ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **23**, Number 1, 2020, 84–92

A Software Tool for Spiking Neural P Systems

Taosheng ZHANG[1], Fei XU[1], and Tingfang WU[2,*]

[1]Key Laboratory of Image Information Processing and Intelligent Control of Education Ministry of China, School of Artificial Intelligence and Automation, Huazhong University of Science and Technology, Wuhan
[2]School of Computer Science and Technology, Soochow University, Suzhou
E-mails: tfwu@suda.edu.cn

Abstract. Software simulators for spiking neural P systems (SN P systems, for short) are the main tool to experimentally explore the computational properties of these systems. Gutiérrez-Naranjo et al. developed a software tool for generating the transition diagram of an SN P system in an automatic way, which can assist in the formal verification of SN P systems. However, the forms of rules accepted in their simulator are restricted, i.e., the simulator accepts only seven syntactically different types of rules. In this work, a software tool that allows to automatically generate the transition diagram of the SN P system is developed based on Python language, and it can accept any form of rules. Hence, the simulator may play an important role in assisting the verification of SN P systems.

Key-words: Membrane computing, Spiking neural P systems, Simulator, Transition diagram.

1. Introduction

Natural computing is a research field which investigates new computation paradigms inspired from natural phenomena [1]. Motivated by the structure and functioning of a living cell or of multiple cells, an active branch of natural computing, called membrane computing, was proposed in order to abstract powerful computation models or design efficient algorithms [2]. Usually, the computation models in the field of membrane computing are called P systems. Since the introduction of P systems, a large number of variants have come forth, mainly falling into three categories according to their topological structure: cell-like P systems, whose structure corresponds to a tree [2]; tissue-like P systems, whose structure corresponds to an undirected graph [3]; neural-like P systems, whose structure corresponds to a directed graph [4]. Most of these P systems were proved to be Turing universal [5–7], also capable of efficiently solving computationally hard problems [8–10].

Spiking neural P systems (for short, SN P systems) are a class of parallel computing models, inspired by the way in which neurons process information and communicate to each other by

*Corresponding author

means of spikes [4]. SN P systems belong to the field of neural-like P systems, incorporating the idea of spiking neurons into membrane computing models [11, 12].

Generally, the structure of an SN P system corresponds to a directed graph. Specifically, each node in the directed graph represents a neuron, and each edge in the directed graph represents a synapse between neurons. Each neuron in the system contains data structure represented in the form of multisets of spikes, and data operation represented in the form of spiking rules and forgetting rules. The application of these two types of rules depends on whether the contents of the neuron is described by a regular expression associated with the rule. By applying a spiking rule, a neuron consumes a predefined number of spikes and produces a predefined number of spikes, then sends these produced spikes to its neighbouring neurons. The application of a forgetting rule only consumes a predefined number of spikes, but produces no spike. The computation result of the system can be encoded in different modes [13], e.g., number encoding mode [14], i.e., the number of spikes in the output region at the moment when the computation halts, or time encoding mode [4], i.e., the time elapsed between the first two consecutive spikes sent to the environment by the system.

A great deal of work with respect to SN P systems has been carried out and led to many achievements. For example, SN P systems have been proved to be computationally powerful, e.g., as number generating/accepting devices [4], string language generators [15–17], and function computing devices [18]. Inspired by different biological phenomena and mathematical motivations, various variants of SN P systems have been proposed, e.g., axon P systems [19], SN P systems with structural plasticity [20], SN P systems with scheduled synapses [21], and cell-like SN P systems [22]. Most of these variants are also proved to be Turing universal. Some practical applications have also been carried out with SN P systems and their variants, e.g., fault diagnosis [23] and image processing [24, 25].

