UNIVERSITY OF SEVILLE
Dpt. of Computer Science
and Artificial Intelligence

Developing efficient simulators for cell machines

A Thesis submitted for the degree of
Doctor of Philosophy
School of Computer Engineering
University of Seville

Luis Felipe Macías Ramos

Approval of the Thesis Supervisors

PhD. Mario de J. Pérez Jiménez PhD. Luis Valencia Cabrera

October 15, 2015

A mis padres,
por ser el sustento de mi vida.

Y amigos y compañeros,
por su apoyo incondicional.

Contents

List of Figures

List of Tables

Motivation

This dissertation is comprised within the field of *Natural Computing*, a broad discipline inspired by dynamic processes that occur in nature and that are subject to be interpreted as calculation procedures. Inside this discipline, models and computational techniques are researched, with the ultimate goal to better understand the world around us, in terms of information processing.

Membrane Computing [157] is an emerging branch within this area, initiated by Gheorghe Păun at the end of 1998 [148]. It starts from the assumption that the processes taking place in the compartmental structure of a living cell can be interpreted as computations. Devices of this machine-oriented computing paradigm are called *P systems*. They have relatively simple *syntactic* ingredients: a *membrane structure* consisting of a hierarchical arrangement of membranes embedded in a *skin* membrane, and delimiting *regions* or compartments where multisets of objects and sets of evolution rules are placed. P systems have also two main *semantic* ingredients: their inherent *parallelism* and *non-determinism*. The objects inside the membranes can evolve according to given rules in a synchronous (in the sense that a global clock is assumed), parallel, and non-deterministic way.

It is worthy to note that we have here a *double parallelism*, once at the level of regions (the rules are used in a parallel way), and once at the level of the system (all regions evolve concurrently). Is this parallelism and non-determinism able to solve computationally hard problems in a "feasible" time? The answer is affirmative [71], but we must point out two considerations. On the one hand, we have to deal with the non-determinism in such a way that the classical notion of acceptance is not a true algorithmic concept [49]. On the other hand, the drastic decrease of the execution time from an exponential to a polynomial one is not achieved for free, but by the use of an exponential workspace (in the form of membranes and objects), although this space is created in polynomial (often linear) time. Nevertheless, designing solutions to computationally hard problems by means of families of P systems often yields stablishing borderlines of tractability (in terms of the complexity classes theory within membrane computing), and moreover can provide new ways to tackle the **P** versus **NP** problem.

Although initially most of the research in P systems concentrated on theoretical results about the computational power and efficiency of the devices involved, lately an increasing attention is paid to applicatications. More precisely, to *model* biological phenomena within the framework of *Computational Systems Biology* and *Population Dynamics*. In this case, P systems are not used as a computing paradigm, but rather as a formalism for describing the behavior of the system to be modeled. They offer an approach to the development of models for biological systems that meets the requirements of a good modeling framework: relevance, understandability, extensibility and computational / mathematical tractability. In this respect, several P systems models have been proposed to describe, for example: *signal transduction* [26], *gene regulation control* [161], or *quorum sensing* (by means of *multicompartmental Gillespie Algorithm*) [160]. These models differ from each other in the type of the rewriting rules, membrane structure and the strategy applied to run the rules in the compartments defined by membranes. Furthermore, *probabilistic P systems* have also been successfully applied as a tool for macroscopic level processes, such as the computational modeling of real ecosystems [17, 29].

It is also worth mentioning promising applications regarding the modelling of *fault diagnosis problems in industrial systems*, that come from the hand of a very recent neural-like membrane system variant called Fuzzy Reasoning Spiking Neural Systems [130, 181, 205, 206, 175, 176, 102, 177, 183, 179, 180, 184, 185, 182].

In order to *experimentally validate* P systems based models, it is necessary to develop simulators able to be executed on electronic computers, which can help researchers to compute, analyze and extract results from a model [36]. These simulators have to be as *efficient* as possible to handle large size instances, what is one of the major challenges facing today's P systems simulators. In this regard, software applications for Membrane Computing typically implement *sequential* (or with a limited parallelism) simulation algorithms adapted to conventional CPU architectures, so they lack the possibility of exploiting the massively parallel nature that P systems present by definition. This is necessary for obtaining a simulation model closer to the theoretical one.

This parallel computation model leads us to look for a massively-parallel technology where a parallel simulator can run more efficiently. The newest generations of *Graphics Processor Units (GPUs)* are *massively parallel processors* which can support several thousand of concurrent *threads*. To date, many general purpose applications have been migrated to these platforms obtaining good *speedups* compared to their corresponding sequential versions. Current NVIDIA GPUs, for example, contain thousands of scalar processing elements per chip, and they are programmed using a *C* programming language extension called *CUDA (Compute Unified Device Architecture)* [74, 202, 116].

This thesis aims at a double scientific contribution involving P systems. On the one hand, to provide new computational complexity frontiers in terms of syntactical ingredients of some P systems variants, namely cell-like P systems with symport/antiport rules with either membrane division or membrane separation rules. This contribution is complemented with the development of the corresponding simulation tools, within the well-known P–Lingua framework, serving as key assistants providing invaluable help in the design and formal verification tasks of solutions to decision problems defined in such variants. The second main contribution involves addressing the notable lack of software tools for assisting in the research on one of the most hot topics in Membrane Computing: neural-like P systems. In this way, applications to simulate this kind of systems have been developed for a wide range of Spiking Neural P systems variants, including an efficient simulation tool working on High Performance Computing platforms.

The source codes of the implemented simulators are available in the P–Lingua web page [199], under the GNU GPLv3 license.

Document structure

This document is structured in three parts, whose content is briefly outlined below.

- **Part I: Preliminaries**

  This part is devoted to present different concepts, notations and preliminary results that will be used throughout this work. The first chapter provides a short overview concerning to languages and multisets, graphs, Hamiltonian cycles, combinatorial optimization problems and decision problems.

  In Chapter 2, the disciplines of Natural Computing and Membrane Computing are introduced. Moreover, formal concepts related to the syntactic and semantics components of P systems are provided. Cell-like P systems and tissue-like P systems are described and some relevant features and results related to computational complexity are presented. Chapter

3 is devoted to the contribution of Membrane Computing to tackle the **P** versus **NP** problem by means of cell-like or tissue-like models. The state of the art concerning to spiking neural P systems is summarized in Chapter 4, and a variant (fuzzy reasoning spiking neural P systems with real numbers) with interesting real applications is described in Chapter 5.

This first part ends with Chapter 6, which describes a chronological overview of the P-Lingua simulation framework, which is the starting point of an important part the described work herein.

- **Part II: Contributions**

  This part contains the most significant contributions of this work. They are structured in four chapters. In Chapter 7, different results about frontiers of the tractability in terms of syntactical ingredients of cell-like P systems with symport/antiport rules are presented. Chapter 8 is devoted to developing simulators of cell-like P systems with symport/antiport rules in the context of P-Lingua. In Chapter 9, new extensions of spiking neural P systems are introduced and P-Lingua based software are developed. This part ends by presenting a simulator of fuzzy reasoning spiking neural P systems with real numbers.

- **Part III: Conclusions**

  The document concludes with a chapter devoted to the presentation of conclusions and suggestions of future research directions.

Results

It is worth to note the following original contributions of the work described in this document:

- *New frontiers of the efficiency in terms of cell-like P systems with symport/antiport rules.*

  In the framework of of cell-like P systems with symport/antiport rules, membrane division rules and membrane separation rules have been considered in order to generate an exponential amount of space in linear time. Specifically, new (optimal) frontiers of the efficiency have been obtained in terms of the length of communication rules: (a) when membrane division rules are allowed, passing from one (non-cooperation) to two (minimal cooperation) amounts to passing from non-efficiency to efficiency; and (b) when membrane separation rules are allowed, passing from two to three amounts to passing from non-efficiency to efficiency. Moreover, the role of the environment (from a computational complexity point of view) in this class of P systems has been studied and similar results to tissue P systems with symport/antiport rules have been obtained.

- *P-Lingua based software for cell-like P systems with symport/antiport rules.*

  The hard tasks of designing families solving **NP**-hard problems in polynomial time within the framework of membrane Computing and the formal verification of these solutions, requires a very important assistant. In this context, P–Lingua simulators of cell-like P systems with symport/antiport rules with either membrane division or membrane separation rules and allowing arbitrary object copies in the environment, have been developed. With respect to the families designs, virtual experiments on the solutions provided in this work of both **HAM-CYCLE** and **SAT** problem, have been implemented. With respect to the formal verification, the simulator has been used to check that the identified invariant formulas were corroborated in the corresponding configurations.

- *New variants of Spiking neural P systems and development of simulators.*

  An extension of the Spiking neural P system framework, within the Membrane Computing paradigm, has been provided, involving both theoretical (variants, simulation algorithms) and practical (simulation tools) aspects. Such extension involves producing a general purpose SN P systems simulator, which has been included into pLinguaCore library. In particular, P–Lingua language syntax has been extended to incorporate new features related to SN P systems. Such features include, among others, different simulation modes, an initial configuration specification, consisting of set of neurons and a collection of synapses, and the possibility of defining a simulation input spike train and input and output neurons.

- *Development of efficient simulators for Fuzzy Reasoning Spiking neural P systems*

  Fuzzy Reasoning Spiking neural P systems (operating with real numbers) incorporate fuzzy logic elements, as they are intended to model the fuzzy diagnosis knowledge and reasoning associated to tackling real-life problems involving uncertain knowledge. In this sense, FRSN P systems have shown promising applications in the engineering field, addressing problems like fault diagnosis in electric power systems. Due to the potential interesting applications related to these systems, key goal of the work object of this dissertation has been providing a first public parallel simulation tool for FRSN P systems, with a first version of such tool successfully simulating rFRSN P systems instances on CUDA-enabled devices. A hybrid sequential/parallel simulator included into pLingua-Core library which, while externally behaving as any other existing (sequential) simulator in the library, has been developed with the ability of making calls to native CUDA kernels executed on an underlying GPU architecture.

Publications

Some of the original works have been published in different journals, book chapters or proceedings of international conferences. Next, we present a list of the most relevant publications.

Indexed Journals

1. L.F. Macías-Ramos, B. Song, L. Valencia-Cabrera, L. Pan, M.J. Pérez-Jiménez. Membrane fission: A computational complexity perspective. *Complexity*, online version 2015 (doi: 10.1002/cplx.21691).

2. L.F. Macías-Ramos, M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera. Membrane fission versus cell division: when membrane proliferation is not enough. *Theoretical Computer Science*, online version 2015 (doi: 10.1016/j.tcs.2015.06.025).

3. L.F. Macías-Ramos, L. Valencia-Cabrera, B. Song, T. Song, L. Pan, M.J. Pérez-Jiménez. A P–Lingua Based Simulator for P Systems with Symport/Antiport Rules. *Fundamenta Informaticae*, 139, 2 (2015), 211-227 (doi: 10.3233/FI-2015-1232).

4. M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. *Fundamenta Informaticae*, 136, 3 (2015), 269-284 (doi: 10.3233/FI-2015-1157).

5. L.F. Macías-Ramos, M.J. Pérez-Jiménez, T. Song, L. Pan. Extending Simulation of Asynchronous Spiking Neural P Systems in P–Lingua. *Fundamenta Informaticae*, 136, 3 (2015), 253-267 (doi: 10.3233/FI-2015-1156).

6. T. Song, L.F. Macías-Ramos, L. Pan, M.J. Pérez-Jiménez. Time-free solution to SAT problem using P systems with active membranes. *Theoretical Computer Science*, 529 (2014), 61-68. (doi: 10.1016/j.tcs.2013.11.014).

Non-indexed Journals

1. C. Graciani, L.F. Macías-Ramos, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, A. Riscos, A. Romero, L. Valencia. Păun's conjecture beyond polarizations: Alternative formulations. *Analele Universitatii Bucuresti - Seria Informatica*, Anul LXII, 2 (2015), 47-60.

2. L.F. Macías-Ramos, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, A. Riscos, L. Valencia. The Role of the Direction in Tissue P Systems with Cell Separation. *Journal of Automata, Languages and Combinatorics*, 19, 1-4 (2014), 185-199.

Chapters of books

1. M.A. Martínez-del-Amor, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez . Accelerated Simulation of P Systems on the GPU: A Survey. In L. Pan, Gh. Paun, M. J. Pérez-Jiménez, T. Song (eds.) *Bio-inspired Computing: Theories and Applications.* Series: Communications in Computer and Information Science, Volume 472, 2014, pp. 308-312. (doi: 10.1007/978-3-662-45049-9).

2. L.F. Macías-Ramos, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera. Unconventional approaches to tackle the P versus NP problem. In Gh. Paun, Gr. Rozenberg, A. Salomaa (eds.) *Discrete Mathematics and Computer Science*, Editura Academiei Romane (The Publishing House of the Romanian Academy), Bucuresti, Romania, ISBN 978-973-27-2470-5, 2014, pp. 223-238.

3. M. A. Colomer, M. García-Quismondo, L. F. Macías-Ramos, M. A. Martínez-del-Amor, I. Perez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez, L. Valencia-Cabrera . Membrane System-Based Models for Specifying Dynamical Population Systems. In P. Frisco, M. Gheorghe, M. J.

Pérez-Jiménez (eds.) *Applications of Membrane Computing in Systems and Synthetic Biology.* Springer, Series: Emergence, Complexity and Computation, Vol. 7, 2014, Chapter 4, pp. 97-132 (doi: 10.1007/978-3-319-03191-0-4).

4. M. García-Quismondo, L.F. Macías-Ramos, M.J. Pérez-Jiménez. Implementing enzymatic numerical P systems for AI applications by means of graphic processing units. In J. Kelemen, J. Romportl and E. Zackova (eds.) *Beyond Artificial Intelligence. Contemplations, Expectations, Applications*, Springer, Berlin-Heidelberg, Series: Topics in Intelligent Engineering and Informatics, Volume 4, 2013, chapter XIV, pp. 137-159.

International Conferences

1. L.F. Macías-Ramos, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez. Simulating FRSN P systems with real numbers in P-Lingua on sequential and CUDA platforms. In J.M. Sempere and C. Zandron (eds) *Proceedings of the 16th International Conference on Membrane Computing (CMC16)*, 17-21 August, 2015, Valencia, Spain, pp. 227-241

2. M.A. Martínez-del-Amor, L.F. Macías-Ramos, M.J. Pérez-Jiménez. Parallel simulation of PDP systems: Update and roadmap. In L.F. Macías-Ramos, Gh. Paun, A. Riscos, L. Valencia (eds) *Proceedings of the Thirteenth Brainstorming Week on Membrane Computing*, 2-6 February, 2015, Sevilla, Spain, pp. 227-243.

3. L. Valencia-Cabrera, B. Song, L.F. Macías-Ramos, L. Pan, A. Riscos-Núñez, M.J. Pérez-Jiménez. Minimal cooperation in P systems with symport/antiport: A complexity approach. In L.F. Macías-Ramos, Gh. Paun, A. Riscos, L. Valencia (eds) *Proceedings of the Thirteenth Brainstorming Week on Membrane Computing*, 2-6 February, 2015, Sevilla, Spain, pp. 301-323.

4. L. Valencia-Cabrera, B. Song, L.F. Macías-Ramos, L. Pan, A. Riscos-Núñez, M.J. Pérez-Jiménez. Computational efficiency of P systems with symport/antiport rules and membrane separation. In L.F. Macías-Ramos, Gh. Paun, A. Riscos, L. Valencia (eds) *Proceedings of the Thirteenth Brainstorming Week on Membrane Computing*, 2-6 February, 2015, Sevilla, Spain, pp. 325-370.

5. L.F. Macías-Ramos, T. Song, L. Pan, M.J. Pérez-Jiménez. Extending SNP systems asynchronous simulation modes in P lingua. In L.F. Macías-Ramos, M.A. Martínez-del-Amor, Gh. Paun, A. Riscos-Núñez, L. Valencia-Cabrera (eds.) *Proceedings of the Twelfth Brainstorming Week on Membrane Computing*, Seville, Spain, February 3-7, 2014, Report RGNC 01/2014, Fénix Editora, 2014, pp. 261-280.

6. M.A. Martínez, I. Pérez, M. García, L.F. Macías-Ramos, L. Valencia, A. Romero, C. Graciani, A. Riscos, M.A. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating population dynamics P systems with proportional objects distribution. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (eds.) Membrane Computing- 13th International Conference CMC 2012 Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. *Lecture Notes in Computer Science*, 7762 (2013), 257-276

7. L.F. Macías-Ramos, M.J. Pérez-Jiménez, A. Riscos, M. Rius. The efficiency of tissue P systems with cell separation relies on the environment. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (eds.) Membrane Computing- 13th International Conference CMC 2012 Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. *Lecture Notes in Computer Science*, 7762 (2013), 243-256.

8. L.F. Macías-Ramos, M.J. Pérez-Jiménez. Spiking Neural P systems with functional astrocytes. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (eds.) Membrane Computing- 13th International Conference CMC 2012 Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. *Lecture Notes in Computer Science, 7762 (2013), 228-242.*

9. L.F. Macías-Ramos, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia, M.J. Pérez-Jiménez, A. Riscos. A P-Lingua based simulator for Spiking Neural P systems. Membrane Computing, 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers. *Lecture Notes in Computer Science*, 7184 (2012), 257-281.

10. L.F. Macías-Ramos, M.J. Pérez-Jiménez. On recent developments in P-lingua based simulators for Spiking Neural P Systems. L. Pan, Gh. Paun, T. Song (eds.) *Pre-proceedings of Asian Conference on Membrane*

Computing (ACMC 2012), Huazhong University of Science and Technology, Wuhan, China, October 15-18, 2012, pp. 14-29.

11. M.A. Martínez del Amor, I. Pérez-Hurtado, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia, A. Romero, C. Graciani, A. Riscos, M.A. Colomer, M.J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P systems with proportional object distribution. In M. García-Quismondo, L.F. Macías-Ramos, Gh. Paun, I. Pérez Hurtado, L. Valencia-Cabrera (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing, Volume II*, Seville, Spain, January 30- February 3, 2012, Report RGNC 01/2012, Fénix Editora, 2012, pp. 27-56.

Part I

Preliminaries

1 Technical prerequisites

This chapter is devoted to present different concepts, notations and preliminary results that will be used throughout this work.

1.1 Languages and Multisets

An *alphabet* Γ is a non-empty set and their elements are called *symbols*. A *string* u over Γ is a mapping from a natural number $n \in \mathbb{N}$ onto Γ. Number n is called *length* of the string u and it is denoted by $|u|$. The empty string (with length 0) is denoted by λ. A *language* over Γ is a set of strings over Γ.

The *Parikh vector* associated with a string $u \in \Sigma^*$ with respect to the alphabet $\Sigma = \{a_1, \dots, a_r\}$ is $\Psi_\Sigma(u) = (|u|_{a_1}, \dots, |u|_{a_r})$, where $|u|_{a_i}$ denotes the number of ocurrences of the symbol a_i in the string u. This is called the *Parikh mapping* associated with Σ. Notice that in this definition the ordering of the symbols from Σ is relevant. If $\Sigma_1 = \{a_{i_1}, \dots, a_{i_s}\} \subseteq \Sigma$ then we define $\Psi_{\Sigma_1}(u) = (|u|_{a_{i_1}}, \dots, |u|_{a_{i_s}})$, for each $u \in \Sigma^*$.

A *multiset* over an alphabet Γ is an ordered pair (Γ, f), where f is a mapping from Γ onto the set of natural numbers $\mathbb{N}$. For each $x \in \Gamma$ we say that $f(x)$ is the *multiplicity* of x in that multiset. The *support* of a multiset $m = (\Gamma, f)$ is defined as *supp*$(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite if its support is a finite set. We denote by $\emptyset$ the empty multiset. Let us note that a set is a particular case of a multiset when each symbol of the support has multiplicity 1.

Let $m_1 = (\Gamma, f_1)$, $m_2 = (\Gamma, f_2)$ be multisets over Γ, then the union of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. We say that m_1 is contained in m_2 and we denote it by $m_1 \subseteq m_2$, if $f_1(x) \leq f_2(x)$ for each $x \in \Gamma$. The relative complement of m_2 in m_1, denoted by $m_1 \setminus m_2$, is the multiset (Γ, g), where $g(x) = f_1(x) - f_2(x)$ if $f_1(x) \geq f_2(x)$, and $g(x) = 0$ otherwise.

1.2 Graphs and Hamiltonian cycles

A *rooted tree* is a connected, acyclic, undirected graph in which one of the vertices (called *the root of the tree*) is distinguished from the others. Given a node x (different from the root) in a rooted tree, if the last edge on the (unique) path from the root to node x is $\{x, y\}$ (so $x \neq y$), then y is **the** *parent* of node x and x is **a** *child* of node y. We denote it by $y = p(x)$ and $x \in ch(y)$. The root is the only node in the tree with no parent. A node with no children is called a *leaf* (see [32] for details).

Let $G = (V, E)$ be a directed graph, where $V = \{1, \dots, n\}$ and the set of arcs is $E = \{(u_1, v_1), \dots, (u_m, v_m)\} \subset V \times V$. We say that a finite sequence $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}, u_{\alpha_{r+1}})$ of nodes of G is a *simple path of G of length* $r \geq 1$ if the following holds:

- $\forall i \ (1 \leq i \leq r \rightarrow (u_{\alpha_i}, u_{\alpha_{i+1}}) \in E)$.
- $|\{u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}\}| = r$.

If $u_{\alpha_{r+1}} \notin \{u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}\}$, then we say that γ is a simple path of length r from u_{α_1} to $u_{\alpha_{r+1}}$. If $u_{\alpha_{r+1}} = u_{\alpha_1}$ and $r \geq 2$, then we say that γ is a *simple cycle* of length r.

A *Hamiltonian path* of G from $a \in V$ to $b \in V$ $(a \neq b)$ is a simple path $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}, u_{\alpha_{r+1}})$ from a to b such that $a = u_{\alpha_1}$, $b = u_{\alpha_{r+1}}$, and $V = \{u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}, u_{\alpha_{r+1}}\}$. A *Hamiltonian cycle* of G is a simple cycle $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}, u_{\alpha_{r+1}})$ of G such that $V = \{u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}\}$.

If $\gamma = (u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_r}, u_{\alpha_{r+1}})$ is a simple path of G then we also denote it by the set $\{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2, \dots, (u_{\alpha_r}, u_{\alpha_{r+1}})_r\}$. That is, $(u_{\alpha_k}, u_{\alpha_{k+1}})_k$ can be interpreted as the k-th arc of the path γ, for each k $(1 \leq k \leq r)$.

Given a directed graph $G = (V, E)$, throughout this paper we denote

$$\begin{array}{rcl} A_G & = & \{(i,j)_k \mid i,j,k \in \{1,\dots,n\} \ \wedge \ (i,j) \in E\} \\ A'_G & = & \{(i,j)'_k \mid i,j,k \in \{1,\dots,n\} \ \wedge \ (i,j) \in E\} \\ A''_G & = & \{(i,j)''_k \mid i,j,k \in \{1,\dots,n\} \ \wedge \ (i,j) \in E\} \end{array}$$

Proposition 1.1. *Let $G = (V, E)$ be a directed graph. Let $V = \{1,\dots,n\}$ and $A_G = \{(i,j)_k| \ i,j,k \in \{1,\dots,n\} \ \wedge \ (i,j) \in E\}$. If $B \subseteq A_G$ then the following assertions are equivalent:*

1. *B is a Hamiltonian cycle.*
2. *$|B| = n$ and the following holds: for each $i, i', j, j', k, k' \in \{1,\dots,n\}$,*
   (*a*) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \to k \neq k']$
   (*b*) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \to i \neq i']$
   (*c*) $[(i,j)_k \in B \wedge (i',j')_{k'} \in B \wedge (i,j)_k \neq (i',j')_{k'} \to j \neq j']$
   (*d*) $[(i,j)_k \in B \wedge (i',j')_{k+1} \in B \to j = i']$

Proof: Let $B = \{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2 \dots, (u_{\alpha_m}, u_{\alpha_{r+1}})_n\}$ be a Hamiltonian cycle of G. Then, $|B| = n$ and conditions (a), (b), (c) and (d) from (2) hold.

Let $B \subseteq A_G$ such that $|B| = n$ and conditions (a), (b), (c) and (d) from (2) hold. Then, from (a) the set B must to be of the form

$$B = \{(u_{\alpha_1}, v_{\alpha_1})_1, (u_{\alpha_2}, v_{\alpha_2})_2 \dots, (u_{\alpha_n}, v_{\alpha_n})_n\}$$

where:

- From (d) we deduce that $\forall s \ (1 \leq s \leq n-1 \to v_{\alpha_s} = u_{\alpha_{s+1}})$.
- From (b) we have $V = \{u_{\alpha_1}, u_{\alpha_2}, \dots, u_{\alpha_n}\}$.

Finally, on the one hand we have $v_{\alpha_n} \in \{u_{\alpha_1}, u_{\alpha_2} \dots, u_{\alpha_n}\}$. On the other hand, by condition (c) we deduce that $v_{\alpha_n} \notin \{v_{\alpha_1}, \dots, v_{\alpha_{n-1}}\} = \{u_{\alpha_2}, \dots, u_{\alpha_n}\}$. Thus $v_{\alpha_n} = u_{\alpha_1}$.

□

Remark 1: Let $B \subseteq A_G$ be a Hamiltonian cycle of G. For each $i, i', j, j', k, k' \in \{1,\dots,n\}$ the following holds:

1. If $(i,j)_k \in B$ and $j \neq j'$ then $(i,j')_{k'} \notin B$.
2. If $(i,j)_k \in B$ and $i \neq i'$ then $(i',j)_{k'} \notin B$.
3. If $(i,j)_k \in B$ and $(i,j) \neq (i',j')$ then $(i',j')_k \notin B$.

4. If $(i, j)_k \in B$ and $(i', j')_{k+1} \in B$ then $j = i'$.

Remark 2: Let us notice that if $(u_{\alpha_1}, u_{\alpha_2}, \ldots, u_{\alpha_n}, u_{\alpha_1})$ is a Hamiltonian cycle of G of length n, then we can describe it by the following subset of A_G:

$$B_1 = \{(u_{\alpha_1}, u_{\alpha_2})_1, (u_{\alpha_2}, u_{\alpha_3})_2, \ldots, (u_{\alpha_n}, u_{\alpha_1})_n\}$$

But $(u_{\alpha_2}, u_{\alpha_3}, \ldots, u_{\alpha_m}, u_{\alpha_1}, u_{\alpha_2})$ also represents the same Hamiltonian cycle. It can be described as follows: $B_2 = \{(u_{\alpha_2}, u_{\alpha_3})_1, (u_{\alpha_3}, u_{\alpha_4})_2, \ldots, (u_{\alpha_1}, u_{\alpha_2})_n\}$. Thus, given a Hamiltonian cycle γ of G, there are exactly n different subsets of A_G codifying that cycle.
Remark 3: Let us suppose that the total number of Hamiltonian cycles of G is q. Then, the number of different subsets B of A_G verifying conditions (a), (b), (c), and (d) from Proposition 1.1 is exactly $n \cdot q$.

1.3 Encoding ordered pairs of natural numbers

The *pair function* $\langle n, m\rangle = ((n+m)(n+m+1)/2) + n$ is a polynomial–time computable function from $\mathbb{N} \times \mathbb{N}$ onto $\mathbb{N}$ which is also a primitive recursive and bijective function.
Notation: Given a sequence $x_1, \ldots, x_n$ of symbols, we write $x_1, \ldots \widehat{x_k}, \ldots, x_n$ to express that term x_k does not appear in that sequence.

1.4 Combinatorial Optimization Problems

Roughly speaking, when we deal with *combinatorial optimization problems* we wish to find the *best* solution (according to a given criterion) among a class of possible (candidate or feasible) solutions. That is, in this kind of problems there can be many possible solutions, each one has associated a value (a positive rational number), and we aim to find a solution with the optimal (minimum or maximum) value.

For example, a *vertex cover* of an undirected graph is a set of vertices such that any edge of the graph has, at least, an endpoint in that set. Then, we may want to find one of the smallest vertex covers among all possible vertex covers in the input graph. This is the combinatorial optimization problem called *Minimum Vertex Cover Problem.*

Definition 1.1. *A combinatorial optimization problem, X, is a tuple (I_X, s_X, f_X) where:*

- *I_X is a language over a finite alphabet.*
- *s_X is a function whose domain is I_X and, for each $a \in I_X$, the set $s_X(a)$ is finite.*
- *f_X is a function (the objective function) that assigns to each instance $a \in I_X$ and each $c_a \in s_X(a)$ a positive rational number $f_X(a, c_a)$.*

The elements of I_X are called *instances* of the problem X. For each instance $a \in I_X$, the elements of the finite set $s_X(a)$ are called *candidate* (or *feasible*) *solutions* associated with the instance a of the problem. For each instance $a \in I_X$ and each $c_a \in s_X(a)$, the positive rational number $f_X(a, c_a)$ is called *solution value* for c_a. The function f_X provides the criterion to determine the *best* solution.

For example, the *Minimum Vertex Cover* problem is a combinatorial optimization problem such that I_X is the set of all undirected graphs, and for each undirected graph G, $s_X(G)$ is the set of all vertex covers of G; that is, each vertex cover of the graph is a candidate solution for the problem. The objective function f_X is defined as follows: for each undirected graph G and each vertex cover C of G, $f_X(G, C)$ is the cardinality of C.

Definition 1.2. *Let $X = (I_X, s_X, f_X)$ be a combinatorial optimization problem. An optimal solution for an instance $a \in I_X$ is a candidate solution $c \in s_X(a)$ associated with this instance such that,*

- *either for all $c' \in s_X(a)$ we have $f_X(a, c) \leq f_X(a, c')$ (and then we say that c is a minimal solution for a),*
- *either for all $c' \in s_X(a)$ we have $f_X(a, c) \geq f_X(a, c')$ (and then we say that c is a maximal solution for a).*

A *minimization* (respectively, *maximization*) *problem* is a combinatorial optimization problem such that each optimal solution is a minimal (respectively, maximal) solution.

That is, an optimization problem seeks the best of all possible candidate solutions, according to a simple cost criterion given by the objective function. For example, the *Minimum Vertex Cover* problem is a minimization problem because a minimal solution associated with an undirected graph G, provides one of the smallest vertex covers of G.

