `@mu` instruction.

Specification of output proposition neurons

In order to specify the output proposition neurons present in the system, the following sentence must be written:

`@mout = po1,...,pow,...,pod;`,

where `pow` is the label of the wth output proposition neuron, and must correspond to a proposition neuron defined in the `@mu` instruction.

Specification of rule neurons

In order to specify the rule neurons present in the system, the following sentence must be written: `@frule(...);`. This sentence format depends on the kind of fuzzy production rule being modelled. The following cases are possible:

– Simple rules of the form R_i : IF p_j THEN p_k (CF $= \tau_i$) are written as

  `@frule(Ri,taui,pj,pk);`.

– Type-1 composite rules (AND rules) of the form R_i : IF p_1 AND p_2 AND ... AND p_{k-1} THEN p_k (CF $= \tau_i$) are written as

  `@frule(Ri,taui,@fand(p1,p2,...,pk-1),pk);`.

– Type-2 composite rules of the form R_i : IF p_1 THEN p_2 AND p_3 AND ... AND p_k (CF $= \tau_i$) are written as

  `@frule(Ri,taui,p1,(p2,p3,...,pk));`.

– Type-3 composite rules (OR rules) of the form R_i : IF p_1 OR p_2 OR ... OR p_{k-1} THEN p_k (CF $= \tau_i$) are written as

  `@frule(Ri,taui,@for(p1,p2,...,pk-1),pk);`.

Next we illustrate the syntax presented above with the specification the FRSN P systems with real numbers exemplified in [13].

```
@model<fuzzy_psystems>

def main()
{
@fvariant = 1;
@parallel;

@mu = p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,p13,p14;

@fpin = (p1,0.8),(p2,0.2),(p3,0.8),(p4,0.8),(p5,0.9),
(p6,0.8),(p7,0.2),(p8,0.9),(p9,0.1),(p10,0.2);

@fpout = p11,p12,p13,p14;

@frule(r1,0.8,@fand(p1,p2),p11);
@frule(r2,0.8,@fand(p3,p4,p5,p6),p12);
@frule(r3,0.8,@fand(p5,p7,p8,p9),p13);
@frule(r4,0.8,@fand(p4,p5,p10),p14);
}
```

In this example, a parallel simulation is performed.

4 Simulating FRSN P Systems with Real Numbers

In this Section we present a matrix-based simulation algorithm for simulating FRSN P systems with real numbers (a modified version from the one shown in [13]) and we discuss on simulation of such systems into `P-Lingua` framework. Two simulators are provided, a sequential one (written in Java), intended to run on traditional CPUs, and a parallel one, able to be executed on `CUDA`-enabled GPUs. This last simulator works by means of a `JAVA-CUDA` binding provided by the open source `JCUDA` library, available for Windows, Linux, MacOS and other operating systems.

4.1 Simulation Algorithm

In what follows, we introduce a simulation algorithm for FRSN P systems with real numbers. In general, simulation algorithms capture semantics of the simulated models, reproducing one or many of the associated computations. In the case of FRSN P systems with real numbers, since these systems are deterministic (and thus confluent), providing an algorithm reproducing a single computation is enough. The algorithm that we are presenting is a revised version of the one introduced in [13], which re-defines the matrix-based functions and operations as well as provides an alternative way to compute fuzzy truth values for rule neurons. As it is a matrix-based algorithm, it is specially suitable to run on parallel platforms such a `CUDA` systems.

Before presenting the simulation algorithm, let us introduce some required notations, operations and functions, which closely follows from [13].