In the framework of membrane computing, the idea of automating the evolution for various classes of P systems is also an important research line. Up to now, a wide range of software simulators have been developed for various types of P systems, such as the Membrane Simulator for catalytic hierarchical cell systems and active membrane systems [26], and the SNUPS Simulator for numerical P systems [27]. Moreover, P-Lingua offers a general syntactic framework that defines a unified standard for different classes of P systems, e.g., cell-like P systems [28, 29], SN P systems [30], and cell-like SN P systems [31]. It is usually a hard task to achieve the formal verification of an SN P system designed for solving a given problem, and there is no general methodology in this respect [32]. In order to reduce the complexity of this task, Gutiérrez-Naranjo et al. developed a software tool which allows to automatically generate the transition diagram of an SN P system [33]. Therefore, this software tool can play an important role assisting in the formal verification of SN P systems. However, the form of rules accepted in this software tool is very restricted; specifically, the simulator accepts only seven syntactically different types of rules.

In this work, in order to overcome the restriction of the aforementioned simulator, a new simulator for generating the transition diagram of an SN P system in an automatic way is developed based on Python language. The input of the simulator is a structured text file with the syntax of an SN P system defined in an intuitive way, including the rules, the set of synapses, and the initial number of spikes present in each neuron. According to the semantics of the SN P system, the simulator executes the given SN P system and automatically generates the transition diagram associated with the processed model. Different from the simulator developed in [33], this new simulator can accept any form of rules. In this way, the simulator developed in this work is more general, and more powerful as an auxiliary tool for assisting in verifying SN P systems.

The rest of this paper is organized as follows. The formal definition of SN P systems is recalled in the next section. The software simulator and some features of its implementation are presented in Section 3. Section 4 provides an example implemented in the simulator. The last section gives some conclusions about this work.

2. Spiking Neural P Systems

This section mainly recalls the formal definition of SN P systems. For more information about these models, please refer to [4]. An SN P system of degree $m \geq 1$ has the following structure:

$$\Pi = (O, \sigma_1, \sigma_2, \cdots, \sigma_m, syn, in, out)$$

where:

- $O = \{a\}$ is a singleton alphabet, and character a represents the spike;
- σ_1, σ_2, $\cdots$, σ_m are m neurons of system Π, and each neuron σ_i is of the form $\sigma_i = (n_i, R_i)$, $1 \leq i \leq m$, where:
  (a) $n_i \geq 0$ represents the initial number of spikes contained in neuron σ_i;
  (b) R_i represents a finite set of rules in neuron σ_i, with the form $E/a^c \rightarrow a^p; d$, where E is a regular expression defined on the singleton alphabet $\{a\}$, $c \geq 1$, $p \geq 0$, $d \geq 0$, and $c \geq p$; particularly, if $p = 0$, then $d = 0$;
- $syn \subseteq \{1, 2, \cdots, m\} \times \{1, 2, \cdots, m\}$ is the set of the synapses. For each i, $1 \leq i \leq m$, $(i, i) \notin syn$;
- $in, out \in \{1, 2, \cdots, m\}$ represent the input neuron and the output neuron, respectively.

As usual, if $p = 0$, then rule $E/a^c \rightarrow a^p; d$ is written in the simpler form $E/a^c \rightarrow \lambda$, which is called a forgetting rule. When the regular expression E associated with rule $E/a^c \rightarrow a^p; d$ is of the form $E = a^c$, the rule is written in the simplified form $a^c \rightarrow a^p; d$ or $a^c \rightarrow \lambda$.