An *approximation computational device*, $\mathcal{D}$, for a combinatorial optimization problem, X, provides a candidate solution $c \in s_X(a)$ for each instance

$a \in I_X$. If the provided solution is always optimal, then $\mathcal{D}$ is called an *optimization computational device* for X.

For instance, an approximation machine for the *Minimum Vertex Cover* problem needs only find some vertex cover associated with each undirected graph, whereas an optimization machine must always find a vertex cover with the least cardinality associated with each undirected graph.

Having in mind that until now polynomial time optimization algorithms have not be found for many presumably intractable problems (it is believed that this kind of solutions can never be found), it is convenient to find an approximation algorithm running in polynomial time and such that, for all problem instances the candidate solution given by the algorithm is *close*, in a sense, to an optimal solution.

1.5 Decision Problems

An important class of combinatorial optimization problems is the class of decision problems, that is, problems that require either an *yes* or a *no* answer.

Definition 1.3. *A decision problem, X, is a pair (I_X, θ_X) such that I_X is a language over a finite alphabet (whose elements are called instances) and θ_X is a total boolean function (that is, a predicate) over I_X.*

Therefore, a decision problem $X = (I_X, \theta_X)$ can be viewed as a combinatorial optimization problem $X = (I_X, s_X, f_X)$ where for each instance $a \in I_X$ we have the following:

- $s_X(a) = \{\theta_X(a)\}$ (the only possible candidate solution associated with instance a is 0 or 1, depending on the answer of the problem to a).
- $f_X(a, \theta_X(a)) = 1$.

Thus, each decision problem can be considered as a minimization (or maximization) problem.

There exists a natural correspondence between languages and decision problems in the following way. Each language L, over an alphabet Σ, has a decision problem, X_L, associated with it as follows: $I_{X_L} = \Sigma^*$, and $\theta_{X_L} = \{(x, 1) \mid x \in L\} \cup \{(x, 0) \mid x \notin L\}$; reciprocally, given a decision problem $X = (I_X, \theta_X)$, the language L_X over the alphabet of I_X corresponding to it is defined as follows: $L_X = \{a \in I_X \mid \theta_X(a) = 1\}$.

Usually, NP-completeness has been studied in the framework of decision problems. Many abstract problems are not decision problems, but combinatorial optimization problems, in which some value must be optimized (minimized or maximized). In order to apply the theory of NP-completeness to combinatorial optimization problems, we must consider them as decision problems.

We can transform any combinatorial optimization problem into a roughly equivalent decision problem by supplying a target/threshold value for the quantity to be optimized, and asking the question whether this value can be attained. Next we give two examples.

1. The *Minimum Vertex Cover Problem.*

   Optimization version: Given an undirected graph G, find the cardinality of a *minimal* set of a vertex cover of G.

   Decision version: Given an undirected graph G, and *a positive integer* k, determine whether or not G has a vertex cover of size *at most* k.

2. The *Common Algorithmic Problem* [62].

   Optimization version: given a finite set S and a family F of subsets of S, find the cardinality of a *maximal* subset of S which does not include any set belonging to F.

   Decision version: given a finite set S, a family F of subsets of S, and a positive integer k, we are asked whether there is a subset A of S such that the cardinality of A is *at least* k, and which does not include any set belonging to F.

If a combinatorial optimization problem can be *quickly* solved, then its decision version can be quickly solved as well (because we only need to compare the solution value with a threshold value). Similarly, if we can make clear that a decision problem is hard, we also make clear that its associated combinatorial optimization problem is hard.

For example, let A be a polynomial time algorithm for the optimization version of the Minimum Vertex Cover problem. Then we consider the following polynomial time algorithm for the decision version of the Minimum Vertex Cover problem: given an undirected graph G, and a positive integer k, if $k < A(G)$ (here $A(G)$ is the cardinality of a smallest vertex cover of G), then answer *no*; otherwise, the answer is *yes*.

Reciprocally, let B be a polynomial time algorithm for the decision version of the Minimum Vertex Cover problem. Then we consider the following polynomial time algorithm for the optimization version of the Minimum Vertex

Cover problem: given an undirected graph G, repeatedly while $k \leq$ number of vertices of G (starting from $k = 0$, and in the next step considering $k + 1$) we execute the algorithm A on input (G, k), until we reach a first *yes* answer, and then the result is k.

1.5.1 Solving Decision Problems

Recall that, in a natural way, each decision problems has associated a language over a finite alphabet. Next, we define the solvability of decision problems through the recognition of languages associated with them.

In order to specify the concept of solvability we work with an universal computing model: Turing machines.

Let M be a Turing machine such that the result of any halting computation is *yes* or *no*. Let L be a language over an alphabet Σ.

If M is a *deterministic* device (with Σ as working alphabet), then we say that M *recognizes* or *decides* L whenever, for any string a over Σ, if $a \in L$, then the answer of M on input a is *yes* (that is, M accepts a), and the answer is *no* otherwise (that is, M reject a).

If M is a *non-deterministic* Turing machine, then we say that M *recognizes* or *decides* L if the following is true: for any string a over Σ, $a \in L$ if and only if there exists a computation of M with input a such that the answer is *yes*. That is, an input string a is accepted by M if there is *an* accepting computation of M on input a. But now we do not have a mechanical criterion to reject an input string.

Recall that any deterministic Turing machine with multiple tapes can be simulated by a deterministic Turing machine with one tape with a polynomial loss of efficiency, whereas the simulation of non-determinism through determinism involves an exponential loss of efficiency.

In the context of computation theory, we consider a problem X to be solved when we have a *general* (definite) *method* (described in a model of computation) that works for any instance of the problem. From a practical point of view, such methods only run over a finite set of instances whose sizes depend on the available resources.

We say that a Turing machine M solves a decision problem X if M *recognizes* the language associated with X; that is, for any instance a of the problem: (1) in the deterministic case, the machine (with input a) output *yes* if the answer of the problem is *yes*, and the output is *no* otherwise; (2) in the non-deterministic case, some computation of the machine (with input a) output *yes* if the answer of the problem is *yes*.

Due to the fact that we represent the instances of abstract problems as strings we can consider their size in a natural manner: the size of an instance is the length of the string. Then, how do the resources required to execute a method increase according to the size of the instance? This is a relevant question in computational complexity theory.

2
Membrane Computing

In the last years several computing models using powerful tools from nature have been developed (because of this, they are known as *bio-inspired* models) and several solutions in polynomial time to hard problems from the class **NP** have been presented, making use of non-determinism and/or of an exponential amount of space. This is the reason why a practical implementation of such models (in biological, electronic, or other media) could provide a significant advance in the resolution of computationally hard problems.

Membrane Computing, introduced by Gh. Păun at the end of 1998, is a relatively young branch of Natural Computing providing distributed parallel computing models whose computational devices are called *membrane systems*. These systems are inspired by some basic biological features, specifically by the structure and functioning of the living cells, as well as from the way the cells are organized in tissues, organs, and organisms.

There are basically three ways to consider computational devices: cell–like membrane systems, tissue–like membrane systems and spiking neural–like membrane systems. The first one ([148]), using the membranes arranged hierarchically, inspired from the structure of the cell, the second one ([92, 93]) using the membranes placed in the nodes of a directed graph, inspired from the cell inter–communication in tissues, and the third one ([67]) is inspired by the neurophysiological behavior of neurons sending electrical impulses (spikes) along axons to other neurons.

Many variants of all these systems were considered; overview of the domain can be found in [151] and [157], with up-to-date information available at the membrane computing website [200]. Friendly introduction to membrane computing is [156] as well as the first chapter of [157].

This chapter addresses how hard abstract decision problems could be solved efficiently using membrane systems. Moreover, the notions from classical *computational complexity theory* are adapted for the membrane computing framework. Let us note that the purpose of computational complexity theory is to provide bounds on the amount of resources necessary for any mechanical procedure (*algorithm*) that solves a problem.

2.1 Cell-like P systems

In the structure and functioning of a cell, biological *membranes* play an essential role. The cell is separated from its environment by means of a *skin membrane*, and it is internally compartmentalized by means of *internal membranes*. The class of cell-like P systems (introduced in [148]) use the biological membranes arranged hierarchically, inspired from the structure of the cell.

Definition 2.1. *A basic transition P system of degree* $q \geq 1$ *is a tuple of the form* $\Pi = (\Gamma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})$*, where:*

- Γ *is a finite alphabet;*
- μ *is a rooted tree whose nodes are injectively labelled with* $1, \dots, q$*;*
- $\mathcal{M}_1, \dots, \mathcal{M}_q$ *are finite multisets over* Γ*;*
- $\mathcal{R}_i$*,* $1 \leq i \leq q$*, are finite sets of rules over* Γ *of the form* $u \to v$*, where* $v = (v_1, here)(v_2, out), (v_3, in_j)\, \delta^*$*, being* u, v_1, v_2, v_3 *finite multisets over* Γ*,* $1 \leq j \leq q$ *and* $\delta^* \in \{\lambda, \delta\}$*;*
- $i_{out} \in \{0, 1, \dots, q\}$*.*

A basic transition P system $(\Gamma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})$ of degree $q \leq 1$ can be viewed as a set of q membranes, injectively labelled with elements in $\{1, \dots, q\}$, arranged in a hierarchical membrane structure μ given by a rooted tree whose root is called the *skin membrane*, labelled by 1, and delimiting *regions* (space bounded by a membrane and the immediately lower membranes) such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ represent the finite multisets of *objects* (symbols of Γ) initially placed in the q regions of the system; (b) $\mathcal{R}_1, \dots, \mathcal{R}_q$

are finite sets of rules over Γ associated with the q; (c) $i_{out} \in \{0, 1, \ldots, q\}$ is a label that represents a distinguished *zone* called the *output zone*; (d) 0 is a label that represents the *environment* of the system where the result is encoded. We use the term *zone* i $(0 \leq i \leq q)$ to refer to membrane i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$.

For each membrane $i \in \{2, \ldots, q\}$ (different from the skin membrane) we denote by $p(i)$ the parent of membrane i in the rooted tree μ. We define $p(1) = 0$, that is, by convention the "parent" of the skin membrane is the environment. The leaves of the rooted tree are called *elementary membranes*.

An *instantaneous description* or a *configuration* $\mathcal{C}_t$ at an instant t of a P system is a tuple whose components are the membrane structure at instant t and all multisets of objects over Γ associated with all the membranes present in μ. The *initial configuration* of Π is $(\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$ is $(\mu, \mathcal{M}_1, \ldots, \mathcal{M}_q)$.

A rule of the form $u \rightarrow (v_1, here)(v_2, out)(v_3, in_j)\, \delta^* \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if the following holds: (a) membrane i is in $\mathcal{C}_t$; (b) multiset u is contained in the multiset associated with such membrane; (c) if multiset v_3 is a nonnempty set then $p(j) = i$; and (d) if $\delta^* = \delta$ then membrane i is different from the skin membrane $(i \neq 1)$ and different from the output membrane, if any.

When applying such a rule, all objects in v_1 will be placed in the same region i where the rule is applied; all objects in v_2 will be placed in the (parent) region $p(i)$, all objects in v_3 will be placed in the region j, and if $\delta^* = \delta$ then membrane i is dissolved. After dissolving a membrane, all objects and membranes previously present in it become elements of the contents of the immediately upper membrane that have not been dissolved.

A configuration is a *halting configuration* if no rule of the system is applicable to it. We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step*, denoted $\mathcal{C}_1 \Rightarrow_\Pi \mathcal{C}_2$, if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks. A *computation* of Π is a (finite or infinite) sequence of configurations such that: (a) the first term of the sequence is the initial configuration of the system; (b) each non-initial configuration of the sequence is obtained from the previous configuration in one transition step; and (c) if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration.

All computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output region i_{out} in the halting configuration.

2.1.1 P systems with active membranes

One of the explicit goals of various branches of natural computing is to find ways to address computationally hard problems (typically, **NP**-complete problems) in order to solve them in a feasible time.

The rules of a basic transition P system are used in parallel. This is a good degree of parallelism, which, however, is not sufficient to devise polynomial time solutions to **NP**-complete problems (unless $\mathbf{P} = \mathbf{NP}$, which is not at all plausible); the proof of this result can be found in [209]. However, biology suggests operations with membranes such as *membrane division* which, sometimes surprisingly, make possible polynomial (often linear) solutions to **NP**-complete problems. Membrane division brings a further level of parallelism, making possible to construct an exponential workspace expressed in terms of the number of membranes and the number of objects in polynomial time.

P systems with active membranes having associated electrical charges with membranes were first introduced by Gh. Păun [149].

Definition 2.2. *A P* system with active membranes *of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$*, where:*

1. Γ *is a finite alphabet;*
2. μ *is a rooted tree whose nodes are injectively labelled with* $H = \{1, \dots, q\}$*;*
3. $\mathcal{M}_1, \dots, \mathcal{M}_q$ *are finite multisets over* Γ*;*
4. $\mathcal{R}$ *is a finite set of rules, of the following forms:*
   - *(a)* $[\,a \to u\,]_h^{\alpha}$*, for* $h \in H$*,* $\alpha \in \{+, -, 0\}$*,* $a \in \Gamma$*,* $u \in \Gamma^*$ (object evolution rules).
   - *(b)* $a\,[\;\;]_h^{\alpha_1} \to [\,b\,]_h^{\alpha_2}$*, for* $h \in H$*,* $\alpha_1, \alpha_2 \in \{+, -, 0\}$*,* $a, b \in \Gamma$ (send–in communication rules).
   - *(c)* $[\,a\,]_h^{\alpha_1} \to [\;\;]_h^{\alpha_2}\, b$*, for* $h \in H$*,* $\alpha_1, \alpha_2 \in \{+, -, 0\}$*,* $a, b \in \Gamma$ (send–out communication rules).
   - *(d)* $[\,a\,]_h^{\alpha} \to b$*, for* $h \in H$*,* $\alpha \in \{+, -, 0\}$*,* $a, b \in \Gamma$ (dissolution rules).
   - *(e)* $[\,a\,]_h^{\alpha_1} \to [\,b\,]_h^{\alpha_2}\;[\,c\,]_h^{\alpha_3}$*, for* $h \in H$*,* $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$*,* $a, b, c \in \Gamma$ (division rules for elementary membranes).
   - *(f)* $[\,[\,]_{h_1}^{\alpha_1} \cdots [\,]_{h_k}^{\alpha_1}\;[\,]_{h_{k+1}}^{\alpha_2} \cdots [\,]_{h_n}^{\alpha_2}\,]_h^{\alpha} \to [\,[\,]_{h_1}^{\alpha_3} \cdots [\,]_{h_k}^{\alpha_3}\,]_h^{\beta}\,[\,[\,]_{h_{k+1}}^{\alpha_4} \cdots [\,]_{h_n}^{\alpha_4}\,]_h^{\gamma}$*, for* $k \geq 1$*,* $n > k$*,* $h, h_1, \dots, h_n \in H$*,* $\alpha, \beta, \gamma, \alpha_1, \dots, \alpha_4 \in \{+, -, 0\}$ *and* $\{\alpha_1, \alpha_2\} = \{+, -\}$ (division rules for non–elementary membranes).

5. $i_{out} \in \{0, 1, \ldots, q\}$.

A *P system with active membranes* $\Pi = (\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, of degree $q \geq 1$ can be viewed as as a set of q membranes (the nodes ot the tree μ) injectively labelled by $1, \ldots, q$, with electrical charges $(+, -, 0)$ associated with them, and with an environment labelled by 0 such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ representing the objects initially placed in the q membranes of the system; (b) $\mathcal{R}$ is the set of rules that allows to evolve the system; and (c) $i_{out} \in \{0, 1, 2, \ldots, q\}$ represents a distinguished *zone* which will encode the *output* of the system.

P systems with active membranes differ from the basic transition P systems on the type of rules. These rules are applied according to the following principles ([151]):

- All the rules are applied in parallel and in a maximal manner. In one step, one object of a membrane can be used by only one rule (chosen in a non deterministic way), but any object which can evolve by one rule of any form, must evolve.
- If a membrane is dissolved, its content (multiset and internal membranes) is left free in the surrounding region.
- If at the same time a membrane labelled by h is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then we suppose that first the evolution rules of type (a) are used, and then the division is produced. Of course, this process takes only one step.
- The rules associated with membranes labelled by h are used for all copies of this membrane. At one step, a membrane can be the subject of *only one* rule of types (b)-(e).

Note that these P systems have some important features: (a) they use three electrical charges; (b) the polarization of a membrane, but not the label, can be modified by the application of a rule; and (c) they do not use cooperation (the left-hand side of the rules consist of only one symbol).

2.1.2 Polarizationless P systems with active membranes

Next, P systems with active membranes without electrical charges and with different kinds of membrane division rules are introduced.

Definition 2.3. *A* polarizationless P system with active membranes *of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, R, i_{out})$*, where:*

1. Γ *is a finite alphabet;*
2. μ *is a membrane structure (a rooted tree) consisting of* q *membranes injectively labeled by elements of* $H = \{1, \ldots, q\}$*;*
3. $\mathcal{M}_1, \ldots, \mathcal{M}_q$ *are finite multisets over* Γ*; describing the multisets of objects placed in the* q *initial regions of* μ*;*
4. R *is a finite set of developmental rules, of the following forms:*
   (a) $[\, a \to u \,]_h$*, for* $h \in H$*,* $a \in \Gamma$*,* $u \in \Gamma^*$ (object evolution rules)*.*
   (b) $a\,[\ \ ]_h \to [\, b \,]_h$*, for* $h \in H$*,* $a, b \in \Gamma$ (send–in communication rules)*.*
   (c) $[\, a \,]_h \to [\ ]_h\, b$*, for* $h \in H$*,* $a, b \in \Gamma$ (send–out communication rules)*.*
   (d) $[\, a \,]_h \to b$*, for* $h \in H$*,* $a, b \in \Gamma$ (dissolution rules)*.*
   (e) $[\, a \,]_h \to [\, b \,]_h\ [\, c \,]_h$*, for* $h \in H$*,* $a, b, c \in \Gamma$ (division rules for elementary *or weak* division rules for non-elementary membranes)*.*
   (f) $[\,[\,]_{h_1} \cdots [\,]_{h_k}\, [\,]_{h_{k+1}} \cdots [\,]_{h_n}\,]_h \to [\,[\,]_{h_1} \cdots [\,]_{h_k}\,]_h\, [\,[\,]_{h_{k+1}} \cdots [\,]_{h_n}\,]_h$*, where* $k \geq 1$*,* $n > k$*,* $h, h_1, \ldots, h_n \in H$ (*strong* division rules for non-elementary membranes)*.*
5. $i_{out} \in \{0, 1, \ldots, q\}$*.*

These rules are applied according to usual principles of polarizationless P systems (see [58] for details). Notice that in this polarizationless framework there is no cooperation, priority, nor changes of the labels of membranes. Besides, throughout this work, rules of type (f) are used only for $k = 1, n = 2$, that is, rules of the form (f) $[\,[\ ]_{h_1}[\ ]_{h_2}\,]_h \to [\,[\ ]_{h_1}\,]_h\ [\,[\ ]_{h_2}\,]_h$.

2.2 Tissue P Systems with symport/antiport rules

In this section we consider computational devices inspired from the cell inter–communication in tissues, and adding the ingredient of cell division rules of the same form as in cell–like membrane systems with active membranes, but without using polarizations. In these systems, the rules are used in the non-deterministic maximally parallel way, as usual, but we suppose that when a cell is divided, its interaction with other cells or with the environment is blocked; that is, if a division rule is used for dividing a cell, then this cell does not participate in any other rule, for division or communication. The set of communication rules implicitely provides the graph associated with the system through the labels of the membranes. The cells obtained by division have the same labels as the mother cell, hence the rules to be used for evolving them or their objects are inherited.

Definition 2.4. *A basic tissue P system with symport/antiport rules of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$*, where:*

1. Γ *is a finite* alphabet*;*
2. $\mathcal{E} \subseteq \Gamma$*;*
3. $\mathcal{M}_1, \dots, \mathcal{M}_q$ *are finite multisets over* Γ*;*
4. $\mathcal{R}$ *is a finite set of communication rules of the form* $(i, u/v, j)$*, for* $i, j \in \{0, 1, 2, \dots, q\}, i \neq j$*,* $u, v \in \Gamma^*$*,* $|u| + |v| > 0$*;*
5. $i_{out} \in \{0, 1, 2, \dots, q\}$.

A *basic tissue P system with symport/antiport rules* of degree $q \geq 1$

$$\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$$

can be viewed as a set of q cells, labelled by $1, \dots, q$, with an environment labelled by 0 such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets of objects (elements in Γ) initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects located initially in the environment of the system, all of them appearing in an *arbitrary number of copies*; and (c) $i_{out} \in \{0, 1, 2, \dots, q\}$ represents a distinguished *zone* called the *output zone*. We use the term *zone* i $(0 \leq i \leq q)$ to refer to cell i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$.

a distinguished cell or the environment which will encode the output of the system.

A communication rule $(i, u/v, j)$ is called a *symport rule* if $u = \lambda$ or $v = \lambda$. A symport rule $(i, u/\lambda, j)$, with $i \neq 0, j \neq 0$, provides a virtual arc from cell i to cell j. A communication rule $(i, u/v, j)$ is called an *antiport rule* if $u \neq \lambda$ and $v \neq \lambda$. An antiport rule $(i, u/v, j)$, with $i \neq 0, j \neq 0$, provides two arcs: one from cell i to cell j and another one from cell j to cell i. Thus, every tissue P systems has an underlying directed graph whose nodes are the cells of the system and the arcs are obtained from communication rules. In this context, the environment can be considered as a virtual node of the graph such that their connections are defined by the communication rules of the form $(i, u/v, j)$, with $i = 0$ or $j = 0$.

A communication rule $(i, u/v, j)$ is applicable to regions i, j if the multiset u is contained in region i and multiset v is contained in region j. When applying a communication rule $(i, u/v, j)$, the objects of the multiset represented by u are sent from region i to region j and, simultaneously, the objects of multiset v are sent from region j to region i. The *length* of communication rule $(i, u/v, j)$ is defined as $|u| + |v|$.

The rules of a system like the one above are used in a non-deterministic maximally parallel manner as it is customary in Membrane Computing. At each step, all cells which can evolve must evolve in a maximally parallel way (at each step we apply a multiset of rules which is maximal, no further applicable rule can be added).

An *instantaneous description* or a *configuration* at any instant of a tissue P system is described by all multisets of objects over Γ associated with all the cells present in the system, and the multiset of objects over $\Gamma - \mathcal{E}$ associated with the environment at that moment. Bearing in mind that the objects from $\mathcal{E}$ have infinite copies in the environment, they are not properly changed along the computation. The *initial configuration* is $(\mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$. A configuration is a *halting configuration* if no rule of the system is applicable to it.

Let us fix a tissue P system with symport/antiport rules Π. We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step*, denoted $\mathcal{C}_1 \Rightarrow_\Pi \mathcal{C}_2$, if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks. A *computation* of Π is a (finite or infinite) sequence of configurations such that: (a) the first term of the sequence is an initial configuration of the system; (b) each non-initial configuration of the sequence is obtained from the previous configuration by applying the rules of the system in a maximally parallel manner with the restrictions previously mentioned; and (c) if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration.

All computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output region (a cell or the environment) i_{out} in the halting configuration.

Notation: If $\mathcal{C} = \{\mathcal{C}_i\}_{i<r+1}$ $(r \in \mathbf{N})$ is a halting computation of Π, then the length of $\mathcal{C}$ is r, that is, the number of non-initial configurations which appear in the finite sequence $\mathcal{C}$. We denote it by $|\mathcal{C}|$. We also denote by $\mathcal{C}_i(j)$ the

contents of cell j at configuration $\mathcal{C}_i$.

Definition 2.5. *A tissue P system* without environment *is a tissue P system* $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ *such that* $\mathcal{E} = \emptyset$.

In this kind of tissue P systems, along any computation, objects in the environment always have a finite multiplicity.

2.2.1 Tissue P Systems with Cell Division

Cell division is an elegant process that enables organisms to grow and reproduce. Mitosis is a process of cell division which results in the production of two daughter cells from a single parent cell. Daughter cells are identical to one another and to the original parent cell. Through a sequence of steps, the replicated genetic material in a parent cell is equally distributed to two daughter cells. While there are some subtle differences, mitosis is remarkably similar across organisms.

Before a dividing cell enters mitosis, it undergoes a period of growth where the cell replicates its genetic material and organelles. Replication is one of the most important functions of a cell. DNA replication is a simple and precise process that creates two complete strands of DNA (one for each daughter cell) where only one existed before (from the parent cell).

Let us recall that the model of *tissue P systems with cell division* is based on the membrane division rules from the cell-like model of *P systems with active membranes* [150]. Cells obtained by division get exactly the same labels as the original cell, but no polarizations are used in this model (unlike the active membranes case).

Definition 2.6. *A tissue P system with cell division of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, *where:*

- $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ *is a basic tissue P system with symport/antiport rules of degree* $q \geq 1$*;*
- $\mathcal{R}$ *may also contain a finite set of rules of the form* $[a]_i \to [b]_i[c]_i$, *being* $i \in \{1, 2, \ldots, q\}$, $i \neq i_{out}$ *and* $a, b, c \in \Gamma$ (Division rules).

A division rule $[a]_i \to [b]_i[c]_i$ is applicable to a configuration at an instant t, if there is a cell i in that configuration and object a is contained in that cell. When applying a division rule $[a]_i \to [b]_i[c]_i$, under the influence of object a, the cell with label i is divided into two cells with the same label; in the first copy, object a is replaced by object b, in the second one, object a is replaced by object c; all the other objects residing in cell i are replicated and copies of them are placed in the two new cells. The output cell i_{out} cannot be divided.

The rules of a tissue P system with cell division are applied in a non-deterministic maximally parallel manner as it is customary in membrane computing but with following important remark: if a cell divides, then the division rule is the only one which is applied for that cell at that step; the objects inside that cell do not evolve by means of communication rules. In other words, before division a cell interrupts all its communication channels with the other cells and with the environment. The new cells resulting from division will interact with other cells or with the environment only at the next step – providing that they do not divide once again. The label of a cell precisely identifies the rules which can be applied to it.

2.2.2 Tissue P Systems with Cell Separation

In formal models of membrane systems with cell separation, the cells are not polarized; the two cells obtained by separation inherit the same labels as the original cell, and if a cell is separated, its interaction with other cells or with the environment is blocked during the separation process. These assumptions, together with the original abstract concept of a P system [148], and previous models studied in [92, 92] and [118], motivated the following definition:

Definition 2.7. *A tissue P system with cell separation and communication rules of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where:*

1. $\Pi = (\Gamma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$ *is a basic tissue P system with symport/antiport rules of degree $q \geq 1$;*

2. $\{\Gamma_0, \Gamma_1\}$ *is a partition of* Γ, *that is,* $\Gamma = \Gamma_0 \cup \Gamma_1$, $\Gamma_0, \Gamma_1 \neq \emptyset$, $\Gamma_0 \cap \Gamma_1 = \emptyset$.

3. $\mathcal{R}$ *may also contain a finite set of rules of the form* $[a]_i \to [\Gamma_0]_i[\Gamma_1]_i$, *where* $i \in \{1, \dots, q\}$, $a \in \Gamma$ *and* $i \neq i_{out}$ (Separation rules).

A separation rule $[a]_i \to [\Gamma_0]_i[\Gamma_1]_i$ is applicable to a configuration at an instant t, if cell i belongs to that configuration and object a is contained in that cell. When applying a separation rule $[a]_i \to [\Gamma_0]_i[\Gamma_1]_i$, in reaction with an object a, the cell i is separated into two cells with the same label; at the same time, object a is consumed; the objects from Γ_0 are placed in the first cell, those from Γ_1 are placed in the second cell; the output cell i_{out} cannot be separated.

The rules of a tissue P system with cell separation are applied in a non-deterministic maximally parallel manner as it is customary in membrane computing but with following important remark: if a cell separates, then the separation rule is the only one which is applied for that cell at that step.

2.3 Recognizer membrane systems

Throughout this work we use the term *membrane system* to refer to both a cell-like P system or a tissue-like P system. In both cases we can describe them by $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$, where in the case of cell-like P system, the structure μ is explicitely given, the set $\mathcal{R}$ of rules is expressed by $\mathcal{R} = \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_q$ (the set $\mathcal{R}_i$ is associated with membrane i), and the alphabet of the environment $\mathcal{E}$ is considered as the empty set. In the case of tissue-like P system, the structure μ is implicitely given by the set of rules $\mathcal{R}$ (associated with the whole system).

Throughout this chapter, it is assumed that each decision problem has an associated fixed *reasonable encoding scheme* that describes the instances of the problem by means of strings over a finite alphabet. We do not define *reasonable* in a formal way, however, following [49], instances should be encoded in a concise manner, without irrelevant information, and where relevant numbers are represented in binary (or any fixed base other than 1). It is possible to use multiple reasonable encoding schemes to represent instances, but it is proved that the input sizes differ at most by a polynomial. The *size* $|u|$ of an instance u is the length of the string associated with it, in some reasonable encoding scheme.

In order to study the computational efficiency of membrane systems, the notions from classical *computational complexity theory* are adapted for Membrane Computing, and a special class of cell-like P systems is introduced in [70]: *recognizer P systems* (called *accepting P systems* in a previous paper [71]).

Definition 2.8. *A* recognizer membrane system *of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$, where:*

1. $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{out})$ *is a membrane system;*

2. *The working alphabet Γ has two distinguished objects* `yes` *and* `no` *being, at least, one copy of them present in $\mathcal{M}_1 \cup \cdots \cup \mathcal{M}_q$, but, in the case of a tissue-like P system, none of them are present in $\mathcal{E}$;*

3. *Σ is a finite (input) alphabet strictly contained in Γ such that $\mathcal{E} \cap \Sigma = \emptyset$;*

4. *$\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over $\Gamma \setminus \Sigma$;*

5. *$i_{in} \in \{1, \dots, q\}$ is the input region (a membrane in the case of cell-like P systems and a cell in the case of tissue-like P systems);*

6. *i_{out} is the label of the environment that represents the output zone;*

7. *All computations halt;*

8. *If $\mathcal{C}$ is a computation of Π, then either object* `yes` *or object* `no` *(but not both) must have been released into the environment, and only at the last step of the computation.*

A *recognizer membrane system* $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$ of degree $q \geq 1$ can be viewed as a membrane system such that has an input alphabet Σ and an input region i_{in}. The initial multisets of the system are multisets over $\Gamma \setminus \Sigma$.