Let $\Pi = (A, \sigma_1, \ldots, \sigma_{n+k}, syn, I, O)$ be a FRSN P system with real numbers modelling all fuzzy production rules in a fuzzy knowledge base. Then, we can consider the following:

1. The set of neurons $\sigma = (\sigma_1, \ldots, \sigma_{n+k})$, composed of n proposition neurons and k rule neurons;
2. The set of n proposition neurons $\sigma_p = (\sigma_{p1}, \ldots, \sigma_{pn})$;
3. The set of k rule neurons $\sigma_r = (\sigma_{r1}, \ldots, \sigma_{rk})$, with each of them being either an AND-type or OR-type rule neuron;
4. The set $I = \{\sigma_{p_{i1}}, \ldots, \sigma_{p_{is}}\}$, of input proposition neurons, corresponding to fuzzy proposition neurons which fuzzy truth values are known;
5. The set $O = \{\sigma_{r_{o1}}, \ldots, \sigma_{r_{od}}\}$, of output proposition neurons, corresponding to fuzzy proposition neurons which fuzzy truth values are unknown and to be determined;

Let us consider the following vector and matrix notations:

1. $U = (u_{i,j})_{n\times k}$ is a binary matrix, where $u_{i,j} \in \{0,1\}$, defined as follows:

$$u_{i,j} = \begin{cases} 1 \text{ if there is a directed arc from } \sigma_{pi} \text{ to } \sigma_{rj}; \\ 0 \text{ otherwise;} \end{cases}$$

2. $V = (v_{i,j})_{n\times k}$ is a binary matrix, where $v_{i,j} \in \{0,1\}$, defined as follows:

$$v_{i,j} = \begin{cases} 1 \text{ if there is a directed arc from } \sigma_{rj} \text{ to } \sigma_{pi}; \\ 0 \text{ otherwise;} \end{cases}$$

3. $\Lambda = diag(\tau_{r1}, \ldots, \tau_{rk})$ is a diagonal real matrix, where τ_{rj} represents the confidence factor of the jth production rule, which is associated with rule neuron σ_{rj};
4. $H_1 = diag(h_1, \ldots, h_k)$ is a diagonal binary matrix, defined as follows:

$$h_j = \begin{cases} 1 \text{ if the } j\text{th rule neuron } \sigma_{rj} \text{ is an AND-type neuron;} \\ 0 \text{ otherwise;} \end{cases}$$

5. $H_2 = diag(h_1, \ldots, h_k)$ is a diagonal binary matrix, defined as follows:

$$h_j = \begin{cases} 1 \text{ if the } j\text{th rule neuron } \sigma_{rj} \text{ is an OR-type neuron;} \\ 0 \text{ otherwise;} \end{cases}$$

6. $\alpha_p = (\alpha_{p1}, \ldots, \alpha_{pn})^T$ is a truth value vector, where $\alpha_{pi} \in [0,1]$ represents the truth value of ith proposition neuron σ_{pi};
7. $\alpha_r = (\alpha_{r1}, \ldots, \alpha_{rk})^T$ is a truth value vector, where $\alpha_{rj} \in [0,1]$ represents the truth value of jth rule neuron σ_{rj};
8. $a_p = (a_{p1}, \ldots, a_{pn})^T$ is an integer vector, where a_{pi} represents the number of spikes received by the ith proposition neuron σ_{pi};

9. $a_r = (a_{r1}, \ldots, a_{rk})^T$ is an integer vector, where a_{rj} represents the number of spikes received by the jth rule neuron σ_{rj};
10. $\lambda_p = (\lambda_{p1}, \ldots, \lambda_{pn})^T$ is an integer vector, where λ_{pi} represents the number of spikes required to fire the ith proposition neuron σ_{pi};
11. $\lambda_r = (\lambda_{r1}, \ldots, \lambda_{rk})^T$ is an integer vector, where λ_{rj} represents the number of spikes required to fire the jth rule neuron σ_{rj};
12. $\beta_p = (\beta_{p1}, \ldots, \beta_{pn})^T$ is a truth value vector, where $\beta_{pi} \in [0, 1]$ represents the truth value exported by the ith proposition neuron σ_{pi} after firing;
13. $\beta_r = (\beta_{r1}, \ldots, \beta_{rk})^T$ is a truth value vector, where $\beta_{rj} \in [0, 1]$ represents the truth value exported by the jth rule neuron σ_{rj} after firing;
14. $b_p = (b_{p1}, \ldots, b_{pn})^T$ is an integer vector, where $b_{pi} \in \{0, 1\}$ represents the number of spikes exported by the ith proposition neuron σ_{pi} after firing;
15. $b_r = (b_{r1}, \ldots, b_{rk})^T$ is an integer vector, where $b_{rj} \in \{0, 1\}$ represents the number of spikes exported by the jth rule neuron σ_{rj} after firing;
16. $o_p = (o_{p1}, \ldots, o_{pn})^T$ is a binary vector, where $o_{pi} \in \{0, 1\}$, defined as follows:

$$o_{pi} = \begin{cases} 1 \text{ if } outdegree(\sigma_{pi}) > 0; \\ 0 \text{ otherwise;} \end{cases}$$

17. $o_r = (o_{r1}, \ldots, o_{rk})^T$ is a binary vector, where $o_{rj} \in \{0, 1\}$, defined as follows:

$$o_{rj} = \begin{cases} 1 \text{ if } outdegree(\sigma_{rj}) > 0; \\ 0 \text{ otherwise;} \end{cases}$$

Let us consider the following matrix functions:

1. diag: $D = diag(b)$, where $D = (d_{i,j})$ is a $f \times f$ diagonal real matrix and $b = (b_1, \ldots, b_f)$ a real vector, such that

$$d_{i,j} = \begin{cases} b_i \text{ if } i = j \\ 0 \ \text{ if } i \neq j \end{cases}, 1 \leq i, j \leq f;$$

2. fire: $\beta = fire(\alpha, a, \lambda, o)$, where $\beta = (\beta_1, \ldots, \beta_f)^T, \alpha = (\alpha_1, \ldots, \alpha_f)^T, a = (a_1, \ldots, a_f)^T, \lambda = (\lambda_1, \ldots, \lambda_f)^T, o = (o_1, \ldots, o_f)^T$, such that

$$\beta_i = \begin{cases} 0 \ \text{ if } a_i < \lambda_i \\ \alpha_i \text{ if } a_i = \lambda_i \wedge o_i = 0 \\ 0 \ \text{ if } a_i = \lambda_i \wedge o_i = 1 \end{cases}, 1 \leq i \leq f;$$

3. update: $\beta = update(\alpha, a, \lambda, o)$, where $\beta = (\beta_1, \ldots, \beta_f)^T, \alpha = (\alpha_1, \ldots, \alpha_f)^T$, $a = (a_1, \ldots, a_f)^T, \lambda = (\lambda_1, \ldots, \lambda_f)^T, o = (o_1, \ldots, o_f)^T$, such that

$$\beta_i = \begin{cases} 0 \ \text{ if } a_i = 0 \\ \alpha_i \text{ if } 0 < a_i < \lambda_i \\ 0 \ \text{ if } a_i = \lambda_i \wedge o_i = 0 \\ \alpha_i \text{ if } a_i = \lambda_i \wedge o_i = 1 \end{cases}, 1 \leq i \leq f;$$

Let us consider the following matrix operations:

1. $\oplus : C = A \oplus B$, where A, B, C are $f \times g$ matrices whose elements are non-negative real numbers, such that

$$c_{i,j} = \begin{cases} 0 & \text{if } a_{i,j} = 0 \wedge b_{i,j} = 0 \\ b_i & \text{if } a_{i,j} = 0 \wedge b_{i,j} > 0 \\ a_i & \text{if } a_{i,j} > 0 \wedge b_{i,j} = 0 \\ max\{a_{i,j}, b_{i,j}\} & \text{if } a_{i,j} > 0 \wedge b_{i,j} > 0 \end{cases}, 1 \leq i \leq f, 1 \leq j \leq g;$$

2. $\ominus : C = A \ominus B$, where A, B, C are $f \times g$ matrices whose elements are non-negative real numbers, such that

$$c_{i,j} = \begin{cases} 0 & \text{if } a_{i,j} = 0 \wedge b_{i,j} = 0 \\ b_i & \text{if } a_{i,j} = 0 \wedge b_{i,j} > 0 \\ a_i & \text{if } a_{i,j} > 0 \wedge b_{i,j} = 0 \\ min\{a_{i,j}, b_{i,j}\} & \text{if } a_{i,j} > 0 \wedge b_{i,j} > 0 \end{cases}, 1 \leq i \leq f, 1 \leq j \leq g;$$