The semantics of applying a rule $E/a^c \rightarrow a^p; d$ is as follows. Assume that at a certain time, neuron σ_i contains k spikes, and the number of spikes inside it satisfies the two firing conditions: (1) $a^k \in L(E)$, and (2) $k \geq c$. Then the rule $E/a^c \rightarrow a^p; d$ can be applied. The result of the application of the rule is that c spikes are consumed, and p spikes are generated after d time units in neuron σ_i. The parameter d is the time interval between the application of the rule and the emitting of spikes, called the delay, which simulates the refractory period in biological neuron. Specifically, if $d = 0$, then the generated spikes are emitted immediately. If $d \geq 1$, and the rule is applied at step t, then at steps t, $t + 1$, $\cdots$, $t + d - 1$, neuron σ_i is closed, that is, the neuron can neither receive spikes from other presynaptic neurons, nor send spikes to other postsynaptic neurons. If spikes are sent to a closed neuron, then the spikes will be lost. At step $t + d$, the neuron becomes open, and it can again receive spikes. The semantics of applying a forgetting rule $E/a^c \rightarrow \lambda$ is similar, and the application of the forgetting rule just consumes c spikes, but no spike is generated and the neuron is continuously open.

The work of the system proceeds as follows. In each time unit, each neuron which can use a rule must apply one of its enabled rules, unless there is no rule satisfying the firing conditions inside it. Therefore, each neuron works sequentially, but the system works concurrently.

A configuration of an SN P system at certain step t is defined as $C_t = \langle k_1/t_1, k_2/t_2, \cdots , k_m/t_m\rangle$, where k_i represents the number of spikes contained in neuron σ_i, and t_i indicates the number of steps needed to go from the closed state to the open one. Using the notation introduced above, the initial configuration of the system is expressed as $C_0 = \langle n_1/0, n_2/0, \cdots , n_m/0\rangle$. The passage from a configuration to a succeeding one is called a transition. Any sequence of transitions starting from the initial configuration forms a computation. When all neurons in the system are open but without any rule that can be used, the computation halts.

3. The Simulator for SN P Systems

As usual in membrane computing, an SN P system is presented in a graphical form: the neurons of the system are represented by labeled rounded rectangles, with the initial number of spikes and a set of spiking rules placed inside, while the synapses are represented by arrows between rounded rectangles. The output neuron of the system has an additional outgoing arrow, indicating its communication with the environment. Figure 1 presents the graphical representation of an SN P system.

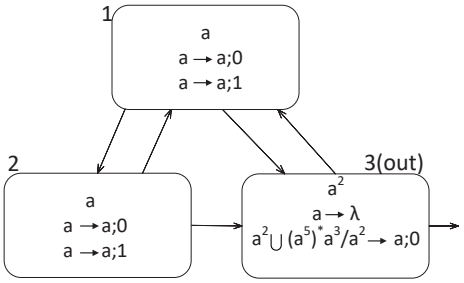

Fig. 1. An SN P system

3.1. Software Framework

The simulator presented in this work is written in Python language. As Python is a type of interpreted language, this simulator can work on different platforms which only needs to support Python interpreter, such as Windows, Linux, and OSX, thus it has a high portability. Since the simulator can automatically generate the transition diagram, besides a standard installation of Python, the library *graphviz* is required to be installed for drawing pictures.

The simulator consists of a single Python script with some necessary libraries, which are used for processing the structured text input file, executing the processed model, and outputting the transition diagram. The object-oriented data structure of the simulator is illustrated in Figure 2 by using an UML diagram. According to Figure 2, the software architecture of the simulator coincides with the formal definition of the theoretical model, thus offering the expected execution pattern.

The software framework of the simulator is shown in Figure 3. Firstly, the simulator reads a structured text input file that stores the description of an SN P system, then converts it into the object-oriented data structure shown in Figure 2. According to the semantics of the SN P system, the simulator executes the system and builds the transition diagram of the current step.

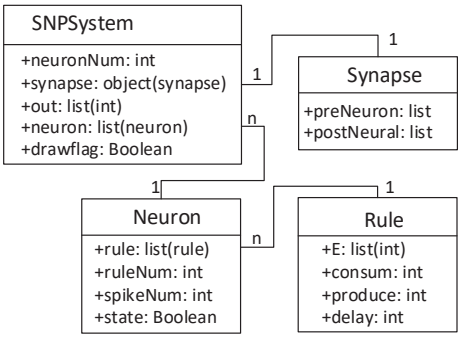

Fig. 2. UML class diagram of the simulator

The termination condition of the simulator is that the computation of SN P system halts or a number of execution steps prescribed in advance is reached. Once the termination condition is reached, then the simulator stops, and outputs the transition diagram associated with the given SN P system.