For each finite multiset m over Σ, the *initial configuration of the system* Π *with input* m is the tuple $(\mu, \mathcal{M}_1, \dots, \mathcal{M}_{i_{in}} + m, \dots, \mathcal{M}_q; \emptyset)$, where the input multiset m is added to the content of the input region i_{in}. That is, we have an initial configuration associated with each input multiset m over Σ in recognizer P systems. We denote by $\Pi + m$ the P system Π with input multiset m.

Given a recognizer membrane system Π, and a halting computation $\mathcal{C} = \{\mathcal{C}_i\}_{i<r+1}$ of Π ($r \in \mathbf{N}$), the result of $\mathcal{C}$ is `yes` (respectively, `no`) if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$, and neither object `yes` nor `no` appears in the environment associated with any non–halting configuration of $\mathcal{C}$. If the result of a computation $\mathcal{C}$ is `yes` (respectively, object `no`), then we say that $\mathcal{C}$ is an *accepting computation* (respectively, *rejecting computation*).

2.3.1 Polynomial Complexity Classes of membrane systems

Many formal machine models (e.g. Turing machines or register machines) have an infinite number of memory locations. At the same time, P systems, or logic circuits, are computing devices of finite size and they have a finite description with a fixed amount of initial resources (number of membranes, objects, gates, etc.). For this reason, in order to solve a decision problem a (possibly infinite) family of P systems is considered.

The concept of solvability in the framework of P systems also takes into account the pre-computational process of (efficiently) constructing the family that provides the solution. In this paper, the terminology *uniform family* is used to denote that this construction is performed by a *single* computational machine.

2.3.1.1 Uniform families of P systems

In the case of recognizer P systems (with input region: membrane/cell), the term uniform family is consistent with the usual meaning for Boolean circuits: a family $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ is uniform if there exists a deterministic Turing machine which constructs the system $\Pi(n)$ from $n \in \mathbb{N}$ (that is, which on input 1^n outputs $\Pi(n)$). In such a family, the P system $\Pi(n)$ will process all the instances of the problem with numerical parameters (reasonably) encoded by n – the common case is that $\Pi(n)$ processes all instances of size n. Note that this means that, for these families of P systems with input region, further pre–computational processes are needed in order to (efficiently) determine which P system (and from which input) deals with a given instance of the problem. The concept of *polynomial encoding* introduced below tries to capture this idea.

In the case of P systems without input region a new notion arises: a family $\mathbf{\Pi} = \{\Pi(w) : w \in I_X\}$ *associated with a decision problem* $X = (I_X, \theta_X)$ is *uniform* (some authors [105, 143, 166] use the term *semi-uniform* here) if there exists a deterministic Turing machine which constructs the system $\Pi(w)$ from the instance $w \in I_X$. In such a family, each P system usually processes only one instance, and the numerical parameters and syntactic specifications of the latter are part of the definition of the former.

It is important to point out that, in both cases, the family should be constructed in an efficient way. This requisite was first included within the term uniform family (introduced by Gh. Păun [151]), but nowadays it is preferred to use the term *polynomially uniform by Turing machines* to indicate a uniform (by a single Turing machine) and effective (in polynomial time) construction of the family.

Definition 2.9. *A family* $\mathbf{\Pi} = \{\Pi(w) : w \in I_X\}$ (*respectively,* $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$) *of recognizer membrane systems without input region (resp., with input region) is* polynomially uniform by Turing machines *if there exists a deterministic Turing machine working in polynomial time which constructs the system* $\Pi(w)$ (*resp.,* $\Pi(n)$) *from the instance* $w \in I_X$ (*resp., from* $n \in \mathbb{N}$).

2.3.1.2 Confluent P systems.

In order for recognizer P systems to capture the true algorithmic concept, a condition of *confluence* is imposed, in the sense that all possible successful computations must give the same answer. This contrasts with the standard notion of accepting computations for non-deterministic (classic) models.

Definition 2.10. *Let $X = (I_X, \theta_X)$ be a decision problem, and $\mathbf{\Pi} = \{\Pi(w) : w \in I_X\}$ be a family of recognizer P systems without input region.*

- $\mathbf{\Pi}$ *is said to be* sound with respect to X *if the following holds: for each instance of the problem, $w \in I_X$, if* there exists *an accepting computation of* $\Pi(w)$*, then $\theta_X(w) = 1$.*

- $\mathbf{\Pi}$ *is said to be* complete with respect to X *if the following holds: for each instance of the problem, $w \in I_X$, if $\theta_X(w) = 1$, then* every *computation of* $\Pi(w)$ *is an accepting computation.*

The concepts of soundness and completeness can be extended to families of recognizer P systems with input region in a natural way. However, an efficient process of selecting P systems from instances must be made precise.

Definition 2.11. *Let $X = (I_X, \theta_X)$ be a decision problem, and $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ a family of recognizer P systems with input region. A* polynomial encoding *of X in $\mathbf{\Pi}$ is a pair (cod, s) of polynomial–time computable functions over I_X such that for each instance $w \in I_X$, $s(w)$ is a natural number (obtained by means of a reasonable encoding scheme) and $cod(w)$ is an input multiset of the system $\Pi(s(w))$.*

Polynomial encodings are stable under polynomial–time reductions [70].

Proposition 2.1. *Let X_1, X_2 be decision problems, r a polynomial–time reduction from X_1 to X_2, and (cod, s) a polynomial encoding from X_2 to $\mathbf{\Pi}$. Then, $(cod \circ r, s \circ r)$ is a polynomial encoding from X_1 to $\mathbf{\Pi}$.*

Next, the concepts of soundness and completeness are defined for families of recognizer P systems with input region.

Definition 2.12. *Let $X = (I_X, \theta_X)$ be a decision problem, $\mathbf{\Pi} = \{\Pi(n) : n \in \mathbb{N}\}$ a family of recognizer P systems with input region, and (cod, s) a polynomial encoding of X in $\mathbf{\Pi}$.*

- $\mathbf{\Pi}$ *is said to be* sound with respect to (X, cod, s) *if the following holds: for each instance of the problem, $w \in I_X$, if* there exists *an accepting computation of* $\Pi(s(w))$ *with input $cod(w)$, then $\theta_X(w) = 1$.*

- $\mathbf{\Pi}$ *is said to be* complete with respect to (X, cod, s) *if the following holds: for each instance of the problem, $w \in I_X$, if $\theta_X(w) = 1$, then* every *computation of* $\Pi(s(w))$ *with input cod(w) is an accepting computation.*

Notice that if a family of recognizer P systems is sound and complete, then every P system of the family processing an instance, is confluent in the sense previously mentioned.

2.3.2 Semi-Uniform Solutions versus Uniform Solutions

The first results showing that membrane systems could solve computationally hard problems in polynomial time were obtained using P systems without input membrane. In that context, a specific P system is associated with each instance of the problem. In other words, the syntax of the instance is part of the description of the associated P system. Thus this P system can be considered *special purpose.*

Definition 2.13. *A decision problem X is* solvable in polynomial time *by a family of recognizer P systems without input membrane $\mathbf{\Pi} = \{\Pi(w) : w \in I_X\}$, denoted by $X \in \mathbf{PMC}^*_{\mathcal{R}}$, if the following holds:*

- *The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines.*
- *The family $\mathbf{\Pi}$ is polynomially bounded; that is, there exists a natural number $k \in \mathbb{N}$ such that for each instance $w \in I_X$, every computation of $\Pi(w)$ performs at most $|w|^k$ steps.*
- *The family $\mathbf{\Pi}$ is sound and complete with respect to X.*

The family $\mathbf{\Pi}$ is said to provide a *semi–uniform solution* to the problem X. Let $\mathcal{R}$ be a class of recognizer P systems. We denote by $\mathbf{PMC}^*_{\mathcal{R}}$ the set of all decision problems which can be solved in a uniform way and polynomial time by means of families of systems from $\mathcal{R}$.

Now, we define what it means to solve a decision problem in the framework of membrane systems efficiently and in a *uniform way.* Since we define each membrane system to work on a finite number of inputs, to solve a decision problem we define a numerable family of membrane systems.

Definition 2.14. *We say that a decision problem $X = (I_X, \theta_X)$ is solvable in a uniform way and polynomial time by a family $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ of recognizer membrane systems if the following holds:*

1. *The family* $\mathbf{\Pi}$ *is* polynomially uniform *by Turing machines, that is, there exists a deterministic Turing machine working in polynomial time which constructs the system* $\Pi(n)$ *from* $n \in \mathbb{N}$.

2. *There exists a pair* (cod, s) *of polynomial-time computable functions over* I_X *such that:*

   (a) *for each instance* $u \in I_X$, $s(u)$ *is a natural number and* $cod(u)$ *is an input multiset of the system* $\Pi(s(u))$;

   (b) *for each* $n \in \mathbb{N}$, $s^{-1}(n)$ *is a finite set;*

   (c) *the family* $\mathbf{\Pi}$ *is* polynomially bounded *with regard to* (X, cod, s), *that is, there exists a polynomial function* p, *such that for each* $u \in I_X$ *every computation of* $\Pi(s(u))$ *with input* $cod(u)$ *is halting and it performs at most* $p(|u|)$ *steps;*

   (d) *the family* $\mathbf{\Pi}$ *is* sound *and* complete *with regard to* (X, cod, s).

From the soundness and completeness conditions above we deduce that every P system $\Pi(n)$ is *confluent*, in the following sense: every computation of a system with the *same* input multiset must always give the *same* answer.

Let $\mathcal{R}$ be a class of recognizer P systems. We denote by $\mathbf{PMC}_{\mathcal{R}}$ the set of all decision problems which can be solved in a uniform way and polynomial time by means of families of systems from $\mathcal{R}$. The class $\mathbf{PMC}_{\mathcal{R}}$ is closed under complement and polynomial–time reductions [71].

2.4 Spiking Neural P systems

Spiking Neural P systems (SN P systems, for short) are a variant of P systems, corresponding to a shift from *cell-like* to *neural-like* architectures. Formally introduced in [67], they take inspiration from the way in which neurons in the brain exchange information by means of the propagation through their synapses of electrical impulses called action potentials or spikes. Due to their entity with respect to the work object of this dissertation, they are covered separately in the Chapter 4.

3 The P versus NP problem from a Membrane Computing perspective

Classical approach to tackle the **P** versus **NP** problem consists of considering an NP-complete problem (it is enough to work with *only one* such problem) and using the following strategy: either (a) try to design a deterministic algorithm solving that problem in polynomial time, concluding then that $\mathbf{P} = \mathbf{NP}$; or else (b) prove that no deterministic algorithm solves that problem in polynomial time, concluding then that $\mathbf{P} \neq \mathbf{NP}$.

This chapter deals with a new approach to tackle the **P** versus **NP** problem by using a bio-inspired computing paradigm that provides borderlines of the tractability of abstract problems in terms of syntactical ingredients of these computing devices. Specifically, the computational complexity theory developed in the framework of membrane computing allow us to characterize the ability to solve **NP**–complete problem in an efficient way.

3.1 Introduction

The **P** versus **NP** question is the problem of determining whether every problem solvable by some non-deterministic Turing machine in polynomial time can also be solved by some deterministic Turing machine in polynomial time. In other words, this question can be informally reformulated as asking whether or not finding solutions of a (search) problem is harder than checking the correctness of a given solution of it. It can also be expressed as asking whether or not deciding membership in a set is harder than being convinced of this membership by an adequate proof ([53]). The classical approach when trying to solve this question is to focus on one **NP**-complete problem and try to prove that it belongs (or that it does not belong) to the class **P**. A membership proof can be obtained by designing a deterministic algorithm solving that problem in polynomial time, and in such case we immediately deduce that $\mathbf{P} = \mathbf{NP}$ holds. In the second case, claiming that the problem does not belong to **P** is equivalent to proving that no deterministic algorithm working in polynomial time provides a solution to the problem, and therefore in this case we deduce $\mathbf{P} \neq \mathbf{NP}$.

The **P** versus **NP** question is one of the outstanding open problems in theoretical computer science. A negative answer to this question would confirm that the majority of current cryptographic systems are secure from a practical point of view. A positive answer would not only show the uncertainty about the security of these systems, but also this kind of answer is expected to come together with a general procedure that provides a deterministic algorithm solving most of the **NP**–complete problems in polynomial time.

This chapter studies a new computing approach to attack the **P** versus **NP** question by obtaining frontiers of the tractability in terms of syntactical ingredients of cell-like P systems and tissue-like P systems. Moreover, each borderline provides a tool to solve the aforementioned question.

3.2 Frontiers of the Efficiency in Membrane Systems

In this section the concept of efficient membrane systems is introduced and we study the efficiency of different classes of recognizer membrane systems with respect to some syntactical ingredients.

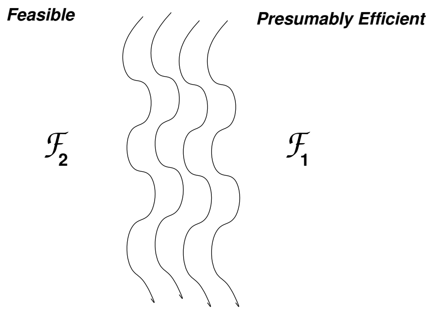
We say that a class of recognizer membrane systems $\mathcal{F}$ is *presumably efficient* if there exists an **NP**–complete problem that can be solved in polynomial time by a family of systems from $\mathcal{F}$. From the properties of the **NP**–completeness, we deduce that *any* **NP**-complete problem can be solved in polynomial time by families of an presumably efficient class of recognizer membrane systems. Because class $\mathbf{PMC}_{\mathcal{F}}$ is closed under complement and polynomial–time reductions (see [71] for details), if the class $\mathcal{F}$ is presumably efficient then $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC}_{\mathcal{F}}$.

We say that a class of recognizer membrane systems $\mathcal{F}$ is *feasible* if only tractable problems can be solved in polynomial time by a family of systems from $\mathcal{F}$, that is, if $\mathbf{PMC}_{\mathcal{F}} = \mathbf{P}$. According with these definitions, if $\mathbf{P} = \mathbf{NP}$ then a class $\mathcal{F}$ is feasible if and only if it is presumably efficient. Besides, if $\mathbf{P} \neq \mathbf{NP}$ then each class feasible is not presumably efficient. Nevertheless, under that hypothesis a class no feasible could be no presumably efficient (as a consequence of the Ladner theorem by which if $\mathbf{P} \neq \mathbf{NP}$ then there exist **NP**-intermediate problems, that is problems which are neither in the class **P** nor in the class of **NP**-complete problems, see [78] for details).

A good strategy to proof that **P**=**NP** is the following: Let us suppose we have two classes $\mathcal{F}_1$ and $\mathcal{F}_2$ of recognizer membrane systems such that:

(a) $\mathcal{F}_1$ is feasible and $\mathcal{F}_2$ is presumably efficient.

(b) Each solution S of a decision problem X in $\mathcal{F}_1$ is also a solution in $\mathcal{F}_2$;



The syntactical ingredients required to add to membrane systems in $\mathcal{F}_1$ to obtain membrane systems in $\mathcal{F}_2$ provide a borderline between tractability and **NP**–hardness. Therefore, *translating* an efficient solution of an **NP**–complete problem by a family of systems in $\mathcal{F}_2$, into an efficient solution by a family of systems in $\mathcal{F}_1$ amounts to proving **P**=**NP**.

Let us recall that there are three important techniques (dependency graph technique, simulation technique and algorithmic technique) that are used when the feasibility of a class of membrane systems is established.

3.2.1 Dependency graph technique

Let Π be a recognizer membrane system where all its communication rules have length 1. In this case, each rule of Π can be activated by a single object (note that this holds also for division or separation rules). Hence, there exists in some sense, a *dependency* between the object triggering the rule and the object or objects produced by its application. Then, a directed graph (*dependency graph*) can be associated with Π verifying the following relevant property: there exists an accepting computation of Π if and only if there exists a path between two distinguished nodes in the dependency graph associated with it (see [60] and [58] for more details).

3.2.2 Simulation technique

Let us define the meaning of efficient simulations in the framework of recognizer membrane systems. Given two recognizer membrane systems, Π and Π', we say that Π' *simulates* Π *in an efficient way* if the following holds: (a) Π' can be constructed from Π by a deterministic Turing machine working in polynomial time; and (b) There exists an injective function, f, from the set $\mathbf{Comp}(\Pi)$ of computations of Π onto the set $\mathbf{Comp}(\Pi')$ of computations of Π' such that:

- ⋆ There exists a deterministic Turing machine that constructs computation $f(\mathcal{C})$ from computation $\mathcal{C}$ in polynomial time.
- ⋆ A computation $\mathcal{C} \in \mathbf{Comp}(\Pi)$ is an accepting computation if and only if $f(\mathcal{C}) \in \mathbf{Comp}(\Pi')$ is an accepting one.
- ⋆ There exists a polynomial function $p(n)$ such that for each $\mathcal{C} \in \mathbf{Comp}(\Pi)$ we have $|f(\mathcal{C})| \leq p(|\mathcal{C}|)$.

3.2.3 Algorithmic technique

The technique consists of the construction of a deterministic algorithm $\mathcal{A}$ working in polynomial time that receives as input a membrane system Π from $\mathcal{F}$ and an input multiset m of Π. Then, algorithm $\mathcal{A}$ reproduces the behaviour of a computation of $\Pi + m$. In particular, if the given membrane system is confluent then the algorithm will provide the same answer of the system, that is, the answer of algorithm $\mathcal{A}$ is affirmative if and only if the system $\Pi + m$ has an accepting computation (and then, any computation is an accepting one).

3.3 On Efficiency of Cell-like P Systems

In this section we study the computational efficiency (ability to provide polynomial time solution for **NP**–complete problems) of different models of cell-like membrane system.

3.3.1 Feasibility of Transition P Systems

First, the computational efficiency of basic transition P system is established. Let us recall that in this kind of cell-like membrane systems, only evolution, communication, and dissolution rules are allowed. Thus, the membrane structure does not increase. Let us denote $\mathcal{T}$ the class of recognizer basic transition P systems.

An efficient simulation of deterministic Turing machines by recognizer basic transition P systems was given in [59] and the following holds.

Proposition 3.1. `(Sevilla theorem)` *Every deterministic Turing machine working in polynomial time can be simulated in polynomial time by a family of recognizer basic transition P systems with input membrane.*

They also proved that each confluent basic transition P system can be (efficiently) simulated by a deterministic Turing machine [59]. As a consequence, these P systems efficiently solve at most tractable problems.

Proposition 3.2. *If a decision problem is solvable in polynomial time by a family of recognizer basic transition P systems with input membrane, then there exists a deterministic Turing machine solving it in polynomial time.*

These results are also verified for recognizer basic transition P systems without input membrane. Therefore, the following holds.

Theorem 3.1. $\mathbf{P} = \mathbf{PMC}_{\mathcal{T}} = \mathbf{PMC}^*_{\mathcal{T}}$.

Thus, the ability of a P system in $\mathcal{T}$ to create exponential workspace (in terms of number of objects) in polynomial time (e.g. via evolution rules of the type $[\, a \rightarrow a^2 \,]_h$) is not enough to efficiently solve **NP**–complete problems (unless $\mathbf{P} = \mathbf{NP}$).

Corollary 3.1. $\mathbf{P} \neq \mathbf{NP}$ *if and only if every, or at least one,* **NP**–*complete problem is not in* $\mathbf{PMC}_{\mathcal{T}} = \mathbf{PMC}^*_{\mathcal{T}}$.

In the framework of P systems without input membrane, C. Zandron, C. Ferretti and G. Mauri [209] proved that confluent recognizer P systems with active membranes making use of no membrane division rule, can be efficiently simulated by a deterministic Turing machine.

Proposition 3.3. `(Milano theorem)`
A deterministic P system with active membranes but without membrane division can be simulated by a deterministic Turing machine with a polynomial slowdown.

Let $\mathcal{NAM}$ be the class of recognizer P systems with active membranes which do not make use of division rules. As a consequence of the previous result, the following holds:

Corollary 3.2. $\mathbf{PMC}_{\mathcal{NAM}} \subseteq \mathbf{P}$.

A.E. Porreca [144] provides a simple proof of each tractable problem being able to be solved (in a semi–uniform way) by a family of recognizer P systems with active membranes (without polarizations) operating in exactly one step and using only send–out communication rules. That proof can be easily adapted to uniform solutions.

Proposition 3.4. $\mathbf{P} \subseteq \mathbf{PMC}_{\mathcal{NAM}}$.

Thus, we have a version of Theorem 3.1 for the class $\mathcal{NAM}$.

Theorem 3.2. $\mathbf{P} = \mathbf{PMC}_{\mathcal{NAM}}$.

3.3.2 Presumable Efficiency of Cell-like Membrane Systems

The first efficient solutions to **NP**–complete problems by using P systems with active membranes were given in a *semi–uniform* way (where the P systems of the family depend on the syntactic structure of the instance) by S.N. Krishna and R. Rama (`Hamiltonian Path`, `Vertex Cover` [77]), A. Obtulowicz (`SAT` [114]), A. Păun (`Hamiltonian Path` [125]), Gh. Păun (`SAT` [149, 126]), and C. Zandron, C. Ferretti and G. Mauri (`SAT`, `Undirected Hamiltonian Path` [209]).

Let $\mathcal{AM}(+n)$ (respectively, $\mathcal{AM}(-n)$) be the class of recognizer P systems with active membranes using division rules for elementary and non–elementary membranes (respectively, only for elementary membranes).

In the framework of $\mathcal{AM}(-n)$, efficient *uniform* solutions to weakly **NP**–complete problems (`Knapsack` [136], `Subset Sum` [137], `Partition` [57]), and strongly **NP**–complete problems (`SAT` [71], `Clique` [4], `Bin Packing` [140], `Common Algorithmic Problem` [139]) have been obtained.

Since $\mathbf{PMC}_{\mathcal{R}}$ is closed under complement and polynomial time reductions, for any class $\mathcal{R}$ of recognizer P systems, the following result is obtained.

Proposition 3.5. $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC}_{\mathcal{AM}(-n)}$.

In the framework of $\mathcal{AM}(+n)$, P. Sosík [166] gave a efficient *semi–uniform* solution to `QBF-SAT` (satisfiability of quantified propositional formulas), a well known **PSPACE**–complete problem [49]. Hence the following is deduced.

Proposition 3.6. $\mathbf{PSPACE} \subseteq \mathbf{PMC}^*_{\mathcal{AM}(+n)}$.

This result has been extended by A. Alhazov, C. Martín–Vide and L. Pan [3] showing that `QBF-SAT` can be solved in a linear time and in a *uniform* way by a family of recognizer P systems with active membranes (without using dissolution rules) and using division rules for elementary and non–elementary membranes.

Proposition 3.7. $\mathbf{PSPACE} \subseteq \mathbf{PMC}_{\mathcal{AM}(+n)}$.

A.E. Porreca, G. Mauri and C. Zandron [143] described a (deterministic and efficient) algorithm simulating a single computation of any confluent recognizer P system with active membranes and without input. Thus,

Proposition 3.8. $\mathbf{PMC}^*_{\mathcal{AM}(+n)} \subseteq \mathbf{EXP}$.

Therefore, $\mathbf{PMC}_{\mathcal{AM}(+n)}$ and $\mathbf{PMC}^*_{\mathcal{AM}(+n)}$ are two membrane computing complexity classes between **PSPACE** and **EXP**.

Corollary 3.3. $\mathbf{PSPACE} \subseteq \mathbf{PMC}_{\mathcal{AM}(+n)} \subseteq \mathbf{PMC}^*_{\mathcal{AM}(+n)} \subseteq \mathbf{EXP}$.

P. Sosík and A. Rodríguez–Patón [167] have proven that the reverse inclusion of Proposition 3.6 holds as well. Nevertheless, the concept of *uniform family* of P systems considered in that paper is different from that of Definition 2.9, although maybe the proof can be adapted to fit into the framework presented in this chapter. In this case the following would hold: $\mathbf{PSPACE} = \mathbf{PMC}^*_{\mathcal{AM}(+n)}$. Thus, the class of recognizer P systems with active membranes, with electrical charges, using division for elementary and non–elementary membranes would be computationally equivalent to standard parallel machine models as PRAMs or alternating Turing machines.

Previous results show that the usual framework of P systems with active membranes for solving decision problems is too powerful from the computational complexity point of view. Therefore, it would be interesting to investigate weaker models of P systems with active membranes able to characterize classical complexity classes below **NP** and providing borderlines between efficiency and non–efficiency.

At the beginning of 2005, Gh. Păun (problem **F** from [127]) wrote:

> *My favorite question (related to complexity aspects in P systems with active membranes and with electrical charges) is that about the number of polarizations. Can the polarizations be completely avoided? The feeling is that this is not possible – and such a result would be rather sound: passing from no polarization to two polarizations amounts to passing from non–efficiency to efficiency.*

This so–called Păun's conjecture can be formally formulated in terms of membrane computing complexity classes as follows: $\mathbf{P} = \mathbf{PMC}^{[*]}_{\mathcal{AM}^0(+d,-ne)}$, where the notation $\mathbf{PMC}^{[*]}_{\mathcal{R}}$ indicates that the result holds for both $\mathbf{PMC}_{\mathcal{R}}$ and $\mathbf{PMC}^{*}_{\mathcal{R}}$ and $\mathcal{AM}^0\,(+d,-ne)$ is the class of polarizationless P systems with active membranes using dissolution, rules and only division for elementary membranes.

Theorem 3.3. $\mathbf{P} = \mathbf{PMC}^{[*]}_{\mathcal{AM}^0\,(-d,\beta)}$, *where* $\beta \in \{-ne, +ne\}$.

Thus, polarizationless P systems with active membranes which do not make use of dissolution rules are non–efficient in the sense that their cannot solve **NP**–complete problems in polynomial time (unless **P**=**NP**).

Let us now consider polarizationless P systems with active membranes making use of dissolution rules. Will it be possible to solve **NP**–complete problems in that framework?

N. Murphy and D. Woods [105] gave a negative answer in the case that division rules are used only for elementary membranes and being *symmetric*, in the following sense $[\,a\,]_h \rightarrow [\,b\,]_h[\,b\,]_h$.

Theorem 3.4. $\mathbf{P} = \mathbf{PMC}^{[*]}_{\mathcal{AM}^0\,(+d,-n(sym))}$.

Several authors [5, 58] gave a positive answer when division for non–elementary membranes, in the strong sense, is permitted. The mentioned papers provide semi–uniform solutions in a linear time to `SAT` and `Subset Sum`, respectively. Thus, we have the following result:

Proposition 3.9. $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC}^{*}_{\mathcal{AM}^0\,(+d,+ns)}$.

As a consequence of Theorem 3.3 and Proposition 3.9, a *partial negative* answer to Păun's conjecture is given: assuming that $\mathbf{P} \neq \mathbf{NP}$ and making use of dissolution rules and division rules for elementary and non–elementary membranes, computationally hard problems can be efficiently solved avoiding polarizations. The answer is partial because efficient solvability of **NP**–complete problems by polarizationless P systems with active membranes making use of dissolution rules and division *only* for elementary membranes is unknown.

The result of Proposition 3.9 was improved by A. Alhazov and M.J. Pérez–Jiménez [6] giving a family of recognizer polarizationless P systems with active membranes using dissolution rules and division for elementary and (strong) non–elementary membranes solving `QBF-SAT` in a *uniform* way and in a linear time. Then,

Proposition 3.10. $\mathbf{PSPACE} \subseteq \mathbf{PMC}_{\mathcal{AM}^0\,(+d,+ns)}$.

3.3.3 Frontiers of the Presumable Efficiency in Cell-like P Systems

Next, based on the results from previous sections, we present different frontiers of the efficiency in terms of syntactical ingredients of recognizer tissue P systems. We can classify them into four categories: the length of communication rules, the direction of such communication rules, or the use of some kind of rules or the environment.

1. The class $\mathcal{NAM}$ is feasible and the class $\mathcal{AM}$ is presumably efficient. Thus, in the framework of polarizationless P systems with active membranes, passing from forbid division rules to allow them amounts to passing from feasibility to presumably efficiency.

2. The class $\mathcal{AM}^0(-d, +ne)$ is feasible and the class $\mathcal{AM}^0(+d, +ne)$ is efficient. Thus, in the framework of polarizationless P systems with active membranes, passing from forbid dissolution rules to allow them amounts to passing from feasibility to presumably efficiency.

Feasible	Presumably Efficient	
$\mathcal{NAM}$	$\mathcal{AM}$	(*adding rules*)
$\mathcal{AM}^0(-d, +ne)$	$\mathcal{AM}^0(+d, +ne)$	(*adding rules*)

3.4 On Efficiency of Tissue-like P Systems

In this section we study the computational efficiency (ability to provide polynomial time solution for **NP**–complete problems) of different models of tissue-like membrane system.

Let us denote **TC** the class of all recognizer basic tissue P systems with symport/antiport rules. For each $k \geq 1$, we denote by $\mathbf{TDC}(k)$ (respectively, $\mathbf{TSC}(k)$) the class of all recognizer tissue P systems with cell division (respectively, with cell separation) which use symport/antiport rules with length at most k. In a similar way, we denote by $\mathbf{TDA}(k)$ (respectively, $\mathbf{TDS}(k)$) the class of all recognizer tissue P systems with cell division which use as communication rules only antiport rules (respectively, symport rules) with length at most k. Similarly, notations $\mathbf{TSA}(k)$ and $\mathbf{TSS}(k)$ are introduced by changing division rules for separation rules. If there is no restriction about the length of communication rules, then we omit the term (k).

3.4.1 Feasiblility of Tissue-like P Systems

By using the dependency graph technique, it has been shown (see [60] for details) that only tractable problems can be efficiently solved by using families of recognizer tissue P systems with cell division (or with cell separation) and communication rules of length 1, that is,

$$\mathbf{P} = \mathbf{PMC}_{\mathbf{TDC}(1)} = \mathbf{PMC}_{\mathbf{TSC}(1)} \tag{3.1}$$

In particular, the classes $\mathbf{TDA}(1)$ and $\mathbf{TDS}(1)$ are feasible.