3. $\otimes : C = A \otimes B$, where A, B, C are $f \times g, g \times h, f \times h$, matrices respectively, whose elements are non-negative real numbers, such that

$$S_{i,j} = \{a_{i,l} \cdot b_{l,j}, 1 \leq l \leq g\} \setminus \{0\}, 1 \leq i \leq f, 1 \leq j \leq h;$$

$$c_{i,j} = \begin{cases} 0 & \text{if } |S_{i,j}| = 0 \\ max\, S_{i,j} & \text{if } |S_{i,j}| > 0 \end{cases}, 1 \leq i \leq f, 1 \leq j \leq h;$$

4. $\odot : C = A \odot B$, where A, B, C are $f \times g, g \times h, f \times h$, matrices respectively, whose elements are non-negative real numbers, such that

$$S_{i,j} = \{a_{i,l} \cdot b_{l,j}, 1 \leq l \leq g\} \setminus \{0\}, 1 \leq i \leq f, 1 \leq j \leq h;$$

$$c_{i,j} = \begin{cases} 0 & \text{if } |S_{i,j}| = 0 \\ min\, S_{i,j} & \text{if } |S_{i,j}| > 0 \end{cases}, 1 \leq i \leq f, 1 \leq j \leq h;$$

Finally, we introduce the matrix-based simulation algorithm for FRSN P systems with real numbers.

FRSN P systems with real numbers simulation algorithm

- INPUT:
  - $U, V, \Lambda, H_1, H_2, \lambda_p, \lambda_r$;
  - $\alpha_p^0 = (\alpha_{p1}^0, \ldots, \alpha_{pn}^0)$, with $\alpha_{pi}^0 = \begin{cases} \tau_{pi} \text{ if } \sigma_{pi} \in I, \tau_{pi} \text{ is the CF of } \sigma_{pi}; \\ 0 \quad \text{otherwise}; \end{cases}$
  - $a_p^0 = (a_{p1}^0, \ldots, a_{pn}^0)$, with $a_{pi}^0 = \begin{cases} 1 \text{ if} \sigma_{pi} \in I; \\ 0 \text{ otherwise}; \end{cases}$
- OUTPUT:
  - $\alpha_{pout} = (\alpha_{p_{i1}}, \ldots, \alpha_{p_{is}})^T$, the vector containing the fuzzy truth values of proposition neurons in O.

Step 1. Let $\alpha_r^0 = (0, \ldots, 0)^T, a_r^0 = (0, \ldots, 0)^T$.

Step 2. Let $t = 0$.

Step 3. Do:

```
(1) Prepare firing of proposition neurons.
      * $\beta_p^t = fire\big(\alpha_p^t, a_p^t, \lambda_p, o_p\big)$.
      * $b_p^t = fire\big(1, a_p^t, \lambda_p, o_p\big)$.
      * $\alpha_p^t = update\big(\alpha_p^t, a_p^t, \lambda_p, o_p\big)$.
      * $a_p^t = update\big(a_p^t, a_p^t, \lambda_p, o_p\big)$.
      * $B_p^t = diag\big(b_p^t\big)$.
(2) Prepare firing of rule neurons.
      * $\beta_r^t = fire\big(\alpha_r^t, a_r^t, \lambda_r, o_r\big)$.
      * $b_r^t = fire\big(1, a_r^t, \lambda_r, o_r\big)$.
      * $\alpha_r^t = update\big(\alpha_r^t, a_r^t, \lambda_r, o_r\big)$.
      * $a_r^t = update\big(a_r^t, a_r^t, \lambda_r, o_r\big)$.
      * $B_r^t = diag\big(b_p^t\big)$.
(3) Update truth values and received spikes for proposition neurons.
      * $\alpha_p^{t+1} = \alpha_p^t \oplus \Big((V \cdot B_r^t) \otimes \beta_r^t\Big)$.
      * $a_p^{t+1} = a_p^t + \Big((V \cdot B_r^t) \cdot b_r^t\Big)$.
(4) Update truth values and received spikes for rule neurons.
      * $\alpha_r^{t+1} = H_1 \cdot \Big[\alpha_r^t \ominus \Big((B_p^t \cdot U)^T \odot \beta_p^t\Big)\Big] + H_2 \cdot \Big[\alpha_r^t \oplus \Big((B_p^t \cdot U)^T \otimes \beta_p^t\Big)\Big]$.
      * $a_r^{t+1} = a_r^t + \Big((B_p^t \cdot U)^T \cdot b_p^t\Big)$.
```