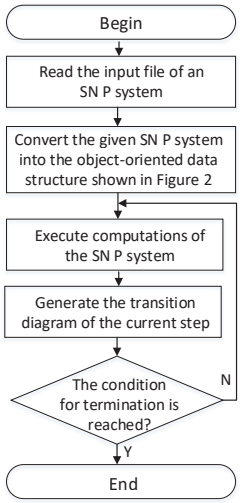

Fig. 3. The software framework of the simulator

3.2. Sample Code Analysis

The syntax of the input files accepted by the simulator is quite intuitive, intended to resemble as closely as possible the formal definition of an SN P system. Figure 4 shows the input file of the example presented in Figure 1.

The input file of the simulator consists on a set of literals with the predicate symbols `snp`, `neuronNum`, `rule`, `synapses`, and `initial`, some of which follow the syntax initially presented in [33].

– The `snp` defines an SN P system;

– The `neuronNum` defines the number of neurons in the SN P system;

– The `rule` maintains the syntax of the rules of the form (`N-X`, `Latex`), where `N` is the label of the neuron, `X` is the label of the rule, and `Latex` is the rule written in Latex mode. In contrast with the simulator developed in [33] that only accepts seven syntactically different types of rules, the simulator developed in this work can accept any type of rules of SN P systems:

- `a^s/a^c -> a^p:d` meaning $a^s/a^c \rightarrow a^p; d$;
- `a^s/a^c -> a:d` meaning $a^s/a^c \rightarrow a; d$
- `a^[`$s_1, s_2, \cdots, s_n$`]/a^c -> a^p:d` meaning $\{a^{s_1} \cup a^{s_2} \cup \cdots \cup a^{s_n}\}/a^c \rightarrow a^p; d$
- `a^[`$s_1, s_2, \cdots, s_n$`]/a^c -> a:d` meaning $\{a^{s_1} \cup a^{s_2} \cup \cdots \cup a^{s_n}\}/a^c \rightarrow a; d$
- `a^[`$y_1 \star + x_1, \cdots, y_m \star + x_m$`]/a^c -> a^p:d` meaning $\{(a^{y_1})^* a^{x_1} \cup \cdots \cup (a^{y_m})^* a^{x_m}\}/a^c \rightarrow a^p; d$;
- `a^[`$y_1 \star + x_1, \cdots, y_m \star + x_m$`]/a^c -> a:d` meaning $\{(a^{y_1})^* a^{x_1} \cup \cdots \cup (a^{y_m})^* a^{x_m}\}/a^c \rightarrow a; d$;
- `a^[`$s_1, s_2, \cdots, s_l, y_1 \star + x_1, \cdots, y_m \star + x_m$`]/a^c -> a^p:d` meaning $\{a^{s_1} \cup a^{s_2} \cup \cdots \cup a^{s_l} \cup (a^{y_1})^* a^{x_1} \cup \cdots \cup (a^{y_m})^* a^{x_m}\}/a^c \rightarrow a^p; d$;
- `a^[`$s_1, s_2, \cdots, s_l, y_1 \star + x_1, \cdots, y_m \star + x_m$`]/a^c -> a:d` meaning $\{a^{s_1} \cup a^{s_2} \cup \cdots \cup a^{s_l} \cup (a^{y_1})^* a^{x_1} \cup \cdots \cup (a^{y_m})^* a^{x_m}\}/a^c \rightarrow a; d$;
- `a^c -> a^p:d` meaning $a^c \rightarrow a^p; d$;
- `a^c -> a:d` meaning $a^c \rightarrow a; d$;
- `a -> a:d` meaning $a \rightarrow a; d$;
- `a^s -> lambda` meaning $a^s \rightarrow \lambda$;
- `a -> lambda` meaning $a \rightarrow \lambda$.