By using the simulation technique, it has been shown that any family of recognizer tissue P systems with communication rules which solves a decision problem can be efficiently simulated by a family of basic recognizer P systems solving the same problem (see [40] for details). This simulation allows us to transfer the result about the limitations in computational power, from the model of basic cell-like P systems to the case of tissue P systems (see [58] for details), that is,

$$\mathbf{P} = \mathbf{PMC}_{\mathbf{TC}}. \tag{3.2}$$

By using the algorithmic technique, it has been shown that only tractable problems can be efficiently solved by using families of recognizer tissue P systems with cell separation with communication rules with length at most 2 (see [120] for details), that is,

$$\mathbf{P} = \mathbf{PMC}_{\mathbf{TSC}(2)} \tag{3.3}$$

In particular, the following result holds: $\mathbf{P} = \mathbf{PMC}_{\mathbf{TSA}(2)}$. In this context and by using the same technique, it has been shown (see [83] for details) that the class of recognizer tissue P systems with cell separation and allowing only the use of symport rules as communication rules, is feasible, that is,

$$\mathbf{P} = \mathbf{PMC}_{\mathbf{TSS}} \tag{3.4}$$

In particular, $\mathbf{P} = \mathbf{PMC}_{\mathbf{TSS}(3)}$ also holds.

In the definition of tissue P systems a special alphabet (the alphabet of the environment) is considered. Specifically, the elements of that alphabet are assumed to appear at the initial configuration of the system in an arbitrary large number of copies. This property seems an unfair tool when designing efficient solutions to computationally hard problems in the framework of membrane computing, by performing a space–time trade-off. In this context, in [138] and [87] the role of the environment in the framework of tissue P systems (with cell division and with cell separation, respectively) has been studied from a computational complexity point of view. Specifically, on the one hand, by using the simulation technique, it has been shown (see [138] for details) that the environment is irrelevant when we use cell division, that is, for each $k \geq 1$ we have:

$$\mathbf{PMC}_{\mathbf{TDC}(k)} = \mathbf{PMC}_{\widehat{\mathbf{TDC}}(k)} \tag{3.5}$$

As a consequence we have the class $\widehat{\mathbf{TDC}}(1)$ is feasible. On the other hand, by using the algorithmic technique, it has been shown (see [87] for details) that the class of recognizer tissue P systems without environment and with cell separation is feasible, that is, only tractable problems can be efficiently solved:

$$\mathbf{P} = \mathbf{PMC}_{\widehat{\mathbf{TSC}}} \tag{3.6}$$

3.4.2 Presumably Efficient Classes of Tissue-like P Systems

In order to establish the presumably efficiency of a class $\mathcal{F}$ of recognizer tissue P systems, the basic technique consists of designing a polynomial time solution of an **NP**–complete problem by using a family of systems from $\mathcal{F}$.

The `HAM-CYCLE` problem, a well known **NP**-complete problem [49], is the following: *given a directed graph, to determine whether or not there exists a Hamiltonian cycle in the graph.* In [143] it has been shown that the `HAM-CYCLE` problem can be solved in a uniform way and in polynomial time in by a family of recognizer tissue P systems with cell division and communication rules with length at most 2, that is, `HAM-CYCLE` $\in \mathbf{PMC}_{\mathbf{TDC}(2)}$. Then, bearing in mind that for each $k \geq 1$ we have $\mathbf{PMC}_{\mathbf{TDC}(k)} = \mathbf{PMC}_{\widehat{\mathbf{TDC}}(k)}$ we deduce that for each $k \geq 2$ the class $\widehat{\mathbf{TDC}}(k)$ is efficient. It is worth pointing out that the referred solution to `HAM-CYCLE` can be slightly adapted so that all communication rules are of symport type, it suffices to consider that these rules have length at most 3, that is, `HAM-CYCLE` $\in \mathbf{PMC}_{\mathbf{TDS}(3)}$.

The `SAT` problem, a well known **NP**-complete problem [49], is the following: *given a Boolean formula in conjunctive normal form, determine whether or not there exists an assignment to its variables on which it evaluates to true.* In [135] it has been shown that the `SAT` problem can be solved in a uniform way and in polynomial time by a family of recognizer tissue P systems with cell separation and communication rules with length at most 3, that is, `SAT` $\in \mathbf{PMC}_{\mathbf{TSC}(3)}$. It is worth pointing out that the referred solution to `SAT` can be slightly adapted so that all communication rules are of antiport type, it suffices to consider that these rules have length at most 3, that is, `SAT` $\in \mathbf{PMC}_{\mathbf{TSA}(3)}$.

3.4.3 Frontiers of the Presumable Efficiency in Tissue-like P Systems

Next, based on the results from previous sections, we present different frontiers of the efficiency in terms of syntactical ingredients of recognizer tissue P systems. We can classify them into four categories: the length of communication rules, the direction of such communication rules, or the use of some kind of rules or the environment.

1. The class **TC** is feasible and the classes **TDC** and **TSC** are presumably efficient. Thus, the kind of rules provides a borderline of the efficiency in the sense that, in the framework of recognizer tissue P systems, allowing the use of division rules or separation rules amounts passing from the feasibility to the presumably efficiency.

2. The class **TDC**(1) is feasible and the class **TDC**(2) is efficient. Thus, in the framework of recognizer tissue P systems with cell division, passing from 1 to 2 in the bound of the length of communication rules amounts to passing from feasibility to presumably efficiency.

3. The class **TDA**(1) is feasible and the class **TDA**(3) is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell division and allowing only antiport rules as communication rules, passing from 1 to 3 in the allowed length of communication rules amounts to passing from feasibility to presumably efficiency.

4. The class **TDS**(1) is feasible and the class **TDS**(3) is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell division and allowing only symport rules as communication rules, passing from 1 to 3 in the allowed length of communication rules amounts to passing from feasibility to presumably efficiency.

5. The class **TSC**(2) is feasible and the class **TSC**(3) is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell separation, passing from 2 to 3 in the allowed length of communication rules amounts to passing from from feasibility to presumably efficiency.

6. The class **TSC**(2) is feasible and the class **TDC**(2) is presumably efficient. Thus, in the framework of recognizer tissue P systems with the length of communication rules bounded by 2, allowing division rules instead of separation rules amounts to passing from feasibility to presumably efficiency.

7. The class $\widehat{\mathbf{TSC}}(3)$ is feasible and the class $\mathbf{TSC}(3)$ is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell separation and communication rules with length at most 3, the use or not of the classical environment amounts to passing from feasibility to presumably efficiency.

8. The class $\mathbf{TSA}(2)$ is feasible and the class $\mathbf{TSA}(3)$ is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell separation and using only antiport rules as communication rules, passing from 1 to 3 in the allowed length of communication rules amounts to passing from feasibility to presumably efficiency.

9. The class $\mathbf{TSS}(3)$ is feasible and the class $\mathbf{TSA}(3)$ is presumably efficient. Thus, in the framework of recognizer tissue P systems with cell separation and communication rules with length at most 3, replacing the restriction of using only symport rules as communication rules, by the opposite restriction (i.e. imposing all communication rules to be of antiport type), amounts to passing from feasibility to presumably efficiency.

10. For each $k \geq 2$, the class $\widehat{\mathbf{TSC}}(k)$ is feasible and the class $\widehat{\mathbf{TDC}}(k)$ is presumably efficient. Thus, in the framework of recognizer tissue P systems without environment whose communication rules have length at most $k \geq 2$, allowing division rules instead of separation rules amounts to passing from feasibility to presumably efficiency.

In this context, there is an open problem: determine whether or not the classes $\mathbf{TDS}(2)$ and $\mathbf{TDA}(2)$ are feasible or presumably efficient.

Feasible	Presumably Efficient	
$\mathbf{TC}$	$\mathbf{TDC}$	$(kind\ of\ rules)$
$\mathbf{TC}$	$\mathbf{TSC}$	$(kind\ of\ rules)$
$\mathbf{TDC}(1)$	$\mathbf{TDC}(2)$	$(length\ of\ rules)$
$\mathbf{TDA}(1)$	$\mathbf{TDA}(3)$	$(length\ of\ rules)$
$\mathbf{TDS}(1)$	$\mathbf{TDS}(3)$	$(length\ of\ rules)$
$\mathbf{TSC}(2)$	$\mathbf{TSC}(3)$	$(length\ of\ rules)$
$\mathbf{TSC}(2)$	$\mathbf{TDC}(2)$	$(kind\ of\ rules)$
$\widehat{\mathbf{TSC}}(3)$	$\mathbf{TSC}(3)$	$(environment)$
$\mathbf{TSA}(2)$	$\mathbf{TSA}(3)$	$(length\ of\ rules)$
$\mathbf{TSS}(3)$	$\mathbf{TSA}(3)$	$(direction\ of\ rules)$
$\widehat{\mathbf{TSC}}(k), k \geq 2$	$\widehat{\mathbf{TDC}}(k), k \geq 2$	$(kind\ of\ rules)$

Table 3.1: Frontiers of the Efficiency

4
Spiking Neural P systems

4.1 Introduction

Spiking Neural P systems (SN P systems, for short) are a variant of P systems, corresponding to a shift from *cell-like* to *neural-like* architectures. Formally introduced in [67], they take inspiration from the way in which neurons in the brain exchange information by means of the synaptic propagation of electrical impulses called *action potentials* or *spikes*. In what follows, we describe the basics of neuron structure and spikes propagation process (additional useful references can be found in [67]).

Neurons are specialized cells existing in virtually all kind of animals, characterized by their electrical excitability and the presence of synapses, complex membrane junctions that transmit signals to other cells. Neurons are assisted by glial cells, which provide structural and metabolic support—neurons and glial cells together constituting the nervous system. Neurons are very diverse, but the structure and functioning of a "typical" neuron can be described as follows. Structurally, three kind of elements are present in neurons, the soma or cell body, a set of dendrites and the axon. Axon and dendrites are filaments that extrude from the soma. Dendrites typically branch profusely, getting thinner with each branching, and extending their farthest branches a few hundred micrometers from the soma. The axon leaves the soma at a swelling called the axon hillock, and can extend for great distances, giving rise to hundreds of branches. Unlike dendrites, an axon usually maintains the same diameter as it extends. The soma may give rise to numerous dendrites, but never to more than one axon. Neurons are connected via synapses. A typical synapse is a contact between the axon of one neuron and a dendrite or soma of another. Synapses allow the transmission of neuronal signals among neurons: signals from other neurons are received by the soma and dendrites; signals to other neurons are transmitted by the axon.

A neuronal signal is an electric pulse called action potential or spike. Neuronal signals are typically identical, having an amplitude of about 100 mV and a duration of 1-2 ms, with pulse shape not changing as the action potential propagates along the axon. As such, neuronal signal themselves do not carry any information. Instead, the information is encoded in the number and timing of the spikes. According to the amount of pulses received by a neuron during a period of time, such pulses may have an inhibitory or excitatory effect over the neuron. In the first case, the neuron does not generate any pulse in response to its input spikes. In the excitatory case, the neuron generates a pulse as well, which originates at the soma and propagates rapidly along the axon, activating synapses onto other neurons as it goes by. There exists a refractory period associated to neurons: even with very strong input, it is impossible to excite a second spike during or immediately after a first one. The minimal (time) distance between two spikes defines the refractory period of the neuron. Working as described above, a neuron generates impulses at regular or irregular intervals, giving place to a sequence of action potentials called *spike train*. It is worth pointing out that, according to the spikes being identical and the described firing mechanism, while the size and the shape of a spike generated by a neuron is independent of the input of the neuron, the time when the neuron fires depends on its input.

To conclude with the biological aspects, we now address the synapse activation mechanism. Synapse activation produces a propagation of the action potential from the presynaptic neuron to the postsynaptic neuron. How this activation and propagation takes place depends on the synapse type. The most common type of synapse in the vertebrate brain is the chemical synapse. When an action potential arrives at a synapse, it triggers a complex chain of bio-chemical processing steps leading to the release of neurotransmitters from the presynaptic neuron into the postsynaptic neuron. As soon as transmitter molecules have reached the postsynaptic side, they will be detected by specialized receptors in the postsynaptic cell membrane, and through specific channels, the ions from the extracellular fluid flow into the target cell. The ion influx, in turn, leads to a change of the membrane potential at the postsynaptic side so that, in the end, the chemical signal is translated into an electrical response. The voltage response of the postsynaptic neuron to a presynaptic action potential is called the postsynaptic potential.

SN P systems incorporate the key elements of the structure and functioning of neurons and synapses described above. In these systems, cells, which are called *neurons*, are placed in the nodes of a directed graph, called the *synapse graph*. The content of any neuron consists of copies of a single object type, called *spike*. Every neuron may also have associated a number of *firing* and *forgetting* rules. Firing rules allow a neuron to send information to its neighbours by means of *spikes*, which are accumulated at target neurons. Executing a firing rule involves removing a certain amount of spikes from the neuron and emitting a spike that is replicated along the outgoing synapses and finally stored at target neurons. This is accomplished in a two-stage process: firstly the spikes are removed from the neuron, and after a specific period of time, determined by the rule, the neuron emits the output spike. During this period, the neuron becomes “closed” (inactive): it does not accept new spikes and cannot “fire” any (firing or forgetting) rule. The period of time is specified by a delay parameter associated to the firing rule. On the other hand, forgetting rules simply remove a certain amount of spikes from the neuron, with no spike being emitted. Applicability of a rule is determined by checking the neuron content against a regular expression associated to the rule. If more than one rule is applicable, then one of them is non-deterministically chosen. As usually happens in membrane systems, a global clock is assumed, which marks the evolution of the system and makes it work in a synchronized way. While individual neurons works sequentially (only a rule at most can be executed at any time by a neuron), the system as a whole works in parallel, since different neurons can execute rules simultaneously.

It is easy to see that the previous model captures, in a general way, the structure and functioning of neurons and synapses. Since in real neurons the shape and size of impulses are not important, having a single object type, the spike, is enough. Following this, the inhibitory or excitatory effect of the received impulses over a neuron is determined by the number of spikes received over time by such neuron so, in the model, a neuron accumulates spikes to count the number of received impulses. When the amount of spikes within the neuron reaches certain levels, firing or forgetting rules become applicable. Executing a firing rule corresponds to the case in which impulses have an excitatory influence over the neuron: a certain amount of spikes is removed from the neuron and a single spike is sent along the outgoing synapses, which models the synapse activation process. Executing a forgetting rule corresponds, in turn, with the inhibitory scenario: spikes are removed from the neuron and no spike is sent out.

The model specified above was originally introduced in [67] and we will refer to it as the *classic* model. Since the classic model introduction, many variants of SN P systems have been developed and their computational properties studied. In this chapter we review the SN P system variants considered within the scope of this dissertation. To this end, the classic model is formally presented firstly in Section 4.2 and, subsequently, variants are specified in an *incremental* way, that is, by constructing them as *extensions* of the classic model. In this way, Section 4.3 covers in detail the variants of interest. Finally, relevant theoretical results involving the considered variants are covered in Section 4.5. It is worth pointing out that the incremental selected approach to present the SN P systems variants, while intended to be as handy as possible, has the side effect of presenting some variants in a slightly different way than in the corresponding introduction in the literature. To minimize this, all the needed references are provided.

4.2 Basic Spiking Neural P systems

This model was introduced in [67]. Its definition follows.

Definition 4.1. *A basic SN P system of degree $m \geq 1$ is a tuple of the form:*

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

where:

1. *$O = \{a\}$ is the singleton alphabet (a is called* spike*);*

2. *$\sigma_1, \sigma_2, \ldots \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$ where:*

   - *$n_i \geq 0$ is the initial number of spikes contained in σ_i;*
   - *R_i is a finite set of rules of the two following forms:*

     (1) $E/a^c \to a; d$, with E a regular expression over O, $c \geq 1, d \geq 0$;

     (2) $a^s \to \lambda$, with $s \geq 1$ and the restriction $a^s \notin L(E)$ for any rule of type (1) $E/a^c \to a; d \in R_i$;

3. *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq n$, is the synapse graph, which defines synapses among neurons;*

4. *$in, out \in \{1, 2, \ldots, m\}$ are the input and output neurons respectively;*

Type (1) rules, with the form $E/a^c \to a; d$, are called *standard* firing (or spiking) rules. The term standard refers to the fact that only one spike copy appears in the right hand side of the rule, which corresponds with the biological interpretation previously discussed. For simplicity, we will refer to them simply as firing rules. As mentioned above, E is a regular expression over O. When $L(E) = a^c$, the rule can be written as $a^c \to a; d$. On the other hand, d is a non-negative integer known as the rule delay. When $d = 0$, the rule can be written as $E/a^c \to a$. If both $L(E) = a^c$ and $d = 0$, the rule can be written as $a^c \to a$. Finally, type (2) rules, with the form $a^s \to \lambda$, are called forgetting rules.

As usually happens with other P system variants, each neuron has a *label*. Notation σ_i is used to refer to a neuron labelled by i. Synapses are noted as (σ_i, σ_j), meaning a synapse from σ_i (called the *source* neuron) to σ_j (called the *target* neuron).

Graphical representation of a SN P system usually consists in a directed graph describing the initial structure of the system. In this directed graph, nodes correspond to neurons, and are labelled after them, while directed arcs correspond to synapses and model the spike flow. Within neurons, initial number of spikes and rules are drawn. The system output is represented by an outgoing synapse, coming from the output neuron, which is not connected to any other neuron of the system, meaning that it is connected to the environment. Similarly, the system input is represented by an ingoing synapse, getting to the input neuron, which is not connected to any other neuron, meaning that it is connected to the environment. For instance, the following system (appearing in [63]):

$\Pi = (O, \sigma_1, \sigma_2, \sigma_3, syn, in, out)$, with:

$O = \{a\}$,

$\sigma_1 = (2, \{a^2/a \to a; 0,\ a \to \lambda\})$,

$\sigma_2 = (1, \{a \to a; 0,\ a \to a; 1\})$,

$\sigma_3 = (3, \{a^3 \to a; 0,\ a \to a; 1,\ a^2 \to \lambda\})$,

$syn = \{(1,2), (1,3), (2,1), (2,3)\}$

$in = 1$.

$out = 3$.

corresponds to the graphical representation given in Fig. 4.1.

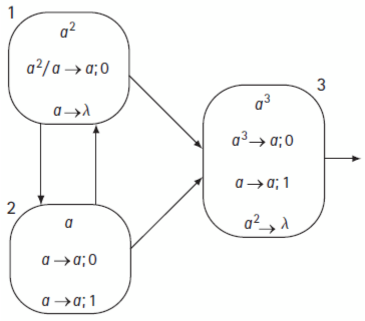


Figure 4.1: A simple SN P system.

In what follows, we specify SN P systems semantics. SN P systems are synchronized devices. A global clock is assumed, which marks the functioning of the system. SN P systems operate in a non-deterministic maximally parallel way with the following special remark: at any time instant t, each neuron operates sequentially, since at most only one of the applicable rules over the neuron is applied, which is non-deterministically chosen from the set of applicable rules for the neuron at t. Nevertheless, neurons, as a whole, operate in parallel, since all the neurons with a selected active rule fire their rules simultaneously. To complete the semantics specification, we now address the concepts of applicability and application for firing and forgetting rules.

Applicability

- Given a neuron σ_i containing k spikes, with $k \geq 0$, at a time instant t, it is said that a firing rule $E/a^c \to a; d \in R_i$ is applicable over σ_i at t if and only if the following conditions hold:
  (a) σ_i is not executing any rule;
  (b) $k \geq c$;
  (c) $a^k \in L(E)$;
- Given a neuron σ_i containing k spikes, with $k \geq 0$, at a time instant t, it is said that a forgetting rule $a^s \to \lambda$ is applicable over σ_i at t if and only if the following conditions hold:

(a) σ_i is not executing any rule;

(b) $k = s$;

A neuron checks for applicable rules whenever it receives new spikes or completes a rule execution. Applicable rules are also said to be active or enabled. Neurons having applicable, active or enabled rules are said to be active or enabled.

It is possible for two firing rules $E_1/a_1^c \rightarrow a; d_1$ and $E_2/a_2^c \rightarrow a; d_2$ belonging to R_i to become simultaneously applicable, since $L(E_1) \cap L(E_2) \neq \emptyset$. On the other hand, when a forgetting rule $a^s \rightarrow \lambda$ belonging to R_i becomes active, only that rule can be applied, since (a) $a^s \notin L(E)$ for any firing rule $E/a^c \rightarrow a; d$ belonging to R_i; and (b) the number of spikes in σ_i must be equal to s. When a neuron has more than one applicable rule at instant t, one, and only one, is non-deterministically selected to be applied.

Application

- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, application of an active firing rule $r \equiv E/a^c \rightarrow a; d \in R_i$ over σ_i at t implies the following:
  - At instant t, neuron σ_i fires rule r and c spikes are removed from σ_i immediately, thus $k - c$ spikes are left in the neuron.
  - If $d \geq 1$, from instant t to $t + d - 1$, σ_i becomes closed and cannot accept incoming spikes, that is, any spike sent to σ_i in the interval $[t, t + d - 1]$ is lost.
  - At instant $t+d$, neuron σ_i becomes open (it accepts incoming spikes) and, simultaneously, it emits a spike (or simply spikes). This spike is replicated onto the outgoing synapses and sent to the target neurons. Spikes reach the target neurons immediately.
  - At instant $t + d + 1$ neuron σ_i can check again for applicable rules.
- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, application of an active forgetting rule $r \equiv a^s \rightarrow \lambda \in R_i$ over σ_i at t implies the following:
  - At instant t, neuron σ_i fires rule r and s spikes are removed from σ_i immediately.
  - At instant $t + 1$ neuron σ_i can check again for applicable rules.

Let us notice that for firing rules, if $d = 0$ the neuron fires (removing the corresponding spikes) and spikes (sending out a spike) at same instant t, never being actually closed, so it can accept incoming spikes arriving at that instant t. Also, it is important to distinguish between the terms *firing*, the action of start a rule application, and *spiking*, the action of sending out a spike. Neurons applying a firing rule fire and spike (thus firing rules are also called *spiking rules*), while neurons applying a forgetting rule fire, but do not spike.

In what follows, we define the concepts of configuration, transition step and computation for SN P systems.

- A *configuration* of a SN P system Π at instant i with $i \geq 0$, denoted by C_i, is an instantaneous description of Π at a given time instant. Usually, C_i denotes the configuration of Π at time instant i, with $i \geq 0$ and C_0 being the initial configuration of the system. The content of a configuration consists in, for each neuron in the system, the number of spikes contained in such neuron and the number of time instants left for the neuron to become open (zero if already open). The initial configuration of Π is defined as $C_0 = \langle n_1/0, n_2/0, \ldots, n_m/0 \rangle$, meaning that in the initial configuration all the neurons of Π are open.

- A *transition step* (or simply *step*) of a SN P system Π at instant i, with $i \geq 1$, is the state transition of Π from configuration C_{i-1} to C_i, denoted by $C_{i-1} \Rightarrow C_i$. The transition step is performed by selecting and applying rules in a synchronous maximal parallel way as described above. Usually, the term transition step i is used to refer to the transition step taking place at instant i.

- A *computation* $\mathcal{C}$ of a SN P system Π is any (finite or infinite) sequence of configurations starting from the initial configuration C_0 and verifying that the i-th term ($i \geq 1$) of the sequence is obtained from the previous one in one transition step. In this way, the computation $\mathcal{C}$ can be written as $\mathcal{C} = C_0 \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \ldots$ If the sequence is infinite, the computation is said to be a *non-halting* computation. If the sequence is finite, the computation is said to be a *halting* computation. The last term of a halting computation is a *halting configuration*, that is, a configuration where all neurons are open and no rules can be applied.

In what follows, we address the notion of *valid* computation for SN P systems. This idea was originally considered for asynchronous SN P systems in [20], but can be extended to other scenarios.

- Given a SN P system $\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out)$, with $\sigma_i = (n_i, R_i)$ and a vector of non-negative integers, $(v_1, v_2, \ldots, v_m)$, a computation of Π is considered valid if and only if it reaches a halting configuration where each neuron σ_i contains exactly v_i spikes. Whenever a neuron σ_i contains exactly v_i spikes along a computation, it is said that such neuron is in a *valid state*. This way of defining a successful computation, based on a vector $(v_1, v_2, \ldots, v_m)$, is called μ-halting.

- It is possible to assure that provided that all neurons reach a valid state in some configuration, this configuration is halting, and thus the computation is valid. To accomplish this, the following constraint is imposed on the rule sets of Π: if $v_i > 0$, then $a^{v_i} \notin L(E)$ for any regular expression E appearing in a rule of neuron σ_i.

- Finally, it is possible to consider a *weak* definition of valid computation. In this case, only some distinguished neurons are required to reach valid state at halting time (if no neurons are required to reach valid state at halting time, then all halting computations are valid). Let us notice that in the weak case, if we want to assure that any computation where all the distinguished neurons reach valid state is a valid computation, imposing that $a^{v_i} \notin L(E)$ for any regular expression E appearing in a rule of any distinguished neuron σ_i is not enough in general. This comes from the fact that non-distinguished neurons can still spike and modify the content of some distinguished neuron.

Next, we discuss (according to [128]) how the output of SN P systems can be encoded and provided. Given a SN P system Π with output neuron *out*, when *out* spikes to the environment it is said that Π spikes. Moreover, it is possible to define the following:

- Let $\gamma = C_0 \Rightarrow C_1 \Rightarrow C_2 \Rightarrow \ldots$ be a computation of Π, with C_0 the initial configuration and $C_{i-1} \Rightarrow C_i$ the i-th step of γ.

- Let $COM(\Pi)$ be the set of all computations of Π.

- Let $HCOM(\Pi)$ be the set of all halting computations of Π.

- Let $st(\gamma) = \langle t_1, t_2, \ldots \rangle$ with $1 \leq t_1 < t_2 < \ldots$ be the *spike train* of γ, defined as the sequence of steps of γ when Π spikes. This sequence can be finite (for halting computations) or infinite (for non-halting computations). Alternatively, the spike train can be defined as a binary sequence containing a 1 for steps i when Π spikes and a 0 for the rest. Within the scope of this dissertation, we will use the term *natural* spike train when referring to the first definition and the term *binary* spike train when referring to the second one. Both definitions are interchangeable, since natural and binary spike trains can be generated from each other.

- Let $ST(\Pi)$ be the set of all spikes trains for all possible computations of Π. If Π is deterministic, $ST(\Pi)$ is singleton.

A system Π can encode and provide its output in many ways from $ST(\Pi)$:

- As the spike train itself, $ST(\Pi)$.
- As the set of instants where Π spikes, for every possible computation, denoted by $T(\Pi)$, and formally defined as follows:
  - Let $T(\gamma) = \{t_1, t_2, \dots \mid st(\gamma) = \langle t_1, t_2, \dots, \rangle\}$ be the set of time instants when Π spikes during γ.
  - Let $T(\Pi) = \bigcup_{\gamma \in COM(\Pi)} T(\gamma)$.

  It is also possible to consider only halting computations, defined as
  $$T^h(\Pi) = \bigcup_{\gamma \in HCOM(\Pi)} T(\gamma).$$
- As the set of numbers made of the difference between all consecutive time instants when Π spikes, for every possible computation, defined as
  $$N(\Pi) = \{n \mid n = t_i - t_{i-1}\}$$
  with $i \geq 2 \wedge \gamma \in COM(\Pi) \wedge st(\gamma) = \langle t_1, t_2, \dots \rangle \wedge |st(\gamma)| \geq 2$.
- As the set of numbers made of the difference between all alternate time instants when Π spikes, for every possible computation, defined as
  $$N^a(\Pi) = \{n \mid n = t_{2i} - t_{2i-1}\}$$
  with $i \geq 1 \wedge \gamma \in COM(\Pi) \wedge st(\gamma) = \langle t_1, t_2, \dots \rangle \wedge |st(\gamma)| \geq 2$.
- As the set of numbers made of the difference between the first two time instants when Π spikes, for every possible computation, defined as
  $$N_2(\Pi) = \{t_2 - t_1\}$$
  with $\gamma \in COM(\Pi) \wedge st(\gamma) = \langle t_1, t_2, \dots \rangle \wedge |st(\gamma)| \geq 2$.

- As the set of numbers made of the difference between the first k time instants when Π spikes, for every possible computation, such that
  - Π spikes at least k times *(weak case)*, defined as
  $$N_k(\Pi) = \{n \mid n = t_i - t_{i-1}\}$$
  with $2 \leq i \leq k \land \gamma \in COM(\Pi) \land st(\gamma) = \langle t_1, t_2, \dots \rangle \land |st(\gamma)| \geq k$.
  - Π spikes exactly k times *(strong case)*, defined as
  $$N_{\underline{k}}(\Pi) = \{n \mid n = t_i - t_{i-1}\}$$
  with $2 \leq i \leq k \land \gamma \in COM(\Pi) \land st(\gamma) = \langle t_1, t_2, \dots \rangle \land |st(\gamma)| = k$.
- As the set of numbers made of the difference between all consecutive time instants when Π spikes, for every possible non-halting computation, defined as
$$N_\omega(\Pi) = \{n \mid n = t_i - t_{i-1}\}$$
with $i \geq 2 \land \gamma \in COM(\Pi) \land st(\gamma)$ infinite.
- As the set of numbers generated by the difference between all possible time intervals, defined as
$$N_{all}(\Pi) = \bigcup_{k \geq 2} N_k(\Pi) \cup N_\omega(\Pi).$$
- From the previous definitions, we can also consider:
  - Only halting computations: $N_k^h(\Pi), N_{\underline{k}}^h(\Pi), N_{all}^h(\Pi), k \geq 2$.
  - Alternate times differences: $N_k^a(\Pi), N_{\underline{k}}^a, N_\omega^a(\Pi), N_{all}^a(\Pi), k \geq 2$.
  - Only halting computations with alternate time differences: $N_\alpha^{ha}(\Pi), \alpha \in \{\omega, all\} \cup \{k \mid k \geq 2\}$ and $N_{\underline{k}}^{ha}(\Pi), k \geq 2$.

Also, it is possible to consider SN P systems with multiple output neurons, and define the output of the system accordingly. Assuming $k \geq 1$, a k-output SN P system with output neurons $o_1, \dots, o_k$ generates a k-tuple $(n_1, \dots, n_k) \in \mathbb{N}^k$ if, starting from the initial configuration, there is a sequence of steps such that each output neuron o_i encodes n_i as output and then the system eventually halts.