Step 4. Check termination condition. If the following conditions hold:

(a) $a_r^{t+1} = (0, 0, \ldots, 0)^T$;

(b) $a_p = (a_{p1}, \ldots, a_{pn})^T$, with: $a_{pi} = \begin{cases} 1 \text{ if } o_{pi} = 1 \\ 0 \text{ otherwise} \end{cases}, 1 \le i \le n$;

then HALT, otherwise go to Step 3.

4.2 P-Lingua Simulators for FRSN P Systems with Real Numbers

In [4], a Java library called `pLinguaCore` was presented, with this package being released under GPL [25] license. The library provides parsers to handle input files, built–in simulators to generate P system computations and is able to export several output file formats that represent P systems. In what follows, we detail how to invoke the brand new built–in simulators for FRSN P systems with real numbers. Two simulators are provided, a sequential one (written in Java), running on traditional CPUs, and a parallel one, able to run on `CUDA`-enabled devices. The parallel simulator uses a `CUDA` kernel in which threads compute the results of the different matrix-based operations executed in the simulation algorithm described above. This paper version of `pLinguaCore` library can be found at www.p-lingua.org/mecosim/.

Invoking the Sequential Simulator. Invoking the sequential simulator requires for the system to host a Java runtime environment properly installed and configured. The Java runtime can be found at https://java.com/es/download/. Also, the following directory structure must be created:

```
plingua/
├── plinguacore.jar
└── input.pli
```

The `plingua` directory contains all the required files to run the simulation. Files description follows:

- `plinguacore.jar` file hosts the `pLinguaCore` library.
- `input.pli` file hosts the FRSN P systems with real numbers model to simulate.

Once the files are ready, to invoke the simulator a system console must be opened and the following command has to be executed from the `plingua` directory:

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

This will produce an output file named `output.txt` in `plingua` directory where information about the parser process and the generated computation is stored.

Invoking the Parallel Simulator. Invoking the parallel simulator requires for the system to host both a Java runtime environment and a `CUDA`-enabled GPU device, with the corresponding `NVIDIA` driver with `CUDA` support and the `CUDA Toolkit` properly installed and configured. The `NVIDIA` software can be found at https://developer.nvidia.com/cuda-downloads. In order to interface the Java `pLinguaCore` library with the `CUDA` platform, a `JAVA-CUDA` binding is required, which is provided by the `JCUDA` library. In the present paper, version 0.6.5 of such library is used, as well as version 0.0.4 of `JCudaUtils` library, which contains a series of utility methods used by `JCUDA` library. Both of them can be found at http://www.jcuda.org/. Also, the following directory structure must be created:

```
plingua/
├── plinguacore.jar
├── input.pli
├── kernelReal.cu
├── jcudaUtils-0.0.4.jar
└── jcuda-0.6.5/
    └── *** jcuda-0.6.5 library files ***
```

The `plingua` directory contains all the required files to run the simulation. Files description follows:

- `plinguacore.jar` file hosts the `pLinguaCore` library.
- `input.pli` file hosts the FRSN P systems with real numbers model to simulate.
- `kernelReal.cu` file hosts the CUDA kernel corresponding to the parallel implementation.
- `jcudaUtils-0.0.4.jar` file hosts `JCudaUtils` library.
- `jcuda-0.6.5` folder hosts the contents of the zip file corresponding to the 0.6.5 version of `JCUDA` library.