– The `synapses` is a list of pairs of neurons `Start-End`;

– The `initial` is the initial configuration of the system. A configuration is a list of pairs `A/B`, where `A` represents the number of spikes, and `B` represents the number of steps for the neuron to be open from the closed state.

The input file is written in standard ASCII format. In order to make the input file read by the simulator, the input file needs to be modified with some machine processing differences, including the tree structure of the file, block containers delimited using '{' and '}' or '[' and ']', and the semi-colon used as an instruction finish marker. The comma is used only to separate list items, e.g., `synapses`, the colon is used to separate the produced spikes and the time delay within each rule, and the arrow `'->'` that is part of each rule is used to separate the consumed and produced spikes.

4. An Example

In this section, the SN P system shown in Figure 1 will be used to illustrate how the simulator described here works. Before the simulator is run on a given SN P system, there are options in the

```
snp= {
        neuronNum = 3;
        rule = {
                (1-1,  a  -> a:0),
                (1-2,  a  -> a:1),
                (2-1,  a  -> a:0),
                (2-2,  a  -> a:1),
                (3-1,  a  -> lambda),
                (3-2,  a^[2,5*+3]/a^2-> a:0),
                };
        synapses = [1-2,2-1,1-3,3-1,2-3];
        initial = [1/0,1/0,2/0];
        }
```

Fig. 4. The input file of the SN P system shown in Figure 1

Python script: (1) the parameter *filename* associated with the name of the input file, which should be placed under the same folder as the source code, (2) the maximum number of execution steps.

After finishing the execution of all five steps (the maximum number of execution steps is set to 5), the simulator will automatically generate the transition diagram associated with the given SN P system, as shown in Figure 5. In the transition diagram of Figure 5, the content in each rectangle is a configuration of the system, while the labeling on each arrow indicates the set of rules applied in this step, where -0 indicates that there is no rule that can be used in the neuron, -*s* indicates that the neuron is closed, and the number in the right hand of the square bracket indicates the current execution step.

The overall simulation for the given SN P system takes about 5ms. Indeed, the most time-consuming step is to write the transition diagram produced into a PDF file. In this way, if a given SN P system is large, in the sense that the system contains a large number of neurons and rules, then the simulation time will greatly increase.

5. Conclusions

In this work, a new simulator based on Python language is described which automatically generate the transition diagram of an SN P system. In contrast with the simulator developed in [33] that only accepts several types of restricted forms of rules, the simulator developed in this work can accept any type of rules of SN P systems. Therefore, the simulator is more general, and the functioning of that is useful as an auxiliary tool in the formal verification of SN P systems.

Our simulator is based on scripting language, thus it is not intuitive and usually not user-friendly. It is desirable to develop a graphical user interface (GUI) that allows to provide a friendly interface to users. In addition, a future work is to develop new simulators for other types of P systems, such as cell-like P systems and tissue-like P systems.

Availability: The source code is available at: https://github.com/pizts/A-Simulator-Tool-for-Spiking-Neural-P-Systems.git.

Acknowledgements.

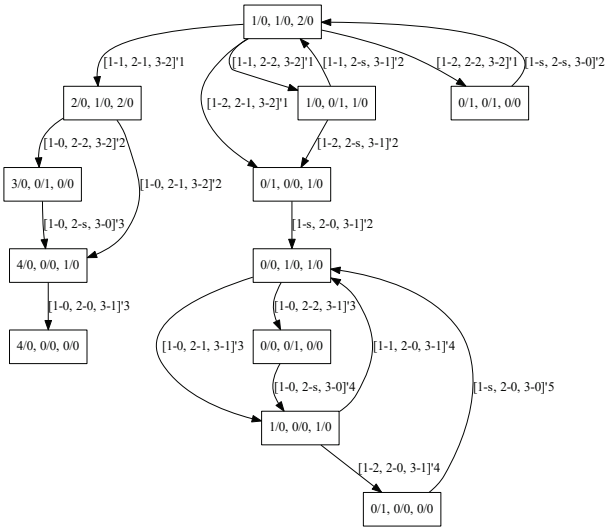

Fig. 5. The transition diagram associated with the SN P system shown in Figure 1

References

[1] G. ROZENBERG, T. BÄCK, J. N. KOK, EDS., *Handbook of Natural Computing*. Springer-Verlag, Berlin, 2012.