Finally, we discuss how the input of SN P systems is encoded and provided. Without loss of generality, we consider only the case of inputting natural numbers, since entering other kind of inputs can be reduced to this case by using an adequate encoding. Given a SN P system Π with input neuron *in*, we can define the following:

- Let $cod(n)$ be a function that encodes any natural number n into a mutiset defined over O.
- Let $\Pi + cod(n)$ be the SN P system obtained from inputting $cod(n)$ into Π through input membrane *in*.

There exist several strategies to encode and provide the input. Some of them are:

- $cod(n) = a^n$; $cod(n)$ is entered in a single step into *in*.
- $cod(n) = a^2$; $cod(n)$ is entered as the difference between the time instants when the first two spikes arrive into *in*.
- $cod(n) = a^b$ with b equal to the number of occurrences of 1 in the binary codification of n; $cod(n)$ is entered into *in* as a input binary spike train corresponding to the binary codification of n.

4.2.1 SN P systems as number accepting devices

Definition 4.2. *A SN P system of degree $m \geq 1$ working as a number accepting device is a tuple of the form:*

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

where:

1. *$O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in$ are defined as in classic SN P systems.*
2. *Semantics of Π are defined as in classic SN P systems.*
3. *Given an encoding function cod, the following holds:*
   - *$n \in \mathbb{N}$ $\underline{\text{is accepted}}$ by Π, if and only if any computation of $\Pi + cod(n)$ is a $\underline{\text{halting}}$ computation.*
   - *$n \in \mathbb{N}$ $\underline{\text{is not accepted}}$ by Π, if and only if any computation of $\Pi + cod(n)$ is a $\underline{\text{non-halting}}$ computation.*

4.2.2 SN P systems as number generating devices

Definition 4.3. *A SN P system of degree $m \geq 1$ working as a number generating device is a tuple of the form:*

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

where:

1. *$O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, out$ are defined as in classic SN P systems.*
2. *Semantics of Π are defined as in classic SN P systems.*
3. *$n \in \mathbb{N}$ is generated by Π if and only if n is encoded as output by some computation of Π.*

In the case of SN P systems working as number generating devices, one of the most common ways of encoding the output is as the time difference between the first two instants when Π spikes.

4.2.3 SN P systems as number computing devices

Definition 4.4. *A SN P system of degree $m \geq 1$ working as a number computing device for a (possibly partial) computable function $f : \mathbb{N} \to \mathbb{N}$ is a tuple of the form:*

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

where:

1. *$O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out$ are defined as in classic SN P systems.*
2. *Semantics of Π are defined as in classic SN P systems.*
3. *Given an encoding function cod, the following holds:*
   - *If $f(x)$ is undefined, any computation of $\Pi + cod(x)$ is a non-halting computation.*
   - *If $f(x)$ is defined, with $f(x) = y, y \in \mathbb{N}$, any computation of $\Pi + cod(x)$ is a halting computation and encodes y as output.*

In the case of SN P systems working as number computing devices, one of the most common ways of inputting a number x is encoding it as the time difference between the instants when the first two spikes are sent into the input membrane *in*. The output result is commonly encoded as the time difference between the first two instants when Π spikes. It is also possible to compute functions of the form: $f : \mathbb{N}^k \to \mathbb{N}$ by either considering a SN P system with k input membranes or using a input single membrane and codifying the input vector $(i_1, i_2, \dots, i_k)$ as a binary sequence of the form: $z = 10^{i_1-1}10^{i_2-1}1 \dots 10^{i_k-1}$.

4.2.4 Recognizer SN P systems

SN P systems can also be used as decision problem solvers. In order to present this functioning mode, it is necessary to define the concepts of SN P systems working in recognizer mode (also called recognizer SN P systems) and decision problems.

Definition 4.5. *A recognizer SN P system of degree $m \geq 1$ is a tuple of the form:*

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

where:

1. *$O, \sigma_1, \sigma_2, \dots, \sigma_m, syn, in, out$ are defined as in classic SN P systems.*
2. *Semantics of Π are defined as in classic SN P systems.*
3. *All computations of Π halt.*
4. *Given an encoding cod, the following holds:*
   - *$n \in \mathbb{N}$ $\underline{\text{is recognized}}$ by Π if and only if any computation of $\Pi + cod(n)$ encodes answer $\underline{\text{yes}}$ as output.*
   - *$n \in \mathbb{N}$ $\underline{\text{is not recognized}}$ by Π if a only if any computation $\Pi + cod(n)$ encodes answer $\underline{\text{no}}$ as output.*

With respect to the way in which answers $\underline{\text{yes}}$ and $\underline{\text{no}}$ are encoded, several alternatives exist. For example:

- As the number of spikes contained in *out* when the system halts; for example, one spike meaning $\underline{\text{yes}}$ and no spike meaning $\underline{\text{no}}$.

- As the number of spikes sent to the environment during the computation; for example, only one spike, which is sent at the last transition step, meaning $\underline{\text{yes}}$, and no spikes sent meaning $\underline{\text{no}}$.

Now we discuss complexity classes of SN P systems regarding the solution of decision problems by means of SN P systems, according to [79].

Definition 4.6. *Let $X = (I_X, \theta_X)$ be a decision problem and $g : \mathbb{N} \to \mathbb{N}$ a computable function. It is said that X is solvable by a family $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$ of recognizer SN P systems, in time bounded by g, in a non-deterministic and uniform way (this is denoted by $X \in NSN(g)$), if the following holds:*

- *The family $\mathbf{\Pi}$ is polynomially uniform by Turing Machines, that is, there exists a deterministic Turing Machine that working in polynomial time constructs $\Pi(n)$ from $n \in \mathbb{N}$.*
- *There exist a pair of polynomial time computable functions (cod, s) over I_X such that:*
  - *The pair (cod, s) is a polynomial encoding of X in $\mathbf{\Pi}$, that is, for each instance $u \in I_X$, $s(u)$ is a natural number and $cod(u)$ is a valid input multiset of $\Pi(s(u))$.*
  - *The family $\mathbf{\Pi}$ is g-bounded with respect to (X, cod, s), that is, for each instance $u \in I_X$, the $\underline{\text{minimum length}}$ of an accepting computation of $\Pi(s(u)) + cod(u)$ $\overline{\text{is bounded by}}$ $g(|u|)$.*
  - *The family $\mathbf{\Pi}$ is sound with respect to (X, cod, s), that is, for each instance $u \in I_X$, if there exists an accepting computation of $\Pi(s(u)) + cod(u)$, then $\theta_X(u) = 1$;*
  - *The family $\mathbf{\Pi}$ is complete with respect to (X, cod, s); that is, for each instance $u \in I_X$, if $\theta_X(u) = 1$ then there exists computation of $\Pi(s(u)) + cod(u)$ which is an accepting one.*

We say that a decision problem $X = (I_X, \theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$ of recognizer SN P systems, in a non-deterministic and uniform way (this is denoted by $X \in NPSN$), if there exists $k \in \mathbb{N}$ such that X is solvable by the family $\mathbf{\Pi}$ in time bounded by a polynomial x^k, in a non-deterministic and uniform way.

Definition 4.7. *Let $X = (I_X, \theta_X)$ be a decision problem and $g : \mathbb{N} \to \mathbb{N}$ a computable function. It is said that X is solvable by a family $\mathbf{\Pi} = (\Pi(u))_{u\in I_X}$ of recognizer SN P systems, in time bounded by g, in a non-deterministic and semi-uniform way (this is denoted by $X \in NSN^*(g)$), if the following holds:*

- *The family* $\mathbf{\Pi}$ *is polynomially uniform by Turing Machines, that is, there exists a Deterministic Turing Machine that working in polynomial time constructs* $\Pi(u)$ *from the instance* $u \in I_X$*.*
- *The family* $\mathbf{\Pi}$ *is g-bounded with respect to* X*, that is, for each instance* $u \in I_X$*, the* $\underline{\textit{minimum length}}$ *of an accepting computation of* $\Pi(u)$ *is bounded by* $g(|u|)$*.*
- *The family* $\mathbf{\Pi}$ *is sound with respect to* X*, that is, for each instance* $u \in I_X$*, if there exists an accepting computation of* $\Pi(u)$*, then* $\theta_X(u) = 1$*.*
- *The family* $\mathbf{\Pi}$ *is complete with respect to* X*, that is, for each instance* $u \in I_X$*, if* $\theta_X(u) = 1$ *then there exists a computation of* $\Pi(u)$ *which is an accepting one.*

We say that a decision problem $X = (I_X, \theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$ of recognizer SN P systems, in a non-deterministic and semi-uniform way (this is denoted by $X \in NPSN^*$), if there exists $k \in \mathbb{N}$ such that X is solvable by the family $\mathbf{\Pi}$ in time bounded by a polynomial x^k, in a non-deterministic and semi-uniform way.

4.3 Spiking Neural P systems variants

Previously, we have presented the SN P system model known as the *classic* model. In what follows, we discuss some variants of that model falling within the object of this dissertation. Variants are obtained from classic SN P systems by extending/restricting their syntactical elements or changing the way in which the system operates, that is, their semantics.

4.3.1 SN P systems with extended rules

This variant was introduced in [24]. Its definition follows.

Definition 4.8. *A SN P system with extended rules of degree* $m \geq 1$ *is a tuple of the form:*

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

where:

1. O, syn, in, out *are defined as in classic SN P systems.*

2. $\sigma_1, \sigma_2, \ldots \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$ where:

- *$n_i \geq 0$ is the initial number of spikes contained in σ_i;*
- *R_i is a finite set of <u>extended rules</u> of the form $E/a^c \to a^p; d$, with E a regular expression over O, $c \geq 1, p \geq 0, c \geq p, d \geq 0$ and with the constraint $p = 0 \Rightarrow d = 0$.*

Extended rules $E/a^c \to a^p; d$ can be written in abbreviated form, as follows: (1) if $E = a^c$, E is omitted; (2) if $c = 1$, c is omitted; (3) if $p = 1$, p is omitted; (4) if $p = 0$, the right hand side of the rule is λ; and (5) if $d = 0$, d is omitted. The possible abbreviated forms for extended rules can be found in Table 4.1.

Attending to their syntactical structure, extended rules can be categorized in the following way:

- Bounded rules. These are rules of the form $a^i/a^c \to a^p; d$, where $1 \leq c \leq i, p \geq 0, c \geq p, d \geq 0$.

- Unbounded rules. These are rules of the form $a^i(a^j)^*/a^c \to a^p; d$, where $i \geq 0, j \geq 1, c \geq 1, p \geq 0, c \geq p, d \geq 0$.

Attending to the kind of associated rules, neurons can be categorized in the following way:

- Bounded neurons. Every associated rule is bounded.

- Unbounded neurons. Every associated rule is unbounded.

- General neurons. They have associated at least one bounded and one unbounded rule.

Attending to the kind of neurons, SN P systems with extended rules can be categorized in the following way:

- Bounded systems. All their neurons are bounded.

- Unbounded systems. All their neurons are unbounded.

- General systems. They contain at least one general neuron.

Condition(s)	Abbreviated form	Denomination
-	$E/a^c \to a^p; d$	Extended rule
$E = a^c$	$a^c \to a^p; d$	-
$c = 1, p = 1$	$E/a \to a; d$	-
$c = 1, p = 0$	$E/a \to \lambda$	-
$p = 1$	$E/a^c \to a; d$	(Standard) Firing rule
$p = 1, d = 0$	$E/a^c \to a$	(Standard) Firing rule without delay
$p = 0, d = 0$	$E/a^c \to \lambda$	Extended forgetting rule
$d = 0$	$E/a^c \to a^p$	Extended rule without delay
$E = a^c, c = 1, p = 1$	$a \to a; d$	-
$E = a^c, c = 1, p = 0$	$a \to \lambda$	-
$E = a^c, p = 1$	$a^c \to a; d$	-
$E = a^c, p = 1, d = 0$	$a^c \to a$	-
$E = a^c, p = 0, d = 0$	$a^c \to \lambda$	(Standard) Forgetting rule
$E = a^c, d = 0$	$a^c \to a^p$	-
$c = 1, p = 1, d = 0$	$E/a \to a$	-
$c = 1, p = 0, d = 0$	$E/a \to \lambda$	-
$E = a^c, c = 1, p = 1, d = 0$	$a \to a$	-
$E = a^c, c = 1, p = 0, d = 0$	$a \to \lambda$	-

Table 4.1: Abbreviated forms for extended rules.

As with classic SN P systems, SN P systems with extended rules work in a non-deterministic maximally parallel way under the guidance of a global clock, with at most only one of the applicable rules over a given neuron being applied at each transition step. Such rule is non-deterministically chosen from the set of applicable rules for the neuron. With respect to the applicability and application of extended rules, the following holds:

Applicability

- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, it is said that a extended rule $E/a^c \to a^p; d \in R_i$ is applicable over σ_i at t if and only if the following conditions hold:

  (a) σ_i is not executing any rule;

  (b) $k \geq c$;

  (c) $a^k \in L(E)$;

Application

- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, application of an active extended rule $r \equiv E/a^c \to a^p; d \in R_i$ over σ_i at t implies the following:

– At instant t, neuron σ_i fires rule r and c spikes are removed from σ_i immediately, thus $k - c$ spikes are left in the neuron.
– If $d = 0$, at instant t neuron σ_i emits p spikes (if $p = 0$, neuron σ_i does not spike). These spikes are replicated onto the outgoing synapses and sent to the target neurons. Spikes reach the target neurons immediately. At the same instant, since σ_i is open, it accepts incoming spikes.
– If $d > 0$ then:
    ∗ From instant t to $t+d-1$, σ_i becomes closed and cannot accept incoming spikes, that is, any spike sent to σ_i in the interval $[t, t + d - 1]$ is lost.
    ∗ At instant $t + d$, neuron σ_i becomes open (it accepts incoming spikes) and, simultaneously, emits p spikes. These spikes are replicated onto the outgoing synapses and sent to the target neurons. Spikes reach the target neurons immediately.
– At instant $t + d + 1$ neuron σ_i can check again for applicable rules.

The concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems.

4.3.2 Asynchronous SN P systems

In what follows, we discuss some variants of SN P systems working in asynchronous mode. In this mode, neurons with applicable rules at a given computation step may choose not to fire. The “choosing mechanism” may vary, giving place to several asynchronous SN P systems variants.

4.3.2.1 Asynchronous SN P systems (classic model)

From both mathematical and neurological points of view, it is rather natural to consider asynchronous SN P systems (ASNPS, for short). This model was introduced in [20]. In this kind of systems a global clock, marking the time for all neurons, is still present, but neurons work asynchronously in the following way: application of rules in each neuron is not obligatory, that is, if a neuron has one or more enabled rules in a given instant, it may (non-deterministically) choose whether to fire or not one of them. If the neuron does not fire, new spikes can come into the neuron rendering some of the previously applicable rules non-applicable. If a rule is still applicable through successive time instants, it can be selected to fire at any time, independently of how much time has passed. Taking this into account, the concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems.

4.3.2.2 1-Asynchronous SN P systems

Let us consider the concept of *dormant step* for SN P systems. In a given configuration of the system, if no neuron is applicable but at least one neuron is closed, the system is said to be in a *dormant step*. If there is at least one applicable neuron in a given configuration, the system is said to be in a *non-dormant step*.

A 1-Asynchronous SN P system is an asynchronous SN P system where it is required that in every non-dormant step at least one rule is applied in the system. On the other hand, a strongly 1-Asynchronous SN P Systems is an asynchronous SN P system which has the property that in a valid computation, every step of the computation has at least one applicable neuron, unless the SN P system is in a halting configuration. Otherwise (i.e., there is a step in which there is no applicable rule), the computation is viewed as invalid and no output from such a computation is included in the generated set by the system.

4.3.2.3 Limited Asynchronous SN P systems

In classic ASNPS there is no restriction imposed on the number of successive time instants that a given neuron can hold firing an enabled rule. Nevertheless, from the biological point of view, it is natural to consider a *bound* imposed on the number of time units that an enabled rule remains unfired, since in nature given a long enough time interval, an enabled chemical reaction will conclude within this interval.

Taking into consideration such biological motivation, Limited Asynchronous SN P systems (LASNPS, for short) were introduced in [124]. In these systems a global bound $b \geq 2$ (imposed on all rules) is specified in such a way that if one (and only one) 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 now the rule non-applicable, then the computation continues in the new circumstance (maybe other rules are enabled now). If more than one rule is applicable, the neuron non-deterministically chooses and fires one of them in the interval t to $t+b-1$.

In addition to the above, there is a special remark with respect to functioning of LASNPS not explicitly given in [124]: whenever a neuron σ_i has to check for new applicable rules, the *implicit counter* controlling the number of steps that σ_i has been holding the execution of its applicable rules is *reset*. Let us recall from classic SN P systems that σ_i has to check for new applicable rules whenever it receives new spikes or completes a rule execution. The described remark implies that at any instant t any applicable rule of a given neuron σ_i has been waiting to be applied the same amount of time.

In LASNPS, a configuration is described by the number of spikes present in each neuron, the number of time units for neurons to become open as well as the time that has elapsed for each rule since it became applicable. Transition steps are carried out in a similar way to classic SN P systems, but according to the limited asynchronous firing mechanism described above. With respect to the output of the system, the following applies: since an enabled rule at instant t can be applied at any moment in the time interval t to $t+b-1$ (that is, in the b steps starting from t), a variable spike train can be produced. For instance, the binary spike train of the system can have a variable number of occurrences of 0 between two occurrences of 1, with this variation going from 0 to b. Consequently, defining the result of a computation as the number of steps between two consecutive spikes (as usual in synchronous systems) is useless. In its place, the result of a computation is usually defined as the total number of spikes sent into the environment by the output neuron.

4.3.2.4 Asynchronous SN P systems with Local Synchronization

In classic ASNPS, neurons (asynchronously) fire their rules in a independent way with respect to each other. Nevertheless, from the biological point of view it is natural to consider *interrelations* between neurons in terms of synchronicity. In a biological neural system, neurons involved in carrying out some specific functions synchronously cooperate with each other to achieve their goals. Groups of neurons like this can exhibit different topologies, such as motifs with 4-5 neurons and communities with 12-15 neurons. On the other hand, non-related neurons in terms of functionality work in an independent, or asynchronous, way.

Taking into consideration such biological motivation, Asynchronous SN P systems with Local Synchronization (ASNPSLS, for short) were introduced in [165]. In ASNPSLS synchronous interrelations between neurons are described via a family of sets denoted *Loc*. Each element of the family is a set of locally synchronous neurons, that is, neurons that work synchronously with each other. As such, each element of *Loc* is called a locally synchronous set (ls-set, for short). A neuron can be placed in zero, one or more ls-sets. Given an ASNPSLS with m neurons $\sigma_1, \sigma_2, \ldots, \sigma_m$, the family *Loc* can be formally defined in the following way:

$$Loc = \{loc_1, loc_2, \ldots, loc_l\} \subseteq \mathcal{P}(\{\sigma_1, \sigma_2, \ldots, \sigma_m\}),$$

where $\mathcal{P}(\{\sigma_1, \sigma_2, \ldots, \sigma_m\})$ is the power set of $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$.

In ASNPSLS, behaviour of individual neurons is similar to classic Asynchronous SN P systems: at any instant t a neuron with enabled rules non-deterministically chooses whether to fire or not one of such rules. Nevertheless, neurons in the same ls-set fire in a synchronous way: if a enabled neuron within a ls-set loc_j fires, then all neurons in loc_j that have enabled rules must fire at that instant t. As such, it is possible that all neurons from loc_j remain unfired even if they have enabled rules, i.e., all neurons from loc_j may remain still, or all neurons from loc_j with enabled rules fire at the same step. Hence, neurons work asynchronously at the global level, while working synchronously within each ls-set.

The concepts of configuration, transition step and computation can be defined in a similar way to classic ASNPS taking into account the locally synchronous firing mechanism described above.

4.3.3 Sequential Spiking Neural P systems

In what follows, we discuss some variants of SN P systems working in sequential mode, introduced in [66]. In this mode, the maximal parallelism present in classic SN P systems is dropped. The motivation is that the maximal parallelism way of rule application (which is widely used in membrane computing) is rather non-realistic in some cases. As such, sequential variants of SN P systems have been introduced, in which only some neurons with applicable rules are chosen to fire, according to a certain strategy. It is precisely this "choosing mechanism strategy" what gives place to different sequential SN P systems variants. Let us notice that semantics of these systems are defined in a similar way to classic SN P system (including the concepts of applicability, application, configuration, transition step, computation), but incorporating the peculiarities of each strategy.

1. **Sequential SN P systems with pure-seq strategy.**

   This variant corresponds to the *classic sequential model* introduced in [66]. The term pure-seq refers to the fact that these systems are totally (pure) sequential: at each computation step, one and only one of the neurons with applicable rules fires. Such neuron is non-deterministically chosen. If there is no applicable rule, then the system is dormant until a rule becomes applicable. However, the clock will keep on ticking. This kind of systems are called *strongly* sequential if at every step, there is at least one neuron with an applicable rule.

Alternatively to the classic model, there exist other sequential variants where the firing strategy depends on the number of spikes stored in each neuron at each computation step. These variants were introduced in [147], and are briefly described next.

2. **Sequential SN P systems with max-seq strategy.**

   In this variant, at each computation step the set of neurons with applicable rules is computed. From this set, neurons containing the maximum number of spikes are chosen. From these chosen neurons, one and only one neuron is non-deterministically selected to fire.

3. **Sequential SN P systems with max-pseudo-seq strategy.**

   In this variant, at each computation step the set of neurons with applicable rules is computed. From this set, neurons containing the maximum number of spikes are chosen. All these chosen neurons are selected to fire. The term pseudo-seq refers to the fact that this strategy is not fully sequential, since more than one rule can be fired.

4. **Sequential SN P systems with min-seq strategy.**

   In this variant, at each computation step the set of neurons with applicable rules is computed. From this set, neurons containing the minimum number of spikes are chosen. From these chosen neurons, one and only one neuron is non-deterministically selected to fire.

5. **Sequential SN P systems with min-pseudo-seq strategy.**

   In this variant, at each computation step the set of neurons with applicable rules is computed. From this set, neurons containing the minimum number of spikes are chosen. All these chosen neuron are selected to fire. Again, the term pseudo-seq refers to the fact that this strategy is not fully sequential, since more than one rule can be fired.

4.3.4 SN P systems with division and budding rules

This variant was introduced in [119] and addresses an increment in the number of neurons and synapses along computations. Its definition follows.

Definition 4.9. *A SN P system with neuron division and budding rules of degree $m \geq 1$ is a tuple of the form:*

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

where:

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

Rules of type (1) are extended rules, as previously defined for SN P systems with extended rules. Rules of type (2) and (3) are division and budding rules respectively, which enable neuron creation. Applying neuron division and budding rules allows for two or more neurons to share the same label so, in general, σ_i denotes any neuron with label $i \in H$. Since new neurons can be created, and neuron can share labels, rules are associated to neuron labels, instead of neurons themselves, as in the case of classic SN P systems. In this way, R_i is the set of the rules associated with neurons labelled by i.

Division rules create a pair of new membranes from a previously existing one (that disappears). These membranes are placed *in parallel*, in the sense that they inherit the ingoing and outgoing synapses of the original neuron. When applying a budding rule, a new neuron is created from an existing neuron, which is preserved. Both neurons are placed *serially*, in the sense that outgoing synapses from the existing neuron are transferred to the newly created neuron and a new synapse is created from the existing neuron to the new neuron.

In SN P systems with neuron division and budding rules, the system structure (both the number of neurons and of synapses between them) is not fixed and may change along the computation. The way in which synapses are created is determined by the neuron division and budding rules as well as the synapse directory *syn*, which is a synapse graph of neuron labels. The initial structure of the system is determined by *syn* in the following way: (a) for each different label $i \in H$ appearing in *syn*, an initial neuron with label i is placed in a node of the SN P system directed graph; and (b) for each arc $(i, j) \in syn$ a synapse is created from the neuron labelled by i to the neuron labelled by j. Let us stress the fact that, according to this, in the initial configuration, only one neuron (at most) with a given label may exist. The synapse dictionary also guides creation of new synapses whenever a new neuron is created, in the following way: if a new neuron with label i is created, then for every pre-existing neuron with label g, (a) if $(i, g) \in syn$ then a synapse from the new neuron with label i to the pre-existing neuron with label g is created; and (b) if $(g, i) \in syn$ then a synapse from the pre-existing neuron with label g to the new neuron with label i is created.

In what follows, we discuss semantics of SN P systems with budding and division rules, starting with the applicability and application of division and budding rules (extended rules were previously addressed).

Applicability

- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, it is said that a division rule $[E]_i \to []_j \parallel []_k$ is applicable over σ_i at t if and only if the following conditions hold:
  (a) σ_i is not executing any rule;
  (b) $a^k \in L(E)$;
  (c) $\nexists\, \sigma_g \,|\, g \in \{j, k\} \wedge ((g, i) \in syn \vee (i, g) \in syn)$;
  (d) $i \notin \{in, out\}$.
- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, it is said that a budding rule $[E]_i \to []_i/[]_j$ is applicable over σ_i at t if and only if the following conditions hold:

(a) σ_i is not executing any rule;

(b) $a^k \in L(E)$;

(c) $\nexists \sigma_j \mid (j, i) \in syn \vee (i, j) \in syn$;

(d) $i \neq out$.

Application

- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, application of an active division rule $r \equiv [E]_i \to []_j \parallel []_k$ over σ_i at t implies the following:
  - At instant t, neuron σ_i fires rule r. Neuron σ_i becomes closed (it cannot accept incoming spikes).
  - At instant t, neuron σ_i is splitted into two newly created closed empty neurons, labelled by j and k respectively, and denoted by σ_j and σ_k. Ingoing and outgoing synapses of σ_i are replicated over σ_j and σ_k in the following way:
    - ∗ For each ingoing synapse (σ_l, σ_i) between a pre-existing neuron σ_l and neuron σ_i, synapses (σ_l, σ_g), with $g \in \{j, k\}$, are created.
    - ∗ For each outgoing synapse (σ_i, σ_l) between neuron σ_i and a pre-existing neuron σ_l, synapses (σ_g, σ_l), with $g \in \{j, k\}$, are created.
  - At instant t new synapses involving neurons σ_j and σ_k and pre-existing neurons are created from syn in the following way:
    - ∗ For every pre-existing neuron σ_l, if $(l, g) \in syn$, with $g \in \{j, k\}$, synapse (σ_l, σ_g) is created.
    - ∗ For every pre-existing neuron σ_l, if $(g, l) \in syn$, with $g \in \{j, k\}$, synapse (σ_g, σ_l) is created.
  - At instant $t + 1$, neurons σ_j and σ_k become open (they accept incoming spikes) and can check for new applicable rules.
- Given a neuron σ_i containing k spikes, with $k \geq 1$, at a time instant t, application of an active budding rule $r \equiv [E]_i \to []_i/[]_j$ over σ_i at t implies the following:
  - At instant t, neuron σ_i fires rule r. Neuron σ_i becomes closed (it cannot accept incoming spikes). All the spikes in σ_i are consumed.

- At instant t, neuron σ_i generates a newly created closed empty neuron, labelled by j and denoted by σ_j. Ingoing synapses of σ_i are preserved, while outgoing synapses of σ_i are transferred to σ_j in the following way:
  - For each outgoing synapse (σ_i, σ_l) between neuron σ_i and a pre-existing neuron σ_l, synapse (σ_j, σ_l) is created.
- At instant t a new synapse (σ_i, σ_j) is created.
- At instant t new synapses involving neuron σ_j and pre-existing neurons are created from syn in the following way:
  - For every pre-existing neuron σ_l, if $(l, j) \in syn$, synapse (σ_l, σ_j) is created.
  - For every pre-existing neuron σ_l, if $(j, l) \in syn$, synapse (σ_j, σ_l) is created.
- At instant $t + 1$, neurons σ_i and σ_j become open (they accept incoming spikes) and can check for new applicable rules.

The concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems, with the following remark: since the structure of the system is not fixed, every configuration must store the instantaneous description of the directed graph associated to the system.

4.3.5 SN P systems with astrocytes

In what follows, we discuss a variant of classic SN P systems taking inspiration from one important biological element existing in the nervous system structure, the *astrocyte*.

Astrocytes, also known collectively as astroglia, are characteristic star-shaped glial cells in the brain and spinal cord that connect to neighbouring synapses. An astrocyte connects to a synapse in the space between the presynaptic and postsynaptic terminals giving place to the so-called "tripartite synapse" [7], with one single astrocyte being able to connect to different synapses in this way. Astrocytes propagate intercellular Ca_2^+ waves over long distances in response to stimulation and, similarly to neurons, release transmitters (called gliotransmitters) in a Ca_2^+-dependent manner. Moreover, within the dorsal horn of the spinal cord, activated astrocytes have the ability to respond to almost all neurotransmitters [61] and, upon activation, release a multitude of neuroactive molecules that influences neuronal excitability. That is, astrocytes can sense the neuronal activity related to their attached synapses and carry out a synaptic modulation in an excitatory or inhibitory way. Other important functionalities of astrocytes include biochemical support of endothelial cells that form the blood-brain barrier, provision of nutrients to the nervous tissue, maintenance of extracellular ion balance, and a role in the repair and scarring process of the brain and spinal cord following traumatic injuries. As a result of all of this, astrocytes constitute an important area of research within the field of neuroscience and, consequently, they also are an interesting element to take inspiration from within the membrane computing paradigm.

In SN P systems with astrocytes, new syntactical ingredients are introduced to model astrocytes. One astrocyte can be attached to one or more synapses and one synapse can be attached to zero, one or more astrocytes. Attached synapses to a given astrocyte are said to be *controlled* by the astrocyte. Astrocytes sense the spike traffic passing along their controlled synapses and can have an excitatory or inhibitory influence on such traffic. In general, excitatory influence implies allowing the spike traffic to go along the controlled synapses, while the inhibitory influence implies destroying such traffic, with the spikes being removed from the system. When a synapse is controlled by two or more astrocytes, only if every astrocyte has an excitatory influence on the synapse the spikes passing along that synapse survive. In this way, when neurons spike, the emitted spikes are transmitted along synapses and reach target neurons unless they are intercepted by astrocytes.