Once the files are ready, to invoke the simulator a system console must be opened and the following command has to be executed from the `plingua` directory:

```
java -Djava.library.path=jcuda/
-cp"pLinguaCore.jar;jcudaUtils-0.0.4.jar;jcuda/jcuda-0.6.5.jar"
org.gcn.plinguacore.applications.AppMain
plingua_sim -pli input.pli -o output.txt
```

This will produce an output file named `output.txt` in `plingua` directory where information about the parser process and the generated computation is stored. Note: the `-cp` parameter uses the symbol "`;`" as element separator in Windows platforms. Other platforms use different separators. For example, Unix platforms use the symbol "`:`".

Parallel Simulator `CUDA` Compatibility and Performance Considerations. When developing the parallel simulator, the main goal was to make it able to handle arbitrary matrix size instances and to run on the majority of `CUDA`-compatible devices. This has involved making conservative choices in the implementation. A standard *block size* equal to 256 (16*16) has been chosen and the *tiling/memory coalescing* optimization technique has been applied, which requires a relatively low amount of *shared memory* for blocks (see [6] for more details). Fixing matrix size instances and minimum requirements for the `CUDA`-compatible device would enable implementing more complex optimization techniques, such as *loop unrolling, data prefetching and thread granularity* as well as a fine grained performance analysis. The appropriate combinations of performance tuning techniques can make tremendous difference in the performance achieved by the simulator; however the programming efforts to manually search through these combinations is quite large [6]. Automation tools to reduce such efforts such as `CUDA`-lite [18] and others become indispensable.

5 Conclusions and Future Work

In this paper we introduce `P-Lingua` framework support for a new P system variant, specifically FRSN P systems with real numbers, which incorporate fuzzy logic elements into SN P systems. The motivation of this variant is to produce a framework bringing together desirable features (understandable, dynamical, synchronized, non-linear, non-deterministic, able to handle incomplete and uncertain information) to model diagnosis knowledge and reasoning in the field of fault diagnosis. Applications of this variant are very promising, which are related to fault diagnosis of electrical systems [13,21]. In consequence, providing the corresponding `P-Lingua` support favours the research on this model within Membrane Computing community as well as in applied fields. Developing such support has involved designing a specific parser (since with respect to `P-Lingua`, FRSN P systems are considered a "separated" variant of SN P systems), a simulation algorithm (which is a variant of the one introduced in [13]) and the corresponding simulators. As the provided simulation algorithm is a matrix-based one, which can take advantage of parallel computing platforms, along with a Java sequential simulator, a parallel one has been developed intended to be able to run on the majority of `CUDA`-compatible devices.

As open research lines, we can identify addressing others FRSN P systems variants, dealing with trapezoidal numbers, weights, etc. and considering the implementation of more complex optimization techniques, possibly assisted by automation optimization tools.

Acknowledgements. This work was supported by Project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain, cofinanced by FEDER funds. The authors also acknowledge the support of the GPU Research Center program granted by NVIDIA to the University of Seville.

References

1. Binder, A., Freund, R., Oswald, M., Vock, L.: Extended spiking neural P systems with excitatory and inhibitory astrocytes. In: Proceedings of the 8th Conference on 8th WSEAS International Conference on Evolutionary Computing, EC 2007, vol. 8, pp. 320–325. World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA (2007)
2. Chen, H., Freund, R., Ionescu, M., Păun, G., Pérez-Jiménez, M.J.: On string languages generated by spiking neural P systems. Fundam. Inform. **75**(1–4), 141–162 (2007)
3. 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. Nat. Comput. **7**(2), 147–166 (2008)
4. García-Quismondo, M., Gutiérrez-Escudero, R., del Amor, M.A.M., Orejuela-Pinedo, E.F., Pérez-Hurtado, I.: P-lingua 2.0: a software framework for cell-like P systems. Int. J. Comput. Commun. Control **4**, 234–243 (2009)
5. Ionescu, M., Păun, G., Yokomori, T.: Spiking neural P systems. Fundam. Inf. **71**, 279–308 (2006)
6. Kirk, D.B., Hwu, WmW: Programming Massively Parallel Processors: A Hands-on Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2010)
7. Macías-Ramos, L.F., Pérez-Jiménez, M.J.: Spiking neural P systems with functional astrocytes. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) CMC 2012. LNCS, vol. 7762, pp. 228–242. Springer, Heidelberg (2013)