[2] G. PĂUN, *Computing with membranes*. J. Comput. Syst. Sci. **61**(1), 108–143, 2000.

[3] C. MARTÍN-VIDE, G. PĂUN, J. PAZOS, A. RODRÍGUEZ-PATÓN, *Tissue P Systems*, Theor. Comput. Sci. **296**(2), 295–326, 2003.

[4] M. IONESCU, G. PĂUN, T. YOKOMORI, *Spiking neural P systems*. Fundam. Inform. **71**(2), 279–308, 2006.

[5] L. PAN, G. PĂUN, B. SONG, *Flat maximal parallelism in P systems with promoters*. Theor. Comput. Sci. **623**, 83–91, 2016.

[6] A. PĂUN, G. PĂUN, *The power of communication: P systems with symport/antiport*. New Generat. Comput. **20**(3), 295–305, 2002.

[7] B. SONG, L. PAN, M.J. PÉREZ-JIMÉNEZ, *Cell-like P systems with channel states and symport/antiport rules*. IEEE T. NanoBiosci. **15**(6), 555–566, 2016.

[8] L.F. MACÍAS-RAMOS, B. SONG, L. VALENCIA-CABRERA, L. PAN, M.J. PÉREZ-JIMÉNEZ, *Membrane fission: A computational complexity perspective*. Complexity. **21**(6), 321–334, 2016.

[9] G. PĂUN, *P systems with active membranes: attacking NP-complete problems*. J. Automata Lang. Combinatorics. **6**(1), 75–90, 2001.

[10] B. SONG, M.J. PÉREZ-JIMÉNEZ, L. PAN, *An efficient time-free solution to SAT problem by P systems with proteins on membranes*. J. Comput. Syst. Sci. **82**(6), 1090–1099, 2016.

[11] W. MAASS, *Computing with spikes*. Foundations of Information Processing of TELEMATIK. **8**(1), 32–36, 2002.

[12] W. MAASS, C.M. BISHOP, EDS., *Pulsed Neural Networks*. MIT Press, Cambridge, 2001.

[13] L. PAN, T. WU, Y. SU, A.V. VASILAKOS, *Cell-like spiking neural P systems with request rules*. IEEE T. NanoBiosci. **16**(6), 513–522, 2017.

[14] M. CAVALIERE, O.H. IBARRA, G. PĂUN, O. EGECIOGLU, M. IONESCU, S. WOODWORTH, *Asynchronous spiking neural P systems*. Theor. Comput. Sci. **410**(24-25), 2352–2364, 2009.

[15] H. CHEN, R. FREUND, M. IONESCU, G. PĂUN, M.J. PÉREZ-JIMÉNEZ, *On string languages generated by spiking neural P systems*. Fundam. Inform. **75**(1-4), 141–162, 2007.

[16] X. ZENG, L. XU, X. LIU, L. PAN, *On languages generated by spiking neural P systems with weights*. Inf. Sci. **278**, 423–433, 2014.

[17] T. WU, Z. ZHANG, L. PAN, *On languages generated by cell-like spiking neural P systems*. IEEE Trans. Nanobiosci. **15**(5), 455–466, 2016.

[18] A. PĂUN, G. PĂUN, *Small universal spiking neural P systems*. BioSystems. **90**(1), 48–60, 2007.