Several variants of SN P systems with astrocytes have been defined and studied. Within the scope of this dissertation we describe the SN P systems with hybrid astrocytes.

4.3.5.1 SN P systems with hybrid astrocytes

SN P systems with *hybrid* astrocytes (SNPSHA, for short) correspond to variant introduced in [123]. In these systems, astrocytes show an excitatory or inhibitory influence on controlled synapses by comparing the spike traffic in such synapses against a threshold associated to the astrocyte. Since they can show either an excitatory or inhibitory influence, they are called *hybrid* astrocytes within the scope of this dissertation (this term not being used in [123]). Definition of this model follows.

Definition 4.10. *A Spiking Neural P system with hybrid astrocytes of degree (m, l), with $m \geq 1$, $l \geq 1$, is a tuple of the form:*

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, ast_1, ast_2, \ldots, ast_l, syn, in, out),$$

where:

1. *$O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out$ are defined as in classic SN P systems.*
2. *$ast_1, ast_2, \ldots, ast_l$ are astrocytes, with $ast_j \, (1 \leq j \leq l)$ of the form*

$$ast_j = (syn_{ast_j}, t_j),$$

where:

- *$syn_{ast_j} \subseteq syn$ is the set of $\underline{\text{controlled synapses}}$ by the astrocyte;*
- *$t_j \in \mathbb{N}$ is the $\underline{\text{threshold}}$ of the astrocyte;*

Hybrid astrocytes are graphically represented as diamond-shaped figures, with a number inside corresponding to the threshold and lines connected to their controlled synapses (see [123] for more details).

Semantics of SNPSHA follows from the classic model, but incorporating hybrid astrocytes behaviour. Astrocytes of this kind work as follows. For an astrocyte ast_j, if at intant t there are k spikes in total passing along its neighbouring synapses syn_{ast_j} then a) if $k > t_j$, the astrocyte ast_j has an inhibitory influence on the neighbouring synapses, and the k spikes are simultaneously suppressed (that is, the spikes are removed from the system); b) if $k < t_j$, the astrocyte ast_j has an excitatory influence on the neighbouring synapses, all spikes survive and get to their destination neurons, reaching them simultaneously; and c) if $k = t_j$, the astrocyte ast_j non-deterministically chooses an inhibitory or excitatory influence on the neighbouring synapses. It is possible for two or more astrocytes to control the same synapse. In this case, only if every astrocyte has an excitatory influence on the synapse the spikes passing along that synapse survive.

The concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems.

4.3.6 SN P systems with anti-spikes

In what follows, we discuss a variant of SN P systems that addresses the inhibitory nature of some neural impulses by introducing an additional object type, the *anti-spike* denoted by $\overline{a}$ and named after the anti-matter. Anti-spikes are present in neurons and participate in firing and forgetting rules along with usual spikes. Spikes and anti-spikes cannot exist simultaneously in the same neuron, since they annihilate each other, as an *implicit* rule of the form $a\overline{a} \rightarrow \lambda$ exists in every neuron.

This model was introduced in [122]. Its definition follows.

Definition 4.11. *A SN P system with anti-spikes of degree $m \geq 1$ is a tuple of the form:*

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

where:

1. *$O = \{a, \overline{a}\}$ is the alphabet (a is called* spike*, $\overline{a}$ is called* anti-spike*);*

2. *$\sigma_1, \sigma_2, \ldots \sigma_m$ are neurons of the form $\sigma_i = (n_i, R_i)$ where:*

   - *$n_i \geq 0$ is the initial number of $\underline{\text{spikes}}$ contained in σ_i;*
   - *R_i is a finite set of rules of the following forms:*

     (0) $a\overline{a} \to \lambda$

     (1) $E/b^c \to b'; d$ with E a regular expression over a or over $\overline{a}$ (but not over a and $\overline{a}$ simultaneously)

     (2) $b^s \to \lambda$ with $s \geq 1$ and $b^s \notin L(E)$ for any type (1) rule in R_i

     verifying that $b, b' \in \{a, \overline{a}\}$;

3. *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$, with $(i, i) \notin syn$ for $1 \leq i \leq n$ is the synapse graph, defining the synapses among neurons;*

4. *$in, out \in \{1, 2, \ldots, m\}$ are the input and output neurons respectively;*

5. *for any type (1) rule $E/b^c \to b'; d \in R_i$, if $i = out$ then $b' = a$.*

Semantics of SN P systems with anti-spikes follows from the classic model, with the following remarks:

- Type (0) rules are *annihilation* rules. They are implicitly defined in every neuron of the system, so they are not specified when describing the model. A type (0) rule is applied with top-most priority in a maximal way taking zero time units to complete whenever the neuron receives spikes or anti-spikes. Consequently, spikes and anti-spikes cannot exist together in the same neuron.

- Type (1) and (2) rules are firing and forgetting rules with anti-spikes respectively. Applicability and application for these rules are defined as in the classic model, with the peculiarity that regular expressions may involve spikes or anti-spikes (but not both) and that spikes and anti-spikes may be consumed/produced when applying the rules.

The concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems. With respect to the input and output of the system, only spikes are allowed to be entered into/sent out the system.

4.4 Computational power of SN P systems

In what follows, we present major theoretical results concerning the computational power of SN P systems variants. Proofs are not provided, but can be found in the corresponding references.

Definition 4.12. *Let $Spik_{\alpha}^{\beta}P_m(rule_k, cons_p, forg_q, dley_r, outd_s)$ be the family of sets $N_{\alpha}^{\beta}(\Pi)$ for all systems with at most m neurons, each neuron having at most k rules, each of the spiking rules consuming at most p spikes and with a delay at most r, each forgetting rule removing at most q spikes and an outdegree of the synapse graph at most s. Then $\alpha \in \{2, \underline{2}\}$ and β is either omitted or it belongs to the set $\{h, \underline{h}\}$. Each of the parameters m, k, p, q is replaced with $*$ if it is not bounded. Notation $rule_k^*$ is used when firing rules are of the form $E/a^c \to a; d$ with the regular expression of the forms $E = a^c$ or $E = a^*$. Prefix D is used to refer to deterministic systems and subscript acc to refer to systems working in accepting mode.*

The following results come from [64]:

Theorem 4.1. *$Spik_{\underline{2}}^{\beta}P_*(rule_3, cons_4, forg_a, dley_0, outd_2) = NRE$, where $\beta \in \{h, \underline{h}\}$ or β is omitted and $a = 5$ for $\beta = \underline{h}$, otherwise $a = 4$.*

Theorem 4.2. *$DSpik_{\underline{2}acc}^{\beta}P_*(rule_2, cons_3, forg_2, dley_0, outd_2) = NRE$, where $\beta \in \{h, \underline{h}\}$ or β is omitted.*

Theorem 4.3. *$Spik_2P_*(rule_k, cons_p, forg_0) = NRE$, for all $k \geq 2, p \geq 3$.*

Theorem 4.4. *$Spik_{\underline{2}}^{\beta}P_*(rule_3, cons_4, forg_4, dley_0, outd_2) = NRE$, where either $\beta = h$ or β is omitted.*

Theorem 4.5. *$Spik_{\underline{2}}^{\beta}P_*(rule_2^*, cons_2, forg_1, dley_2, outd_2) = NRE$, where either $\beta = h$ or β is omitted.*

Theorem 4.6. *$Spik_{\underline{2}}^{h}P_*(rule_3^*, cons_2, forg_{15}, dley_2, outd_2) = NRE$.*

Theorem 4.7. *$DSpik_{\underline{2}acc}^{\beta}P_*(rule_2, cons_2, forg_1, dley_2, outd_2) = NRE$, where either $\beta = h$ or β is omitted.*

Definition 4.13. *A partially blind k-output multicounter machine (k-output PBCM) is a k-output multicounter machine where the registers/counters cannot be tested for zero. The output counters are non-decreasing. The other counters can be incremented by 1 or decremented by 1, but if there is an attempt to decrement a zero counter then the computation aborts (i.e. the computation becomes invalid). By definition, a successful generation of a k-tuple requires that the machine enters an accepting state with all non-output counters set to zero. Such counter machines are known to be not universal.*

Definition 4.14. *A k-output monotonic CM is a non-deterministic machine with k counters, all of which are output counters. The counters are initially zero and can only be incremented by 1 or 0 (they cannot be decremented). When the machine halts in an accepting state, the k-tuple of values in the k-counter is said to be generated by the machine. Clearly, a k-output monotonic CM is a special case of a PBCM, where all the counters are output counters and all the instructions are addition instructions. It is known that a set* $Q \subseteq \mathbb{N}^k$ *is semilinear if and only if it can be generated by a k-output monotonic CM.*

The following results come from [20]:

Theorem 4.8. *A set* $Q \subseteq \mathbb{N}^k$ *is recursively enumerable if and only if it can be generated by an asynchronous k-output general SN P system with extended rules. The result holds for systems with or without delays.*

Theorem 4.9. *A set* $Q \subseteq \mathbb{N}^k$ *is generated by a k-output PBCM if and only if it can be generated by an asynchronous k-output unbounded SN P system without delays. Hence, such SN P systems are not universal.*

Theorem 4.10. *A set* $Q \subseteq \mathbb{N}^k$ *is generated by a k-output PBCM if and only if it can be generated by an asynchronous k-output unbounded SN P system with delays. Hence, such SN P systems are not universal.*

Theorem 4.11. *1-Asynchronous unbounded k-output SN P systems (with delays) are universal.*

Theorem 4.12. *Strongly 1-Asynchronous unbounded k-output SN P systems with delays and k-output PBCMs are equivalent.*

Theorem 4.13. *A set* $Q \subseteq \mathbb{N}^k$ *can be generated by a k-output monotonic CM if and only if it can be generated by a k-output asynchronous bounded SN P system with extended rules. The result holds for systems with or without delays.*

Definition 4.15. *Let $\mathbb{N}_{gen}^{asynch}$ be the set of numbers generated in the limited asynchronous way by an SN P system with general rules.*

The following result comes from [124]:

Theorem 4.14. $\mathbb{N}_{gen}^{asynch} = NRE$.

Definition 4.16. *Let $NSpik_{out}P_m^{locsyn}(\alpha, ls)$ with $\alpha \in \{gen, boun, unb\}$ and $ls \geq 0$, be the family of number sets generated by asynchronous SN P systems with local synchronization of type α, such that gen stands for general, boun for bounded and unb for unbounded, with at most m neurons, and local synchronization degree at most ls. If one of the parameters m and ls is not bounded, then it is replaced with $*$. The subscript out refers to the fact that the output of the systems is obtained by interpreting as computation result the count of all spikes sent into the environment.*

The following results come from [165]:

Theorem 4.15. $NSpik_{out}P_*^{locsyn}(boun, *) = NRE$.

Theorem 4.16. $NSpik_{out}P_*^{locsyn}(unb, *) = NRE$.

Theorem 4.17. $NSpik_{out}P_*^{locsyn}(boun, *) = SLIN$.

The following results come from [65]:

Theorem 4.18. *The following results hold for sequential SN P systems with delays:*

1. *Sequential k-output unbounded SN P systems with standard rules and strongly sequential k-output general SN P systems with standard rules are universal.*
2. *Strongly sequential k-output unbounded SN P systems with standard rules and k-output PBCMs are equivalent.*

The above results also hold for systems with extended rules.

Theorem 4.19. *Unbounded SN P systems with delays working in the max-seq mode are universal, even in the strongly sequential setting.*

Theorem 4.20. *Extended SN P systems working in max-seq mode with unbounded rules and no delays (thus strongly sequential) are universal.*

Theorem 4.21. *Unbounded SN P systems working in the max-pseudo-seq mode, without delays, are universal.*

Theorem 4.22. *A system of deterministic neurons working in a max-pseudo-seq manner (as a generator) is non-universal.*

Theorem 4.23. *A system of deterministic neurons working in a max-pseudo-seq manner (as an acceptor) is universal.*

Theorem 4.24. *Unbounded SN P systems without delays, working in the min-seq mode (hence strongly sequential), are universal. In this case we consider the output to be given as the number of spikes in a given neuron.*

4.5 Computational efficiency of SN P systems

In what follows, we present major theoretical results concerning the computational efficiency of SN P systems variants. Proofs are not provided, but can be found in the corresponding references.
The following results come from [80]:

Theorem 4.25. *Consider a (possibly universal) deterministic accepting SN P system* Π*, of degree* $m \geq 1$*, in which all the regular expressions are of the following restricted forms:* a^i*, with* $i \leq 3$*, or* $a(aa)^+$*. Then, any* t *steps of computation of* Π *can be simulated by a deterministic Turing machine in a time which is polynomial with respect to* t *and to the description size of* Π*.*

Corollary 4.1. *Polynomial size (with respect to the problem instance size) deterministic SN P systems cannot solve* ***NP****-complete problems in polynomial time, unless* $\boldsymbol{P} = \boldsymbol{NP}$*.*

The following result comes from [81]:

Theorem 4.26. *Non-deterministic SN P systems are able to solve in the semi-uniform setting any instance of* ***3-SAT*** *in constant time.*

The following result comes from [79]:

Theorem 4.27. *A uniform family* $\{\Pi_{SAT}(\langle n, m\rangle)\}_{n,m\in\mathbb{N}}$ *solves all the instances of* ***SAT(n,m)*** *in a number of steps which is linear in* n *and independent of* m*.*

Theorem 4.28. *A uniform family* $\{\Pi_{3-SAT}(n)\}_{n\in\mathbb{N}}$ *solves all the instances of* ***3-SAT(n)****.*

The following result comes from [80]:

Theorem 4.29. *A non-deterministic semi-uniform family of SN P systems solves any instance of* ***SUBSET SUM*** *in only two computation steps.*

The following result comes from [79, 81]:

Theorem 4.30. *A deterministic uniform family of SN P systems solves any instance of* ***SUBSET SUM****.*

The following result comes from [119]:

Theorem 4.31. *A deterministic uniform family* $\{\Pi_{SAT}(\langle n, m\rangle)\}_{n,m\in\mathbb{N}}$ *of SN P systems with budding and division rules solves any instance of* ***SAT(n,m)****.*

The following result comes from [123]:

Theorem 4.32. *Asynchronous extended SN P systems with hybrid astrocytes are universal.*

5 Fuzzy Reasoning SN P systems

5.1 Introduction

In classical logic, there are only two possible truth values, named *true*, represented by natural value 1, and *false*, represented by natural value 0. To the contrary, in fuzzy logic, truth values vary in a range of real numbers in $[0, 1]$ representing different degrees of truth. Fuzzy logic is derived from the theory of fuzzy sets, which extends the classical notion of set. This theory was proposed by Zadeh [207], with Klaua also developing a similar concept around the same time [75, 76]. A fuzzy set is a class of objects with a continuum grade of membership characterized by a membership (characteristic) function which assigns to each object existing in the universe of discourse a grade of membership in $[0, 1]$. Value 0 stands from *"the object does not belong to the set"*, value 1 stands for *"the object belongs to the set"*, while others values in the interval stand for different degrees of membership to the set.

Soon after the introduction of fuzzy sets, it was pointed out in [54], starting from the Zadeh approach, and in [76], starting from the Klaua approach, the intimate relation between fuzzy sets and non-classical logics, i.e. multiple-valued logics, since membership degrees could be considered as generalized truth values, i.e. as truth degrees. Fuzzy logic development is motivated by the need of providing a conceptual framework for dealing with the representation of common sense knowledge, since such knowledge is by its nature both lexically imprecise and non-categorical, that is, a framework that can address the issues of uncertainty and imprecision [208]. Consequently, fuzzy logic can be applied to scenarios where the nature of the information is uncertain.

Since its introduction, fuzzy set theory has been studied and applied extensively. Most of the early interests in fuzzy set theory addressed representing uncertainty in human cognitive processes, while nowadays fuzzy set theory is combined with other methods and applied to problems in engineering, business, medical and related health sciences, the natural sciences and other disciplines. As such, different computational models incorporating fuzzy logic elements have surfaced, oriented to deal with the aforementioned problems. Within the Membrane Computing framework, SN P system variants incorporating fuzzy logic elements have been defined. Such variants are collectively called Fuzzy Reasoning SN P systems (FRSN P systems, for short) and are intended to model fuzzy diagnosis knowledge and reasoning as required in fault diagnosis applications, although other practical scenarios have been also considered. A detailed survey on FRSN P systems can be found in [182], which is the main reference material for this chapter. Within the scope of this dissertation, two simulators have been developed for one of the aforementioned variants, which is called Fuzzy Reasoning SN P systems with real numbers (rFRSN P systems, for short). One of the simulators is intended to run on sequential platforms, while the other one runs on the GPU.

This chapter is structured as follows. Section 5.2 covers theoretical aspects about modelling problems within the fuzzy logic paradigm. Section 5.3 introduces FRSN P systems, focusing on the common features of this family of membrane systems. Section 5.4 addresses rFRSN P systems in detail. Finally, an application of rFRSN P systems within the area of fault diagnosis of power systems is briefly described.

5.2 Modelling problems in fuzzy logic

When modelling a problem within the fuzzy logic framework, usually a fuzzy knowledge base is built, were fuzzy propositions, expressing facts with an uncertain truth value and fuzzy production rules, representing logic consequences involving fuzzy propositions are used to analyse and determine the solution to the problem. Several approaches to define the fuzzy propositions can be taken. In what concerns to this work, we will consider the fuzzy production rules types as defined in [25]. These types are (1) simple fuzzy production rules, (2) composite fuzzy conjunctive rules in the antecedent, (3) composite fuzzy conjunctive rules in the consequent, (4) composite fuzzy disjunctive rules in the antecedent and (5) composite fuzzy disjunctive rules in the consequent. Their definitions follow:

A *simple fuzzy production rule* is of the form

$$\text{Type 1 } R_i\text{: IF } p_j \text{ THEN } p_k \text{ (CF=}\tau_i\text{)}$$

where R_i indicates the i-th fuzzy production rule; τ_i represents its certainty factor; p_j and p_k represents two propositions, each of which has a fuzzy truth value. If fuzzy truth values of propositions p_j and p_k are α_j and α_k, respectively, then $\alpha_k = \alpha_j * \tau_i$.

A *composite fuzzy conjunctive rule in the antecedent* is of the form

$$\text{Type 2 } R_i\text{: IF } p_1 \text{ and } \cdots \text{ and } p_{k-1} \text{ THEN } p_k \text{ (CF=}\tau_i\text{)}$$

where R_i indicates the i-th fuzzy production rule; τ_i represents its certainty factor; $p_1, \cdots, p_{k-1}$ are propositions in the antecedent part of the rule; p_k is the proposition in the consequent part of the rule. If fuzzy truth values of propositions $p_1, \cdots, p_{k-1}$ are $\alpha_1, \cdots, \alpha_{k-1}$, respectively, then $\alpha_k = \min(\alpha_1, \ldots, \alpha_{k-1}) * \tau_i$.

A *composite fuzzy conjunctive rule in the consequent* is of the form

$$\text{Type 3 } R_i\text{: IF } p_1 \text{ THEN } p_2 \text{ and } \cdots \text{ and } p_k \quad \text{(CF=}\tau_i\text{)}$$

where R_i indicates the i-th fuzzy production rule; τ_i represents its certainty factor; p_1 is a proposition in the antecedent part of the rule; $p_2, \cdots, p_k$ are propositions in the consequent part of the rule. If fuzzy truth value of proposition p_1 is α_1, then $\alpha_2 = \alpha_1 * \tau_i, \ldots, \alpha_k = \alpha_1 * \tau_i$.

A *composite fuzzy disjunctive rule in the antecedent* is of the form

$$\text{Type 4 } R_i\text{: IF } p_1 \text{ or } p_2 \text{ or } \cdots \text{ or } p_{k-1} \text{ THEN } p_k \text{ (CF=}\tau_i\text{)}$$

where R_i indicates the i-th fuzzy production rule; τ_i represents its certainty factor; $p_1, \cdots, p_{k-1}$ are propositions in the antecedent part of the rule; p_k is the proposition in the consequent part of the rule. If fuzzy truth values of propositions $p_1, \cdots, p_{k-1}$ are $\alpha_1, \cdots, \alpha_{k-1}$, respectively, then $\alpha_k = \max(\alpha_1, \ldots, \alpha_{k-1}) * \tau_i$.

A *composite fuzzy disjunctive rule in the consequent* is of the form

$$\text{Type 5 } R_i\text{: IF } p_1 \text{ THEN } p_2 \text{ or } \cdots \text{ or } p_{k-1} \text{ or } p_k \text{ (CF=}\tau_i\text{)}$$

where R_i indicates the i-th fuzzy production rule; τ_i represents its certainty factor; p_1 is a proposition in the antecedent part of the rule; $p_2, \cdots, p_{k-1}$ are propositions in the consequent part of the rule. This type of rules is unsuitable for knowledge representation due to the fact that it does not make any specific implication [25]. Thus, this type of rules are not considered in relation to FRSN P systems.

5.3 FRSN P systems generalities

Fuzzy Reasoning SN P systems (FRSN P systems, for short) comprise a somehow *exotic* family of SN P systems variants intended to model fuzzy diagnosis knowledge and reasoning as required in fault diagnosis applications. The term *exotic* comes from the fact that, although basic elements of SN P systems like a synapse graph of neurons that exchange spikes are still present in FRSN P systems, new ingredients are added to incorporate fuzzy reasoning elements, such as several types of neurons, spike potentials (modelled after fuzzy truth values) and a new firing mechanism. In this way, FRSN P systems put together desirable features (understandable, dynamical, synchronized, non-linear, non-deterministic, able to handle incomplete and uncertain information) required to model fault diagnosis problems.

To date, five types of FRSN P systems have been proposed: fuzzy reasoning spiking neural P systems with real numbers (rFRSN P systems, introduced in [130] and later updated in [181] and to be further investigated in [205, 206], fuzzy reasoning spiking neural P systems with linguistic terms (lFRSN P systems, introduced in [175]), adaptive fuzzy reasoning spiking neural P systems with real numbers (AFRSN P systems, introduced in [176] and further investigated in [102]), weighted fuzzy reasoning spiking neural P systems (WFRSN P systems, introduced in [177] and further investigated in [183]) and fuzzy reasoning spiking neural P systems with trapezoidal fuzzy numbers (tFRSN P systems, introduced in [179] and further expanded in [180, 184, 185]). A comprehensive survey on these variants can be found in [182]. Despite of the different existing FRSN P systems variants, all of them share common features, described as follows:

- FRSN P systems model propositions and rules of a fuzzy knowledge base. As such, two kinds of neurons exist: proposition neurons, modelling propositions in the fuzzy knowledge base, and rule neurons, modelling fuzzy production rules. Besides, rule neurons have associated a fuzzy truth value that can be interpreted as the confidence/certainty factor

(CF) of the fuzzy production rule. Finally, there exist both a set of input and output proposition neurons.

- With respect to the synapse graph, proposition neurons are always connected to rule neurons while rule neurons are always connected to proposition neurons. Besides, FRSN P systems are feed-forward systems, presenting an acyclic synapse graph. As such, (a) computations associated FRSN P systems always halt; and (b) any synapse can carry spikes at most once during a computation. Input proposition neurons have zero indegree, while output proposition neurons have zero outdegree.

- With respect to spikes, contrary to classic SN P systems, in FRSN P systems a neuron may contain at most one spike. Each spike has associated a fuzzy truth value (that we denote α) which can be interpreted as the potential value of the pulse/spike from the point of view of biological neurons. Nevertheless, contrary to biological neurons, the pulse value may differ among spikes. In the initial configuration of the FRSN P system every input proposition neuron contains exactly one spike, while the rest of neurons contain no spike. Neurons store the pulse value of its internal spike. If a neuron contains a spike, the pulse value α associated to its internal spike is not null, while if the neuron contains no spike, α is null. Neurons update the pulse value of its internal spike depending on pulse values of received spikes and the neuron kind.

- With respect to rules, neurons only have associated one rule, which is of the firing/spiking kind. As such, any FRSN P system is deterministic so only one possible computation is associated to its execution. Contrary to classic SN P systems, a neuron with zero outdegree cannot fire. When a neuron fires, it consumes its internal spike of pulse value α and sends out a spike which pulse value β depends on the neuron kind. Neurons fire in the following circumstances:

  – In the case of input proposition neurons, when the number of received pulses equals to one.
  – In the case of the rest of neurons, when the number of received pulses equals to the neuron number of ingoing synapses.

  As such, in general neurons must track the number of received pulses in order to determine when to fire.

5.4 rFRSN P systems

In rFRSN P systems fuzzy truth values are modelled with real numbers (other variants use either real numbers, linguistic terms or trapezoidal fuzzy numbers). In [130, 181] rFRSN P systems were shown to provide promising applications in the field of fault diagnosis of electrical systems. Also, for this variant, a matrix-based algorithm was provided which, when executed on parallel computing platforms, fully exploits the model maximally parallel capacities.

In what follows, we introduce syntax and semantics for rFRSN P systems. What we present next is a specification incorporating in a explicit way the general common features for FRSN P systems as described above. As such, this definition does not exactly match the one introduced in [130], since it aims to specify features of rFRSN P systems in the most formal possible way.

Definition 5.1. *A rFRSN P system of degree (l, q, n, s, k), with $l, k \geq 1$, $q, s \geq 0$ and $n \geq l + q + 1$, is a construct of the form:*

$$\Pi = (O, \sigma_1, \ldots, \sigma_l, \sigma_{l+1}, \ldots, \sigma_{l+q}, \sigma_{l+q+1}, \ldots, \sigma_n,$$
$$\sigma_{n+1}, \ldots, \sigma_{n+s}, \sigma_{n+s+1}, \ldots, \sigma_{n+k}, syn, in, out),$$

where:

1. *$O = \{a\}$ is the alphabet (a is called* spike*);*

2. *$\sigma_1, \ldots, \sigma_{n+k}$ are neurons, of the form $\sigma_i=(p_i, \alpha_i, \tau_i, r_i)$, $1 \leq i \leq n + k$, where:*

   - *$\sigma_1, \ldots, \sigma_n$ are proposition neurons, from which:*
     - *$\sigma_1, \ldots, \sigma_l$ are input proposition neurons;*
     - *$\sigma_{l+1}, \ldots, \sigma_{l+q}$ are internal proposition neurons;*
     - *$\sigma_{l+q+1}, \ldots, \sigma_n$ are output proposition neurons;*
   - *$\sigma_{n+1}, \ldots, \sigma_{n+k}$ are rule neurons, from which:*
     - *$\sigma_{n+1}, \ldots, \sigma_{n+s}$ are AND rule neurons;*
     - *$\sigma_{n+s+1}, \ldots, \sigma_{n+k}$ are OR rule neurons;*
   - *$p_i \in \mathbb{N}$ is the number of pulses received by σ_i;*
   - *$\alpha_i \in [0, 1]$ is a real fuzzy truth value representing the potential value of the pulse/spike contained in neuron σ_i such that if $\alpha_i = 0$ there is no spike, while if $\alpha_i > 0$ there is one (and only one) spike of pulse value α_i;*

- $\tau_i \in [0, 1]$ *is a real fuzzy truth value representing:*
  - *the truth value of the fuzzy proposition in case σ_i is a proposition neuron, and verifying in this case that (a) $\tau_i = \alpha_i$ and (b) $\tau_i = 0$ means "unknown";*
  - *the confidence factor (CF) of the fuzzy production rule in case σ_i is a rule neuron;*
- *r_i is a firing/spiking rule associated to σ_i, $r_i \equiv [E/a^\alpha \to a^\beta]_i$, with E being a regular expression over $\{a\}$, and α, β representing arbitrary real fuzzy truth values in $[0, 1]$ but ensuring that $\alpha \geq \beta$; assuming that $n_i = indegree(\sigma_i)$, rule r_i has the following form:*
  - *$[a^\alpha \to a^\alpha]_i$, if σ_i is an input proposition neuron;*
  - *$[a^{n_i}/a^\alpha \to a^{\alpha * \tau_i}]_i$ if σ_i is any other kind of neuron;*

3. *$syn \subseteq \{1, \ldots, n + k\} \times \{1, \ldots, n + k\}$ with $i \neq j$ for all $(i, j) \in syn$, $1 \leq i, j \leq n + k$, is the* directed synapse graph *between neurons; the following properties hold:*

   - *syn is acyclic;*
   - *syn is bipartite with respect to proposition and rule neurons, that is, for all $(i, j) \in syn$, $1 \leq i, j \leq n+k$ either (a) $i \in \{1, \ldots, n\}$ and $j \in \{n + 1, \ldots, k\}$ or (b) $i \in \{n + 1, \ldots, k\}$ and $j \in \{1, \ldots, n\}$;*
   - *syn is connected, as follows:*
     - *$indegree(\sigma_i) = 0$ and $outdegree(\sigma_i) > 0$, $i \in \{1, \ldots, l\}$;*
     - *$indegree(\sigma_i) > 0$ and $outdegree(\sigma_i) = 0$, $i \in \{l + q + 1, \ldots, n\}$;*
     - *$indegree(\sigma_i) > 0$ and $outdegree(\sigma_i) > 0$, $i \in \{l + 1, \ldots, n + k\}$;*

4. *$in = \{1, \ldots, l\}$ and $out = \{l + q + 1, \ldots, n\}$ are the sets of input and output proposition neurons respectively, with $in \cap out = \emptyset$;*

5. *In the initial configuration of Π, neurons σ_i $(1 \leq i \leq n + k)$ verify the following:*

   - *$p_i = 1$ for all $i \in \{1, \ldots, l\}$, otherwise $p_i = 0$, that is, any input proposition neuron has received exactly one pulse, while any other neuron has received no pulse;*
   - *$\alpha_i > 0$ for all $i \in \{1, \ldots, l\}$, otherwise $\alpha_i = 0$, that is, any input proposition neuron has a spike, while any other neuron has no spike;*

When a neuron σ_i receives spikes/pulses from its ingoing synapses, a three-stage process takes place: firstly, σ_i updates its content, secondly, σ_i checks if its rule r_i is applicable and thirdly, in case r_i is enabled, σ_i fires the rule. In what follows, we illustrate this process for a neuron i which state at instant t is $\sigma_i = (p_{i,t}, \alpha_{i,t}, \tau_{i,t})$. As such, content of σ_i is a spike $a^{p_{i,t}}$ of value, $a^{\alpha_{i,t}}$, with either (a) $p_{i,t} = 0, \alpha_{i,t} = 0$; or (b) $p_{i,t} > 0, \alpha_{i,t} > 0$. Also, $indegree(\sigma_i) = n_i \geq 1$, ingoing synapses of σ_i are labelled as $\{e_{i,1}, \ldots, e_{i,n_i}\}$ and at instant t neuron σ_i receives h pulses of values $a^{\alpha_{e'_{i,1}}}, \ldots, a^{\alpha_{e'_{i,h}}}$ from the ingoing synapses labelled as $\{e'_{i,1}, \ldots, e'_{i,h}\} \subseteq \{e_{i,1}, \ldots, e_{i,n_i}\}$ respectively, with $1 \leq h \leq n_i - p_{i,t}$. The process takes place as follows:

- *Update.* After updating, neuron state is $\sigma_i = (p_{i,t+1}, \alpha_{i,t+1}, \tau_{i,t+1})$, where:
  - $p_{i,t+1} = p_{i,t} + h$;
  - If $p_{i,t} = 0, \alpha_{i,t} = 0$ then:
    * $$\alpha_{i,t+1} = \begin{cases} max\{\alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is prop. neuron;} \\ max\{\alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is OR rule neuron;} \\ min\{\alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is AND rule neuron;} \end{cases}$$
  - If $p_{i,t} > 0, \alpha_{i,t} > 0$ then:
    * $$\alpha_{i,t+1} = \begin{cases} max\{\alpha_{i,t}, \alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is prop. neuron;} \\ max\{\alpha_{i,t}, \alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is OR rule neuron;} \\ min\{\alpha_{i,t}, \alpha_{e'_{i,1}}, \ldots, \alpha_{e'_{i,h}}\} & \text{if } \sigma_i \text{ is AND rule neuron;} \end{cases}$$
  - $$\tau_{i,t+1} = \begin{cases} \alpha_{i,t+1} & \text{if } \sigma_i \text{ is proposition neuron;} \\ \tau_{i,t} & \text{if } \sigma_i \text{ is rule neuron;} \end{cases}$$
- *Rule checking.* Neuron σ_i checks its contents to determine if rule r_i is enabled, as follows:

  $$enabled(r_i) \Leftrightarrow \begin{cases} p_{i,t+1} = 1, \alpha_{i,t+1} > 0 & \text{if } \sigma_i \text{ is input prop. neuron;} \\ p_{i,t+1} = n_i, \alpha_{i,t+1} > 0 & \text{if } \sigma_i \text{ is internal prop. neuron;} \\ p_{i,t+1} = n_i, \alpha_{i,t+1} > 0 & \text{if } \sigma_i \text{ is rule neuron;} \end{cases}$$

  Let us notice that output proposition neurons are never enabled, thus they cannot fire in any case.
- *Rule application.* Neuron σ_i fires r_i in case it is enabled. Firing involves consuming the whole pulse contained in σ_i and emitting a pulse $a^{\beta_{i,t+1}}$ that immediately reaches the neurons connected to the outgoing synapses

from σ_i. As such, after firing neuron i state is $\sigma_i = (0, 0, \tau_{i,t+1})$. On the other hand, truth value $\beta_{i,t+1}$ is defined as follows:

$$\beta_{i,t+1} = \begin{cases} \alpha_{i,t+1} & \text{if } \sigma_i \text{ is proposition neuron;} \\ \alpha_{i,t+1} * \tau_{i,t+1} & \text{if } \sigma_i \text{ is rule neuron;} \end{cases}$$

A configuration of the rFRSN P system is given by the state of its neurons at any time. The concepts of transition step and computation are defined in a similar way to classic SN P systems. The output of the system is given by the pulse values of output neurons when the computation halts.