8. Martín-Vide, C., Păun, G., Pazos, J., Rodríguez-Patón, A.: Tissue P systems. Theor. Comput. Sci. **296**(2), 295–326 (2003)
9. Pan, L., Păun, G.: Spiking neural P systems with anti-spikes. Int. J. Comput. Commun. Control **4**, 273–282 (2009)
10. Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with neuron division and budding. Sci. China Inf. Sci. **54**(8), 1596–1607 (2011)
11. Pan, L., Wang, J., Hoogeboom, H.J.: Asynchronous extended spiking neural P systems with astrocytes. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) CMC 2011. LNCS, vol. 7184, pp. 243–256. Springer, Heidelberg (2012)
12. Pan, L., Wang, J., Hoogeboom, H.J.: Limited asynchronous spiking neural P systems. Fundam. Inform. **110**(1–4), 271–293 (2011)
13. Peng, H., Wang, J., Pérez-Jiménez, M.J., Wang, H., Shao, J., Wang, T.: Fuzzy reasoning spiking neural P system for fault diagnosis. Inf. Sci. **235**, 106–116 (2013)
14. Păun, A., Păun, G.: Small universal spiking neural P systems. Biosystems **90**(1), 48–60 (2007)
15. Păun, G.: Computing with membranes. J. Comput. Syst. Sci. **61**, 108–143 (1998)
16. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press Inc, New York (2010)
17. Song, T., Pan, L., Păun, G.: Asynchronous spiking neural P systems with local synchronization. Inf. Sci. **219**, 197–207 (2013)
18. Ueng, S.-Z., Lathara, M., Baghsorkhi, S.S., Hwu, W.W.: CUDA-Lite: reducing GPU programming complexity. In: Amaral, J.N. (ed.) LCPC 2008. LNCS, vol. 5335, pp. 1–15. Springer, Heidelberg (2008)
19. Wang, J., Hoogeboom, H.J., Pan, L., Păun, G., Pérez-Jiménez, M.J.: Spiking neural P systems with weights. Neural Comput. **22**(10), 2615–2646 (2010)
20. Wang, T., Zhang, G., Pérez-Jiménez, M.J.: Application of weighted fuzzy reasoning spiking neural P systems to fault diagnosis in traction power supply systems of high-speed railways. In: Twelfth Brainstorming Week on Membrane Computing (BWMC2014), pp. 329–350 (2014)
21. Wang, T., Zhang, G., Pérez-Jiménez, M.J.: Fault diagnosis models for electric locomotive systems based on fuzzy reasoning spiking neural P systems. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) CMC 2014. LNCS, vol. 8961, pp. 385–395. Springer, Heidelberg (2014)
22. Wang, T., Zhang, G., Rong, H., Pérez-Jiménez, M.J.: Application of fuzzy reasoning spiking neural P systems to fault diagnosis. Int. J. Comput. Commun. Control **9**, 720–733 (2014)
23. Wang, T., Zhang, G., Zhao, J., He, Z., Wang, J., Pérez-Jiménez, M.: Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems. IEEE Trans. Power Syst. **30**(3), 1182–1194 (2015)
24. Web-page: The CUDA Website. https://developer.nvidia.com/cuda-zone
25. Web-page: The GNU GPL Website. http://www.gnu.org/copyleft/gpl.html
26. Web-page: The Java Website. https://www.java.com/
27. Web-page: The JCUDA Website. http://www.jcuda.org/
28. Web-page: The P-Lingua Website. http://www.p-lingua.org/