[19] X. ZHANG, L. PAN, A. PĂUN, *On the universality of axon P systems*. IEEE T. Neur. Net. Lear. **26**(11), 2816–2829, 2015.

[20] F.G.C. CABARLE, H.N. ADORNA, M.J. PÉREZ-JIMÉNEZ, T. SONG, *Spiking neural P systems with structural plasticity*. Neural. Comput. Appl. **26**(8), 1905–1917, 2015.

[21] F.G.C. CABARLE, H.N. ADORNA, M. JIANG, X. ZENG, *Spiking neural P systems with scheduled synapses*. IEEE T. Nanobiosci. **16**(8), 792–801, 2017.

[22] T. WU, Z. ZHANG, G. PĂUN, L. PAN, *Cell-like spiking neural P systems*. Theor. Comput. Sci. **623**, 180–189, 2016.

[23] H. PENG, J. WANG, M.J. PÉREZ-JIMÉNEZ, H. WANG, J. SHAO, T. WANG, *Fuzzy reasoning spiking neural P systems for fault diagnosis*. Inf. Sci. **235**, 106–116, 2013.

[24] T. WANG, G. ZHANG, J. ZHAO, Z. HE, J. WANG, M.J. PÉREZ-JIMÉNEZ, *Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural P systems*. IEEE Trans. Power Syst. **30**(3), 1182–1194, 2015.

[25] D. DÍAZ-PERNIL, F. PEÑA-CANTILLANA, M.A. GUTIÉRREZ-NARANJO, *A parallel algorithm for skeletonizing images by using spiking neural P systems*. Neurocomputing. **115**, 81–91, 2013.

[26] G. CIOBANU, D. PARASCHIV, *P system software simulator*. Fund. Inform. **49**, 61–66, 2002.

[27] C. BUIU, O. ARSENE, C. CIPU, M. PĂTRAŞCU, *A software tool for modeling and simulation of numerical P systems*. BioSystems. **103**(3), 442–447, 2011.

[28] D. DÍAZ-PERNIL, I. PÉREZ-HURTADO, M.J. PÉREZ-JIMÉNEZ, A. RISCOS-NÚÑEZ, *A P-Lingua programming environment for membrane computing*. In: Corne, D.W., Frisco, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) LNCS, vol. 5391, pp. 187-203. Springer, Heidelberg, 2009.

[29] M. GARCÍA-QUISMONDO, R. GUTIÉRREZ-ESCUDERO, M.A. MARTÍNEZ-DEL-AMOR, E. OREJUELA PINEDO, I. PÉREZ-HURTADO, *P-Lingua 2.0: A software framework for cell-like P systems*. Int. J. Comput. Commun. **4**(3), 234–243, 2009.

[30] L.F. MACÍAS-RAMOS, I. PÉREZ-HURTADO, M. GARCÍA-QUISMONDO, L. VALENCIA-CABRERA, M.J. PÉREZ-JIMÉNEZ, A. RISCOS-NÚÑEZ, *A P-Lingua based simulator for spiking neural P systems*. In Proc. 12th International Conference on Membrane Computing, Springer, Berlin, Heidelberg, 257–281, 2011.

[31] L. VALENCIA-CABRERA, T. WU, Z. ZHANG, L. PAN, M.J. PÉREZ-JIMÉNEZ, *A simulation software tool for cell-like spiking neural P systems*. Rom. J. Inform. Sci. Technol. **20**(1), 71–84, 2016.

[32] G. PĂUN, M.J. PÉREZ-JIMÉNEZ, G. ROZENBERG, *Spike trains in spiking neural P systems*. Int. J. Found. Comput. S. **17**(04), 975–1002, 2006.

[33] M.A. GUTIÉRREZ-NARANJO, M.J. PÉREZ-JIMÉNEZ, D. RAMÍREZ-MARTÍNEZ, *A software tool for verification of spiking neural P systems*. Nat. Comput. **7**(4), 485–497, 2008.