5.4.1 Mapping fuzzy knowledge into rFRSN P systems

In order to use fuzzy production rules for fuzzy knowledge representation, it is necessary to map them into rFRSN P systems. In what follows, we summarize how to produce rFRSN P systems to model individual fuzzy production rules belonging to the types previously defined, where value τ_i of rule neuron i is assigned to the certainty factor of the associated fuzzy production rule.

A *simple fuzzy production rule* can be modelled by an rFRSN P system Π_1, as shown in Figure 5.1. The system is defined as follows:

$\Pi_1 = (O, \sigma_i, \sigma_j, \sigma_k, syn, in, out)$, where:

(1) $O = \{a\}$.

(2) σ_i is an OR rule neuron with confidence factor τ_i. Its spiking rule is of the form $a^\alpha \to a^\beta$, where $\beta = \alpha * \tau_i$.

(3) σ_j and σ_k are two proposition neurons associated with propositions p_j and p_k with truth values α_j and α_k respectively. Their spiking rules are of the form $a^\alpha \to a^\alpha$.

(4) $syn = \{(j, i), (i, k)\}$.

(5) $in = \{\sigma_j\}$, $out = \{\sigma_k\}$.

A *composite fuzzy conjunctive rule in the antecedent* can be modelled by an rFRSN P system Π_2, as shown in Figure 5.2. The system is defined as follows:

$\Pi_2 = (O, \sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}, syn, in, out)$, where:

(1) $O = \{a\}$.

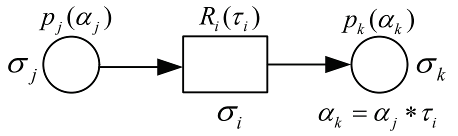


Figure 5.1: An rFRSN P system Π_1 for simple fuzzy production rules

(2) $\sigma_j (j = 1, 2, \ldots, k)$ are proposition neurons associated with propositions $p_j (j = 1, 2, \ldots, k)$ with truth values $\alpha_j (j = 1, 2, \ldots, k)$ respectively. Their spiking rules are of the form $a^\alpha \to a^\alpha$.

(3) σ_{k+1} is an AND rule neuron with confidence factor τ_i. Its spiking rule is of the form $a^{k-1}/a^\alpha \to a^\beta$, where $\beta = \alpha * \tau_i$.

(4) $syn = \{(1, k+1), (2, k+1), \ldots, (k-1, k+1), (k+1, k)\}$.

(5) $in = \{\sigma_1, \sigma_2, \ldots, \sigma_{k-1}\}$, $out = \{\sigma_k\}$.

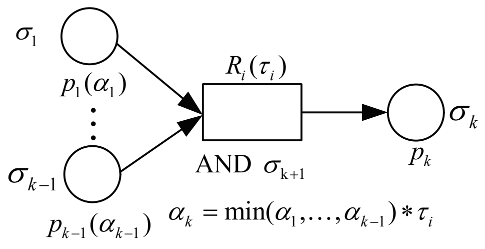


Figure 5.2: An rFRSN P system Π_2 for composite fuzzy conjunctive rules in the antecedent

A *composite fuzzy conjunctive rule in the consequent* can be modelled by an rFRSN P system Π_3, as shown in Figure 5.3. The system is defined as follows:

$\Pi_3 = (O, \sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}, syn, in, out)$, where:

(1) $O = \{a\}$.

(2) $\sigma_j (j = 1, 2, \ldots, k)$ are proposition neurons associated with propositions $p_j (j = 1, 2, \ldots, k)$ with truth values $\alpha_j (j = 1, 2, \ldots, k)$ respectively. Their spiking rules are of the form $a^\alpha \to a^\alpha$.

(3) σ_{k+1} is an OR rule neuron with confidence factor τ_i. Its spiking rule is of the form $a^\alpha \to a^\beta$, where $\beta = \alpha * \tau_i$.

(4) $syn = \{(1, k+1), (k+1, 2), (k+1, 3), \ldots, (k+1, k)\}$.

(5) $in = \{\sigma_1\}$, $out = \{\sigma_2, \sigma_3, \ldots, \sigma_k\}$.

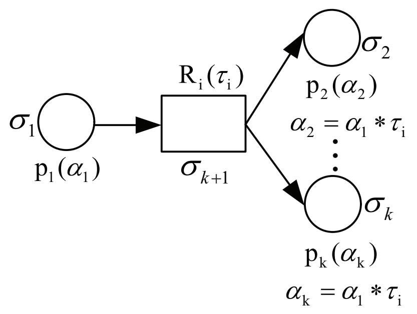


Figure 5.3: An rFRSN P system Π_3 for composite fuzzy conjunctive rules in the consequent

A *composite fuzzy disjunctive rule in the antecedent* can be modelled by an rFRSN P system Π_4, as shown in Figure 5.4. The system is defined as follows:

$\Pi_4 = (O, \sigma_1, \sigma_2, \ldots, \sigma_k, \sigma_{k+1}, syn, in, out)$, where:

(1) $O = \{a\}$.

(2) $\sigma_j (j = 1, 2, \ldots, k)$ are proposition neurons associated with propositions $p_j (j = 1, 2, \ldots, k)$ with truth values $\alpha_j (j = 1, 2, \ldots, k)$ respectively. Their spiking rules are of the form $a^\alpha \to a^\alpha$.

(3) σ_{k+1} is an OR rule neuron with confidence factor τ_i. Its spiking rule is of the form $a^{k-1}/a^\alpha \to a^\beta$, where $\beta = \alpha * \tau_i$.

(4) $syn = \{(1, k+1), (2, k+1), \ldots, (k-1, k+1), (k+1, k)\}$.

(5) $in = \{\sigma_1, \sigma_2, \ldots, \sigma_{k-1}\}$, $out = \{\sigma_k\}$.

5.5 Power system fault diagnosis modelling with rFRSN P systems

This section briefly describes a relevant application of rFRSN P systems within the area of fault diagnosis of power systems. For more detailed information, see [130].

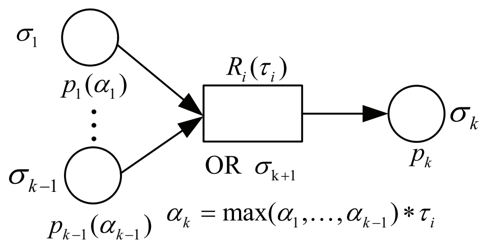


Figure 5.4: An rFRSN P system Π_4 for composite fuzzy disjunctive rules in the antecedent

The contents of this section is as follows. Firstly, some preliminary concepts and the associated problems about fault diagnosis will be introduced in Section 5.5.1. Secondly, some general principles conercing the modelling of fault diagnosis methods will be outlined in Section 5.5.2. Thirdly, the application of rFRSN P systems to fault diagnosis will be brifely described in Section 5.5.3.

5.5.1 Essentials of electric power system fault diagnosis

Fault diagnosis of a power system is a complex process involving many different sections such as generators, transmission lines, bus bars and transformers, protected by a protective system consisting of protective relays, circuit breakers (CBs) and communication equipments.

Fault diagnosis includes a number of processes, as fault detection, fault section identification, fault type estimation, failure isolation and recovery [204]. Among the five processes, fault section identification is especially important. When faults occur in a power system, protective relays detect the faults and trip their corresponding circuit breakers (CBs) to isolate faulty sections from the operation of this power system and guarantee the other parts can operate normally.

The protective system of an electric power system is very important in fault diagnosis, as well as the protection configuration of this protective system. The protective relays consist of main protective relays (MPRs), first backup protective relays (FBPRs) and second backup protective relays (SBPRs). It is worth pointing out that there is not any FBPR for buses.

5.5.2 Principles of fault diagnosis modelling

Fault diagnosis of power systems based on SN P systems belongs to the so–called model–based fault diagnosis methods. The framework of fault diagnosis in power systems using reasoning model-based method is described in detail in [206, 55].

There are three essential parts in this framework: real-time data, static data and a flowchart of identification fault sections.

1. *Real-time data.* The real-time data about the system, including protective relay operation and circuit breaker tripping information, is used to estimate the outage areas to obtain candidate faulty sections using a network topology analysis method, so as to reduce the subsequent computational burden [55].

2. *Static data.* The static data, as the network topology and the protection configuration of a power system, provide the starting point to build a fault diagnosis model for each candidate section. The inputs of each fault diagnosis model are initialized by both real-time data and static data.

3. *Flowchart to identify fault sections.* Each diagnosis model performs some reasoning algorithm to obtain fault confidence levels of candidate faulty sections to determine the ones actually failing. The diagnosis results include this faulty sections, along with their fault confidence levels.

In recent decades, fault diagnosis has been implemented by various approaches, such as expert systems, fuzzy logic, artificial neural networks, Petri nets, Bayesian networks, multi Agent systems, optimization methods, cause-effect networks or information theory, among others. Each method has its own merits and demerits.

To a certain extent, these methods deal with the uncertainty in failure processes with fast diagnosis speed, but it is difficult to dynamically describe the fault information needed. Therefore, much attention should be paid to the improvements of the aforementioned methods and the exploration of new ones to solve fault diagnosis problems.

5.5.3 Fault diagnosis with rFRSN P systems

As just mentioned, significant efforts should be made in finding relevant improvements on the methods to model fault diagnosis in electrical power systems, and new approaches to attack this kind of complex problems are more than welcomed.

In this sense, fuzzy reasoning spiking neural P systems with real numbers (rFRSN P systems) [130] have been proposed for solving fault diagnosis problems. To come up with this task, these systems make use of the so–called fuzzy reasoning algorithm (FRA), as detailed in Section 10.4. In [130] fault diagnosis in a transformer by rFRSN P systems is considered as a case study.

In order to use fuzzy production rules for this purpose, they need to be mapped into rFRSN P systems, as extensively described in Section 5.4.1. An application example might be used to test the feasibility and effectiveness of rFRSN P systems and their associated algorithm (FRA) in the fault diagnosis of a transformer. Thus, the following fuzzy production rules are obtained from knowledge base of a transformer fault diagnosis system.

6 Software for Membrane Computing

6.1 Introduction

As previously discussed, *Membrane Computing* is a branch of Natural Computing aiming to study computational devices, called membrane systems or *P systems*, as abstractions inspired from the functioning and structure of *living cells*. Contrary to other branches of Natural Computing, such as DNA Computing, current human technology is unable to implement P systems on their natural substrate, the biological cells. Hence, simulation tools, working on conventional electronic devices, are indispensable to further develop Membrane Computing discipline. In this way, simulators are crucial assistants when researching on P systems computational properties and their application to relevant real-life problems. At this respect, a key design goal is developing simulators being as efficient as possible, as well as meeting the semantics specifications of the simulated systems.

Software applications for Membrane Computing appeared right in the early times of the discipline. With an increasing number of P systems models being introduced, several simulators were developed to handle many of them. This resulted in the availability of a mix of coexisting tools, each one usually targeting at a single variant and with its own way to specify the associated models.

In 2009, `P-Lingua` framework is introduced [39], aiming to provide a unified software environment to define and simulate a wide variety of P systems models. P–Lingua consists of a general programming language for P systems called `P-Lingua` itself and a Java [195] based open source library called `pLinguaCore`. While P–Lingua language provides a common syntax for specifying P systems variants, pLinguaCore provides both parsers and simulators for such variants. The notable variety of supported models, both of the cell-like and tissue-like kind, contributed to make P–Lingua widely used among members of the Membrane Computing community, turning its specification language into *a sort of standard*.

In [133], `MeCoSim`, a visual environment for P systems is introduced, aiming to assist model designers and end users dealing with problems modelled after membrane systems. On the one hand, MeCoSim provides model designers with a graphic tool to design, simulate, analyse and verify their models. On the other hand, end users are provided with customizable applications, whose user interfaces are adapted for each problem, allowing them to enter the input data and check the results. MeCoSim *is built on top* of P–Lingua: models are specified in P–Lingua language and simulations are performed by executing simulation algorithms, most of which are provided by pLinguaCore library (also external simulators might be used).

With some remarkable exceptions (i.e. [36, 94]), it is worth pointing out that the vast majority of software simulation tools for P systems works on *sequential* platforms, using non-parallel-oriented programming languages such as Java, CLIPS, Prolog or C, where performance is compromised. Sequential P systems simulators performance is dramatically decreased as they serialize the natural double massively parallelism of P systems: execution of rules within each membrane, and the evolution of each membrane. Thus, the sequential simulation time proportionally increases as long as the quantity of parallelism presented in the P system. Although the last generation of commodity PCs is able to support the fast execution of sequential simulators, thus managing problem instances of enough size for current research lines [131], very large instance sizes are still unfeasible for them. Handling such instances requires shifting to *parallel* architectures. In this regard, *High Performance Computing (HPC)* is the field which studies the set of techniques needed to accelerate the execution of applications using parallel platforms. We can find these techniques in modern supercomputers and new parallel architectures (GPUs, FPGAs, CellBE, etc.).

6.2 Membrane Computing software overview

In this Section, an overview of software applications related to P systems is presented, in order to provide a context in which the work object of this dissertation is set. Including each existing piece of software would be an impossible task, and would go beyond the scope of this document. An exhaustive reference about P systems simulators developed from 2000 to 2010 can be found in [36], which serves as a basis for this overview. In addition, the reader interested on this is advised to check the software area of the P Systems Webpage `http://ppage.psystems.eu` to stay updated with the latest application releases.

The overview is structured as follows. Firstly, an introduction detailing P systems software applications purpose and architecture is presented. To continue, a reference on software applications is provided.

6.2.1 Purpose and architecture

According to [36], the *purpose* of P systems related software can be categorized in the following way:

- Pedagogical use.
- Assistant in the formal verification of P systems models.
- Solve real-life problems modelled after P systems.

The expected users and interoperability of Membrane Computing applications differ depending on the their purpose. Software intended to pedagogical use or to assist in formal verification tasks are usually targeted at P system designers, while applications solving real-life problems are used by experts in the domain of the processes under study. With respect to the interoperability, while in the first and second scenarios P systems software applications usually works stand-alone, in the last one they may act as modules of complex applications conforming problem solving suites.

Architecture of Membrane Computing related software is typically structured into three main modules: *input* (P system definition), *core* (simulation engine itself), and *output* (presentation of results). A discussion on the features of each module follows.

P system definition module

Simulating a P system requires providing the following information to the simulator: the kind of P system model to be simulated, the initial membrane structure and multisets of objects, the set of rules and, when simulating a P system family, the corresponding values for the initial parameters. Three alternatives exist when tackling the design of the P system definition module, depending on how such definition is provided to the simulator: included into the application source code, through a user interface or by means of external files.

Including the P system definition into the *source code* is a good choice when developing simulator prototypes, since it is the fastest alternative: the same programming language to both develop the simulator and specify the P system, saving time during the debugging and testing stages. Nevertheless, this alternative presents important drawbacks when used for final versions. Developing an application to simulate a single model is very uncommon and a more than probable change in the P system definition would require accessing and modifying the source code. This a difficult task to accomplish by end users who may not have access to the code or not be familiarised with the application programming language and its internal data structures, ultimately experiencing a strong dependence on the programmer. Moreover, even when the changes are made by the programmer, unnoticed bugs can be introduced.

Another option is to provide the P system definition through a *user interface*, which is commonly supported on visual elements that can be manipulated by the user, conforming a *Graphic User Interface* (or GUI for short). Designing a good GUI to specify the P system definition is a hard task in comparison to including it in the source code. It is a long and complex technical process that requires continuous interaction with the different kind of users susceptible to work with the application, in order to produce a friendly and flexible enough solution satisfying their diverse needs. Nevertheless, this effort results in an input module being highly independent from the rest of the application, thus enabling its modification without adverse side effects and making it relatively reusable, since it can be integrated in other products as long as they support its same programming language and software dependencies.

Finally, it is also possible to write the P system definition in *external files* that are processed or parsed by the input module of the simulator. In this way, the definition of the P system is completely independent from the application, being reusable for several software solutions that can share the same parsing specification logic. Hybrid designs can also be considered, in which GUIs are used to load/edit/store the definitions residing in external files, increasing the usability of the software. Alternatively, the edition of the files can be delegated to well-known third party applications.

Simulation core module

This module is tasked with simulating the defined P system (already parsed and checked by the input module), executing a simulation algorithm that captures its semantics and reproduces one or several computations. The simulation core can be designed with a sequential or parallel orientation, which will be ultimately constrained by the underlying hardware platform where it is executed. In this way, while single CPUs allow an inexistent or reduced parallelism level, specialized architectures such as clusters, GPGPUs and FPGAs allow a high level of real parallelism, enabling the execution of very efficient simulators. On the other hand, the simulation algorithm can be designed to (a) execute a single P system definition (typical in solutions where it is included in the source code); (b) simulate a kind of P system models; or (c) execute several kind of P system models sharing common semantics.

Presentation of results module

As previously stated, a P system simulation involves reproducing the configurations of one or more computations. This process generates a huge amount of information, relative to the membrane structure, object multisets, rules executed, etc. Nevertheless, this raw data is not always of interest for the user. For an application intended to either pedagogical uses or to assist in the formal verification of models, where the user is commonly a P systems designer, providing the full computation details is required. However, for those applications oriented to solve real-life problems, where the user is an expert on the phenomena under study and not necessarily familiar with the Membrane Computing paradigm, a filtering process to extract relevant information is indispensable.

6.2.2 Software reference

In what follows, a reference about developed software for Membrane Computing is presented, which is largely based in [36]. The evolution of P systems applications has taken place as the discipline itself has been developed, with a change in the orientation of the tools, as mentioned above. Two software generations (which are overlapped in time) can be identified. First generation corresponds to software intended to pedagogical/teaching tasks related with

P systems and to assist in the design and formal verification of complex P systems, saving researchers from heavy hand-done calculations. As a result of this, first generation tools provide full information related to computations, thus requiring a large amount of resources to store/deal with that volume of data. Consequently, they can only support small instance problems. In the second generation, the focus shifts from studying P systems themselves to apply them to model and study real-world problems, specially in the field of biology. In this generation, only relevant data for the user is handled by the applications since getting solutions as efficiently as possible becomes one of the main goals.

The first generation

Some well-known tools falling into the first generation of P system related software are the following (a complete reference is listed in [36]):

- *Malița Simulator (2000).* Introduced in [91], it was the first developed simulator. Intended for simulating transition P systems without membrane division, this early simulator applied a restricted parallelism: in each step, for each membrane, only one rule could be selected and subsequently applied as many times as possible.

- *Suzuki and Tanaka Simulator (2000).* Introduced in [169], it was intended for simulating transition P systems without membrane division, with an important constraint: size of multisets was bounded. This simulator aimed to tackle real-life problems, being successfully used to simulate the Brusselator model as well as ecological systems [168].

- *Balbotin et al. Simulator (2002).* Introduced in [11], it was intended for simulating transition P systems. This simulator outputs the computation tree step by step until the desired number of evolution steps is reached. Consequently, when the branching rate of the computation tree is large, only a few steps can be simulated.

- *Baranda Simulator (2002).* Introduced in [9], it was intended for simulating transition P systems based on a theoretical formalisation proposed by members from the Natural Computing Group of the Technical University of Madrid [188].

- *Ciobanu and Paraschiv Simulator (2002).* Introduced in [27], it was intended for simulating catalytic hierarchical cell systems and P systems with active membranes. It allowed interactive definition and visualization of membrane systems and a graphical representation of the computation and final result.

- *Sevilla Team Simulators (2004).* Two simulators were developed by the Sevilla Team [191] intended to assist in tasks related with the design and formal verification of cellular solutions to **NP**–complete problems. The first one was written in CLIPS and introduced in [134], while the second one was implemented in Prolog and introduced in [31]. These simulators were intended for assisting in the design and verification of cellular solutions to **NP**–complete problems, with only confluent P systems being considered.

The second generation

Some well-knows tools falling into the second generation of P system related software are the following (a complete reference is listed in [36]):

- *Ardelean and Cavaliere Simulator (2003).* Introduced in [8], it was intended for simulating a special variant of transition P systems, allowing rewriting and symport/antiport rules. In this simulator, available rules are applied in the maximally parallel mode by using the *weak priority approach.* This software has been used to simulate several important biological processes [19].

- *Verona Team Simulator (2006).* Psim simulator was introduced in [12], intended for simulating metabolic P systems.

- *Romero-Campero et al. Simulator (2006).* Sevilla and Sheffield Teams presented two software tools for simulating biological processes with P systems by implementing the multi-compartmental Gillespie algorithm. They have allowed addressing several real-life problems such as simulation of pathways associated to the Epidermal Growth Factor Receptor (EGFR) [141], simulation of FAS–induced apoptosis [26], modelling of gene expression control [161], or a first computational model of Quorum Sensing [160] in *Vibrio Fischeri.*

- *Sedwards and Mazza Simulator (2007.)* The Cyto-Sim simulator was introduced in [163], intended for simulating stochastic processes at micro and macro level.

- *Cazzaniga and Pescini Simulators (2006).* One of them simulates the gene regulation system of the bacterium *Vibrio Fischeri* using the multi-compartmental Gillespie algorithm.

- *Nishida Simulator (2006).* This simulator implements the "membrane algorithm" presented by Nishida in [113] for **NP**–complete optimization problems.

Finally, two recent developments belonging to the second generation and not included in [36] are:

- *Meta P-lab (2009).* Introduced in [18], this software consists in a virtual laboratory aiming to assist designers in both understanding the internal mechanisms of biological systems and to forecast, in silico, their response to external stimuli, environmental condition alterations or structural changes.

- *Infobiotics Workbench (2011).* Introduced in [14], it provides an integrated software suite incorporating model specification, simulation, parameter optimization and model checking for Systems and Synthetic Biology.

6.3 P–Lingua framework

P–Lingua framework was introduced in [39] providing a general programming language for P systems, called *P–Lingua* itself, and a Java [195] based open source library called *pLinguaCore*. On the one hand, P–Lingua language provides a common syntax for specifying P systems of the cell-like, tissue-like and, as a result of the work discussed in this dissertation, neural-like kind. On the other hand, pLinguaCore library provides both parsers and simulators for variants supported in P–Lingua language. The notable variety of supported models has made P–Lingua a widely used software among members of the Membrane Computing community, turning its specification language into *a sort of standard.*

6.3.1 P–Lingua language

P–Lingua is the P systems specification language of P–Lingua framework. Its main features are the following:

- Specification provided in plain *text files*. In this way, the specification is not coupled to any application, favouring interoperability.

- Close to real mathematical P systems specification language. As a result of this, the learning curve for those familiar with Membrane Computing paradigm is short.

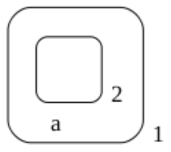
- *Parametrisation* is allowed. Thus, P system families can be specified.

- Specification is organized into *modules*. This favours reusability, since a piece of the specification can be used several times in the same or different P–Lingua programs.

- *Common syntax* to specify several P systems variants. The language uses the same directives to specify membrane structures, initial multisets, etc. independently of the kind of the model defined.

- Easily *extensible*.

Fig. 6.1 shows a small specification in P–Lingua language, corresponding to the definition of a simple P system with active membranes and membrane division rules.



$$[a \longrightarrow a, b]_1$$
$$b[\ ]_2 \longrightarrow [c]_2^+$$
$$[c]_2^+ \longrightarrow [d]_2\ [e]_2^-$$

```
1: @model<membrane_division>
2: def main()
3: {
4: @mu = [[]'2]'1;
5: @ms(1) = a;
6: [a --> a,b]'1;
7: b[]'2 --> +[c]'2;
8: +[c]'2 --> [d]'2 -[e]'2;
9: }
```

Figure 6.1: A simple P system specification in P–Lingua

In the P–Lingua definition (right), the following elements are specified: P system variant (line 1), mandatory main module (line 2), initial membrane structure (line 4), initial multisets (line 5), rules (lines 6-9).

6.3.2 pLinguaCore library

pLinguaCore is a software library belonging to P–Lingua framework, providing both parsers and simulators for supported variants in P–Lingua language, plus other utilities. Its main features are the following:

- Implemented in Java[1] [195]. Consequently, it is a multi-platform software that can integrated with other Java based applications such as MeCoSim.
- Free software, released under GNU GPL license [194].
- Supports P system definitions in several input formats, that can be either simulated or converted to other output formats.
- Multiple simulation algorithms provided. Each variant supported comes with one or more simulation algorithms.
- Text interface.

pLinguaCore supports specifying P systems in the following file formats:

- *P–Lingua* language format. This is the "native" format in P-Lingua framework enabling the specification of any supported variant. A references list for P–Lingua format syntax is provided below.
- *XML* format. This format is intended for interoperability of the framework with other general-purpose systems. Only cell-like variants can be defined in XML format. Parametrisation is not supported.
- *Binary* format. This format is intended for interoperability of the framework with CUDA simulators. As these simulators are expected to work with really huge systems, a compact way to specify them becomes necessary. At present, P systems with active membranes are supported, with support for PDP systems to be incorporated in the next framework version.

The previously described formats can be categorized into input or output formats. *Input formats* provide P systems specifications to be either parsed and simulated or compiled to a different *output format* by the corresponding simulation and compilation tools included in pLinguaCore. These tools are described in [45]. Supported *input formats* are P–Lingua and XML, while *output formats* are XML and binary.

[1] current version 5.0 of the library includes some kernel CUDA code that is compiled on the fly along simulation.

6.3.3 Currently supported P systems variants

Current version of P–Lingua framework is 5.0, which was released coinciding with the publishing of this dissertation. A list of supported P systems variants follows (full details about these variants can be found in previous Chapters).

- *Cell-like systems.* Introduced in [148], they are inspired from the hierarchical structure of eukaryotic cells. The following variants are supported:
  - *Transition P systems.* The basic P systems were introduced in [148], allowing definition of priority-based rules.
  - *Symport/antiport P systems.* Introduced in [145], allowing only communication rules of the symport or antiport kind.
  - *Active membranes with division rules.* Introduced in [149], allowing membrane multiplication by means of division of membranes. Support for object-fired membrane division rules of both elementary and non-elementary membranes is included.
  - *Active membranes with creation rules.* Considered for the first time in [69] and [106], allowing membrane multiplication by means of creation of membranes.
  - *Probabilistic P systems.* Population Dynamics P systems, were introduced with this specific denomination in [29]. This variant takes its name from its original application to model phenomena related to real-life ecosystems. It has also proved successful to deal with other phenomena such as gene networks.
- *Tissue-like systems.* Introduced in [92, 93], they take inspiration from the way in which cells organize and communicate within a net-like structure in tissues. The following variants are supported:
  - *Tissue P systems with communication and division rules.* Introduced in [154], allowing only communication of objects among cells (or the environment) and multiplication of cells through cell division.
  - *Tissue P systems with communication and separation rules.* Introduced in [121], allowing only communication of objects among cells (or the environment) and multiplication of cells through cell separation.

- *Neural-like systems.* Introduced in [67]. Collectively called Spiking Neural P systems (SN P systems, for short), they are inspired from the way in which neurons in the brain exchange information by means of the propagation of spikes. The following variants are supported:

  - *SN P systems with neuron division and budding rules.* Introduced in [119], allowing neuron multiplication by means of division and budding rules.
  - *SN P systems with functional astrocytes.* Introduced in [86], allowing astrocytes changing the synapse spike traffic according to certain functions.
  - *SN P systems with hybrid astrocytes.* Introduced in [123], allowing astrocytes permitting (excitatory behaviour) or forbidding (inhibitory behaviour) synapse spike traffic according to a certain threshold.
  - *SN P systems with anti-spikes.* Introduced in [122], allowing anti-spikes objects which take inspiration by the behaviour of matter vs. anti-matter.

- *Fuzzy Reasoning SN P systems.* Introduced in [130], they incorporate fuzzy diagnosis knowledge and reasoning elements into SN P systems, in order to model fault diagnosis and other real-life problems. The following variants are supported:

  - *Fuzzy Reasoning SN P systems with real numbers.* These systems were introduced in [130].

- *Kernel P systems.* Introduced in [50, 51], they aim to produce a single model incorporating features from many other P systems variants, in order to provide a unified modelling framework suitable to handle a wide range of scenarios. The following variants are supported:

  - *Simple Kernel P systems.* Introduced in [52]. Kernel P Systems define a rather complex model; they encompass a wide range of features from a variety of P system models. With the aim of providing, a simpler exploratory first step towards this direction was introduced, through the formalization of a subset of Kernel P systems known as simple Kernel P systems (skP systems), including several features and limiting the execution strategies of the former ones.

- *Probabilistic Guarded P systems.* Introduced in [43]. They comprise a computational probabilistic framework which takes inspiration from different Membrane Computing paradigms, mainly from Tissue-like P systems, PDP systems and Kernel P systems. This variant proposes a modelling framework for ecology in which inconsistency is handled by the framework itself, instead of delegating to simulation algorithms.

- *Numerical P systems.* Introduced in [155]. A non-deterministic, parallel model originally aimed to model the underlying uncertainty of economical processes. In these P systems, the traditional multisets of objects associated with membranes are replaced by sets of *numerical variables.* These variables evolve by means of *programs* associated with the membranes. The following variants are supported:

  – *Enzymatic Numerical P systems.* Introduced in [129]. Contrary to Numerical P systems, this variant describes a deterministic model of computation. Thus, instead of non-deterministically chosen, the programs to be applied are controlled by specific variables known as *enzyme-like variables.*

  Both Numerical and Enzymatic Numerical P systems have been successfully applied for modelling robot controllers.

- *Simple Regenerative P systems.* Introduced in [46]. They aim to model regenerative processes by means of P systems.

6.3.4 P–Lingua framework version history

P–Lingua has evolved trough its different versions, each one adding new supported models and implementing new simulation algorithms. Details on P–Lingua framework (language syntax and library features) are spread over several conference and journal papers as well as Ph.D. theses, that can be found at [199]. The central Ph.D. thesis on the framework is [131]. Next, a chronological enumeration of the different versions of P-Lingua framework is presented, including the features of each version and related papers.

1. **P-Lingua 1.0** [38]: only active membranes P systems with cell division are supported.

2. **P-Lingua 2.0** [44, 45]: support for several cell-like P system models is added:

- Transition P systems.
- Symport/antiport P systems.
- Active membranes with division rules.
- Active membranes with creation rules.
- Stochastic P systems.
- Probabilistic P systems.

3. **P-Lingua 2.1** [100]: support for tissue-like P systems with division rules is added.

4. **P-Lingua 3.0** [97]: support for Population Dynamics P systems (PDP Systems) is added, and several built-in simulators for this model are included. Support of stochastic P systems is discontinued, in favour of Infobiotics Workbench [14].

5. **P-Lingua 4.0** [84, 85, 132]: support for tissue-like P systems separation rules is added. Also, support for SN P systems is added, considering the following elements[2]: *neuron budding and division rules, functional astrocytes, hybrid astrocytes and anti-spikes.*

6. **P-Lingua 5.0**: Coinciding with the publishing of this dissertation, version 5.0 is released, adding support for the following variants:

   - Cell-like systems with symport/antiport rules and *arbitrary* number of copies of initial environment objects[2] [90].
   - Limited Asynchronous SN P systems and Asynchronous SN P systems with Local Synchronization[2] [88].
   - Fuzzy Reasoning SN P systems with real numbers[2] [130].
   - Simple Kernel P systems [51, 52].
   - Probabilistic Guarded P systems [43].
   - Enzymatic Numerical P systems [48, 47].
   - Simple Regenerative P systems [46].

[2] support is added as a result of the work related to the object of this dissertation.

6.4 MeCoSim (Membrane Computing Simulator)

MeCoSim is a visual environment oriented to assist model designers and end users who deal with problems modelled after P systems. While model designers are interested in solving abstract problems by defining the corresponding computational models, end users are interested in solving concrete instances and may not be familiar with the Membrane Computing paradigm. On the one hand, MeCoSim provides model designers with a graphic tool to design, simulate, analyse and verify their models. On the other hand, end users are provided with customizable applications, whose user interfaces are adapted for each problem, allowing them to enter the input data and check the results. MeCoSim is built on top of P–Lingua: models are specified in P–Lingua language and simulations are performed by executing simulation algorithms provided by pLinguaCore library.

MeCoSim features can be reviewed in detail in [173]. Some of its main features are the following:

- P–Lingua as its main *inference engine*. The MeCoSim GUI allows loading, parsing and debugging models written in P–Lingua format. Besides, available simulation algorithms can be selected for the corresponding model variant.

- Extended support for *parametrised models*. MeCoSim enables handling models whose parameters instantiation is not hardcoded in the model file. To accomplish this, MeCoSim allows the definition of a custom interface for each model. In this way, the user can enter the input data for the model. Similarly, MeCoSim provides a *parameter generation language* that can be used to specify how parameters are instantiated from the input data.

- *Customizable output*. MeCoSim provides a mechanism to define which output data should be computed from simulation results and how that information should be displayed to the user. In this way, complex outputs can be obtained through a post-processing that includes filtering and grouping techniques. The output data can be displayed in tables and charts.

- *Plugins* architecture. MeCoSim supports extending its core functionalities through plugins. In this way, MeCoSim can make calls to Java-coded libraries or external programs.

- Invariant extraction support. A plugin allows MeCoSim to interoperate with *Daikon* [193] invariant detector. A trace of the model/solution computation is sent to Daikon and the result of the invariant detection is shown.

- *Formal verification* support. Another plugin allows MeCoSim to interoperate with *Spin* [201] model checker. A *Promela* model checker language file is generated from the model and scenario loaded that, after edition to include the properties to verify, is sent to Spin.

- *Repositories* management. MeCoSim provides a repository system able to handle public online resources, such as plugins, custom applications, models and scenarios.

6.5 Parallel simulation of P systems

Most Membrane Computing simulators have been exclusively implemented on sequential architectures, which constrain the theoretical maximum performance that could be obtained by fully exploiting the parallel nature of P systems. Consequently, simulators working on high performance architectures come into scene. In what follows, we review these architectures and the corresponding developed P systems applications. Contents of this Section follows from [43].

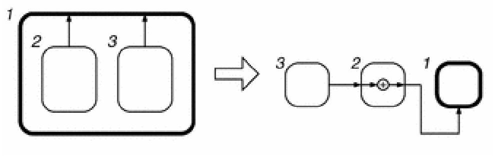
6.5.1 FPGA boards

A Field Programmable Gate Array (FPGA) circuit [162, 171, 174] is an array of (a usually large number of) logic cells placed in a highly configurable infrastructure of connections. Each logic cell, also known as Control Logic Block (*CLB*) can be programmed to realize a certain function [171].

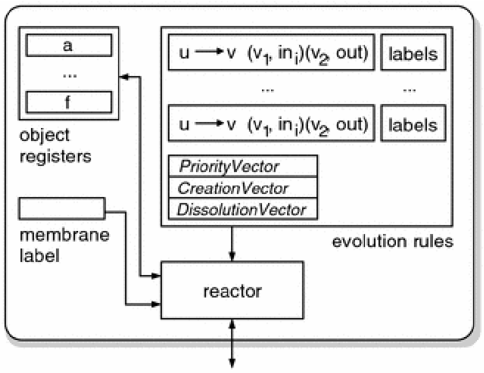
The seminal work on parallel simulation of P systems on FPGA boards is a Transition P system simulator by Petreska and Teuscher [142], working as follows. Each membrane is a construct composed of an 8–bit register per object in the alphabet, an 8–bit register to store the membrane label and one bit *status flag* to indicate if the membrane is *enabled* (1) or *disabled* (0). Membranes are connected by means of bidirectional buses. Instead of individually connecting membranes to their children, each non–elementary membrane is connected to one child, which is in turn connected to one of its siblings and so on, in a linked list manner.

All rules follow a common pattern, which is $u \to v(v_1, in_i), (v_2, out)$, where i is the label of a children membrane and u, v, v_1, v_2 are multisets over the alphabet. Upon the application of a rule, multiset u evolves into v, multiset v_1 is sent to membrane i and multiset v_2 is sent to the parent membrane. Rules are applied according to a priority list in a strong sense, which indicates the order in which rules must be applied. Hence, their implementation is non–deterministic as long as this priority list is randomly generated on each simulation.

Rule registers in a membrane are connected to a circuit known as *reactor*. In addition, each membrane integrates three arrays of 8–bit registers: *UpdateBuffer*, *FromUpperBuffer* and *ToUpperBuffer*, each one with as many positions as objects are in the alphabet. At every step, each applicable rule issues an *applicable* signal. For every rule which has issued signal *applicable*, the reaction circuit subtracts cardinalities in u from w and adds objects in v to buffer *UpdateBuffer*, objects in v_1 to buffer *FromUpperBuffer* in membrane i and objects in v_2 to buffer *ToUpperBuffer*. If the rule creates a membrane, a disabled membrane is enabled, copying all information into the membrane registers and setting its status flag to enabled. Finally, the membrane label is set according to rule's created membrane label. When all applicable rules are applied, each membrane adds objects in buffer *ToUpperBuffer* to buffer *UpdateBuffer* of its parent membrane. Next, each membrane adds objects in *UpdateBuffer* and *FromUpperBuffer* into w. Finally, if the rule dissolves the membrane, its status flag is set to disabled and all its objects are sent to its parent membrane multiset registers. This structure is shown in Fig. 6.2.



Membrane bus wiring



Structure of a membrane circuitry

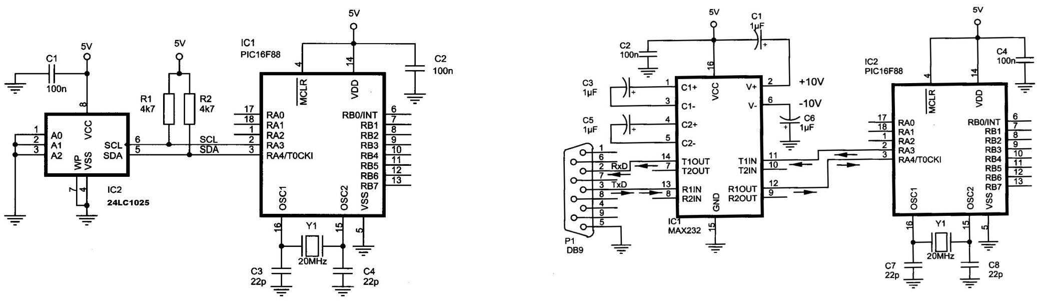
Figure 6.2: Overview of Petreska and Teuscher's FPGA Membrane Computing simulator

Another interesting work on the FPGA simulation of P systems is authored by Nguyen *et al.* [110]. In their work, they present Reconfig–P, a Java application which generates a FPGA circuit description out of Handel–C code. Handel–C [82] is an automated synthesis language based on ANSI–C for defining reconfigurable hardware at a high level of abstraction. Reconfig–P integrates P Builder, an application which takes into account the variety of specificities of the input P system and generates a circuit description accordingly. Reconfig–P allows the designer to define plenty of fine–grained details about the system to simulate, such as resource allocation approaches (object–oriented and rule–oriented) [111] and conflict resolution strategies to resolve object competition (time–oriented and space–oriented) [107, 108, 109, 111]. Reconfig–P supports membrane division and dissolution, by allocating circuitry for new membranes and freeing components from dissolved membranes. In addition, Reconfig–P and P Builder are designed for extensibility, in such a way that new P systems and features can be easily incorporated.

6.5.2 Microcontrollers

Microcontrollers have been also used to simulate P systems. Gutiérrez *et al.* [56] proposed a network of these devices. The computational workload is assigned to microcontroller PIC16F88. These devices are low-frequency computers working at 20Mhz, 8 bits of bus width and 8 bits of word size. Their relatively low cost ($1.90) makes them appropriate to assemble a massive network, in which each one of them simulates a different membrane. Their main drawback is its scarcity of memory, which implies that a different model of microcontroller needs to be used to store data. In their solution, the authors suggest devices of type 24LC1025 for memory storage. Due to their Harvard architecture [164, 10], these devices are appropriate to hold different kind of data: they contain a non–volatile memory (128 Kbytes) and a volatile memory. In addition, they can work on fast mode (at 400Khz) and on slow mode (at 100Khz). However, although their embedded memory suffices to store local data, external modules are required to store global variables.

To implement a general clock which synchronizes the whole system, microcontrollers are connected to a Personal Computer (PC) which sets the current execution time. The whole architecture is interconnected by a I2C, which defines a synchronous, bidirectional protocol ensuring that information concerning modification in the cardinality of objects due to the application of rules is properly transmitted. They estimated the whole cost of a system composed of 1000 membranes at about $10000, a much more affordable solution than cluster–based platforms. Figure 6.3 displays the architecture of the simulator.



General structure **Monitoring and clock systems**

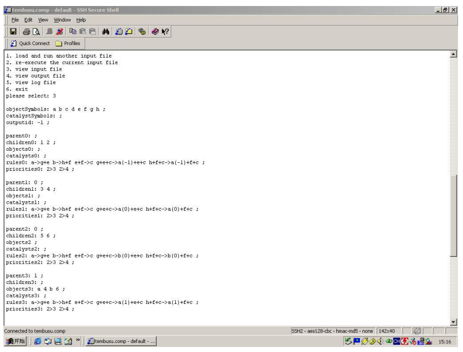
Figure 6.3: General structure (left) and monitoring system (right) of a Membrane Computing simulator based on microcontroller technology

6.5.3 Computer clusters

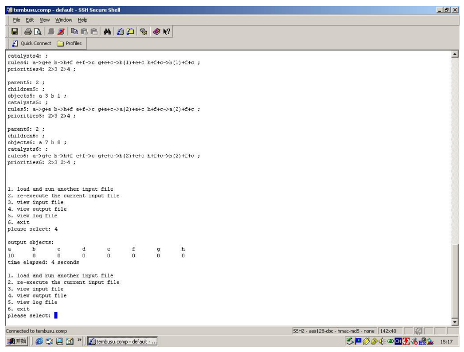
Distributed simulation on computer clusters adds up to the parallel approaches considered to simulate P systems.

In their work, Ciobanu and Guo [28] present a simulator for P systems on C++ which runs on a computer grid. Workload distribution was achieved by using Message Passing Interface (*MPI*) [117]. MPI is a popular middleware in which chunks of data are transmitted among distributed nodes in a cluster by means of function calls. These calls rely on low–level communication structures such as sockets, semaphores, stubs and message buffers. The authors focus on transition P systems, which do not implement membrane division. The P system to simulate is specified on an input file. An output file is produced containing the current configuration upon the halting of the simulator. In the implementation, simulation of each membrane is allocated to a different node in the grid. In each node, each rule is assigned to a different thread, in such a way that rules are applied concurrently. Issuance of objects is achieved by relying on MPI messages among nodes. When a node detects that there are no more applicable rules at a time step, it sends a message to a central node playing the role of the skin membrane. If, on a transition step, the skin membrane receives such messages from all nodes in the grid, then it broadcasts a halting signal to all nodes and the simulation halts. Rule priority is also implemented on this simulator; prior to the application of a rule, its thread checks that there are not applicable rules with higher priority. When object competition among rules takes place, objects are assigned among competing rules at random, therefore implementing non–determinism. The simulator's interaction consoles are displayed on Figure 6.4.

There are also some works on the distributed simulation of P systems with *Hadoop* [41], a popular framework for Parallel Computing based on the *Map–Reduce* pattern [178]. This framework applies in parallel an operation *Map* to each data and unifies the results of these operations by applying operation *Reduce*.


tembusu.comp - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
Connected to tembusu.comp
SSH2 - aes128-cbc - hmac-md5 - none 142x40
15:16

```
1. load and run another input file
2. re-execute the current input file
3. view input file
4. view output file
5. view log file
6. exit
please select: 3

objectSymbols: a b c d e f g h ;
catalystSymbols: ;
outputid: -1 ;

parent0: ;
children0: 1 2 ;
objects0: ;
catalysts0: ;
rules0: a->g+e b->h+f e+f->c g+e+c->a(-1)+e+c h+f+c->a(-1)+f+c ;
priorities0: 2>3 2>4 ;

parent1: 0 ;
children1: 3 4 ;
objects1: ;
catalysts1: ;
rules1: a->g+e b->h+f e+f->c g+e+c->a(0)+e+c h+f+c->a(0)+f+c ;
priorities1: 2>3 2>4 ;

parent2: 0 ;
children2: 5 6 ;
objects2 ;
catalysts2: ;
rules2: a->g+e b->h+f e+f->c g+e+c->b(0)+e+c h+f+c->b(0)+f+c ;
priorities2: 2>3 2>4 ;

parent3: 1 ;
children3: ;
objects3: a 4 b 6 ;
catalysts3: ;
rules3: a->g+e b->h+f e+f->c g+e+c->a(1)+e+c h+f+c->a(1)+f+c ;
priorities3: 2>3 2>4 ;
```


tembusu.comp - default - SSH Secure Shell
File Edit View Window Help
Quick Connect Profiles
Connected to tembusu.comp
SSH2 - aes128-cbc - hmac-md5 - none 142x40
15:17

```
catalysts4: ;
rules4: a->g+e b->h+f e+f->c g+e+c->b(1)+e+c h+f+c->b(1)+f+c ;
priorities4: 2>3 2>4 ;

parent5: 2 ;
children5: ;
objects5: a 3 b 1 ;
catalysts5: ;
rules5: a->g+e b->h+f e+f->c g+e+c->a(2)+e+c h+f+c->a(2)+f+c ;
priorities5: 2>3 2>4 ;

parent6: 2 ;
children6: ;
objects6: a 7 b 8 ;
catalysts6: ;
rules6: a->g+e b->h+f e+f->c g+e+c->b(2)+e+c h+f+c->b(2)+f+c ;
priorities6: 2>3 2>4 ;



1. load and run another input file
2. re-execute the current input file
3. view input file
4. view output file
5. view log file
6. exit
please select: 4

output objects:
a       b       c       d       e       f       g       h
10      0       0       0       0       0       0       0
time elapsed: 4 seconds

1. load and run another input file
2. re-execute the current input file
3. view input file
4. view output file
5. view log file
6. exit
please select:
```

Figure 6.4: Input (left) and output (right) from Ciobanu and Guo's cluster simulator

6.6 GPU Computing

In what follows, simulation of P systems by means of Graphic Processing Units (GPUs) is discussed. Contents of this Section follows from [43], with updated information included were appropriate. GPUs were originally devised as auxiliary computers to assist the main Central Processing Unit (CPU) in carrying out graphical computations, but it became evident that the parallel architecture of GPUs could be successfully applied for other parallel applications. Within a GPU-enabled device, graphical data is streamlined into the GPU memory by using Direct Memory Address (DMA) [42] circuitry, thus releasing the CPU from graphical computation. As new data chunks arrived at GPU memory, they were dynamically allocated to idle graphic processors. This workflow configures a *Parallel Computing* scheme: provided a certain degree of independence between chunk processing, computations carried out at different processors do not (at least heavily) depend on each other. This property is called *Data Parallelism*. GPUs integrate auxiliary hardware to perform common graphical tasks, such as raytracing [158], antialiasing [103, 34] and pixel shading [33]. In reference to general purpose application of GPU technology, Mark Harris, engineer at NVIDIA Corp., coined the term General–Purpose GPU (GPGPU) Computing in 2002 [116]. Nevertheless, the lack of appropriate development frameworks to implement general–purpose parallel algorithms on GPUs prevented a big development hit in this application field: general–purpose algorithms had to be translated to graphical structures, i.e., numerical values had to be mapped to pixels and colour levels so that GPUs could process them [116].

6.6.1 CUDA programming model

The appearance of GPGPU Computing software development kits (*SDKs*) removed the previous translation requirement. In 2007 NVIDIA announced CUDA (*Compute Unified Device Architecture*), a programming model specifically designed for GPGPU Computing. CUDA defines an abstraction of a GPU known as *grid*, which mirrors the memory hierarchy and processor distribution in commercial graphic cards. This abstraction allows the developer to allocate resources and tasks among processors and memory segments without depending on any specific device. Hence, from the developer's perspective, what runs his program is a parallel architecture of processors, in which information is expressed in terms of standard data structures.

An NVIDIA GPU is composed of a set of cores or *Streaming Processors* (SPs), whose number varies with the graphic cards. SPs are arranged in *Streaming Multiprocessors* (SMs). Each SP has access to a set of extremely low latency registers and to a section of low latency *shared memory* along with the other processors in the SM. All SPs, independently of their SM, have access to a common region of high latency global memory, which reaches 4 GB in modern Tesla cards [198].

In CUDA programming model [74, 116, 112], threads are arranged in *blocks*. CUDA implements synchronization directives at a level of blocks and at a level of the device as a whole. Threads in the same block have access to a common, low–latency *shared memory* hidden from other blocks. Threads in a block can be easily synchronized with barrier–like directives. In addition, each processor integrates a set of almost immediate access registers and a quick–access local memory. Moreover, all processor have access to a global, high–latency memory to store massive amounts of data. Finally, all threads have access to large, read–only memory units known as *Constant* and *Texture* data. This model is represented in Figure 6.5.

On runtime, CUDA dynamically assigns bundles of threads or *warps* to idle SMs. Each one of these threads executes the same code on different,

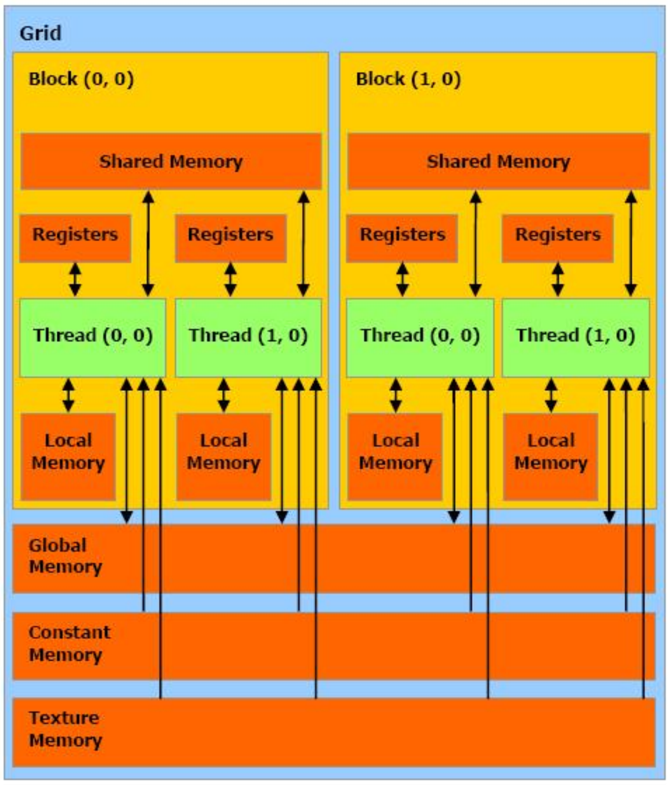


Figure 6.5: The CUDA programming model

thread–dependent data. This Parallel Computing paradigm is known as Single Instruction Multiple Data (*SIMD*). It is worth pointing out that the output of the program should not depend on the order in which these warps are computed. Otherwise, correct execution is not guaranteed [112]. Warps are the smallest units of parallelism; if a warp is broken, then its whole execution is performed sequentially.

CUDA/C++ is a programming language based on C/C++ which implements the CUDA programming model. A CUDA/C++ program is divided into two main parts: the *host* part and the *device* part. The host is the part of the code to be run on the CPU, whilst the device is the part to be executed on the GPU [22]. The host part includes calls to functions belonging to the device part or *kernels*. The device part can be composed by one or more kernels that are suitable for execution on the GPU. A kernel executes a scalar sequential program on a set of parallel threads. The programmer organizes these threads in two ways, showing the two levels of parallelism inside the kernel (threads and thread blocks). In this way, both parts of the program can cooperate in order to obtain a global result [112].

6.6.2 OpenCL

Another major standard in GPGPU Computing is OpenCL, a programming language supported by a consortium of enterprises which seeks out interoperability between parallel technologies, not only limiting to GPU devices but also including other platforms such as FPGA boards. Unlike CUDA, OpenCL is a free standard, independent from any particular company [170, 104, 189]. OpenCL is also compatible with AMD devices, such as AMD Fusion cards or Accelerated Processing Unit (APU) [186], which integrate GPU and CPU features so as to achieve a higher performance than each one of its parts separately. OpenCL is also compatible with Intel graphic cards and heterogeneous architectures composed of a CPU and external computing peripherals, not only GPUs but also FPGA boards and microcontroller networks. OpenCL defines a programming model similar to CUDA, but with its own specificities. More information about OpenCL can be found at [189].

6.6.3 GPGPU simulation in Membrane Computing

A variety of P system models have been simulated by applying GPGPU paradigm. The parallel architecture of GPUs, along with its relatively easy programming with GPGPU tools, accounts for its suitability as an appropriate simulation platform for P systems. In what follows, we discuss GPGPU based simulators developed for different P systems variants.

6.6.3.1 P systems with Active Membranes

The first Membrane Computing application on GPU technology was a simulator for P systems with active membranes developed by Cecilia *et al.* [22]. In their implementation, each step on a simulation consists on two stages: a *selection* stage and an *execution* one. Moreover, a simulator specifically developed for a model of P systems with Active Membranes solving the SAT problem is presented in [23, 21]. This simulator is limited to P systems with two membrane levels: one for the skin membrane and other for its inner membranes. That is, inner membranes must be elementary membranes. These restrictions meet in the case of the SAT–solver P system introduced in their case study. Only one computation is simulated, which is enough due to the fact that the simulated P systems in the family are *recognizer P systems* solving a decision problem, and that family is sound and complete, hence it is *confluent*. In their work, the authors reported execution times up to 63 times faster than the sequential counterpart. The experiments were undertaken on a Linux server with a NVIDIA Tesla C1060 graphic card at 1.3 Ghz with 240 processors installed.

6.6.3.2 Spiking Neural P systems

The success obtained by Cecilia *et al.* encouraged Cabarle *et al.* [15, 16] to develop a simulator for SN P systems on the same technology. The features of these systems are challenging for implementing a GPGPU simulator. Rules application depends on regular expressions, while rules might define a time delay which states the number of step cycles after the rule issues spikes to its neighbours. Due to the inherent difficulty in simulating the model, systems without delays were first targeted. The implemented simulator is based on a matrix representation of SN P systems [210], in which rule applications are mapped into matrix operations. Taking advantage of the suitability of GPUs for algebraic computing, these operations are accelerated on the GPU. Their implementation sequentially matches the content of each neuron against each rule's expression. Then, it generates all spike trains conducing to all possible next configurations. All feasible transition steps are applied in parallel, repeating this process until a halting condition is met or until there is only one possible next configuration. Finally, the authors report a 2.31x speedup for 16 neurons in their benchmark, on a Linux server with two NVIDIA Tesla C1060 graphic card like the one mentioned above. Some ideas of this matrix representation are also used to simulate SN P systems with energy [72].

6.6.3.3 Population Dynamic P systems

Population Dynamics P (PDP) systems [29], is a framework that was initially devised to model ecosystems. PDP systems are composed of a directed graph whose nodes are called *environments*. Each one of these environments has an inner cell–like membrane structure, and a set of rules which communicate membranes inside and among environments. In PDP systems, the membrane structure and associated rules inside each environment are the same, solely varying the initial multisets and the probabilities associated with the rules in each environment. In addition, each rule has an associated probability function which dictates, provided it is applicable at a given configuration, how likely it is to be applied.

In their work, Martínez–del–Amor *et al.* [99] first developed a C++ simulator for PDP systems, which was further improved with OpenMP [203]. OpenMP is a programming library for Uniform Memory Access (UMA) parallel architectures, that is to say, architectures in which memory is centralized in a single device rather than distributed among nodes. This implementation was later adapted to CUDA, so as to run it in NVIDIA graphic cards. This simulator consists in a parallel implementation of DCBA algorithm (see [98] for further details). The authors achieve an acceleration up to 7x in comparison with its sequential counterpart and up to 3x compared to a 4–core CPU using OpenMP. This result proves the power of GPGPU Computing on the field of Membrane Computing simulation, and shows some limitations on the parallel simulation of P systems since it is memory bandwidth–bounded. Like in the case described above, the simulations were carried out on a Linux server with two NVIDIA Tesla C1060 graphic cards.

6.6.3.4 Kernel P systems

Recently, a simulator on Simple Kernel P systems has been developed by Ipate *et al.* [68] In this model, in order for a rule to be applied, it is not enough the availability of the objects acting as reactants with the needed amount. In addition to that constraint, a condition over a set of objects must be satisfied. In their work, the authors test their simulator on a P system solving the Subset Sum problem [37]. Although authors do not precisely specify how the non–determinism is implemented in their simulator, they report an acceleration of 10x for 16 subsets. In this case, the authors employed a personal computer with Windows 7 Professional and a NVIDIA GeForce GT650M with 1 GB of dedicated RAM installed.

6.6.3.5 Tissue P systems

Martínez–del–Amor *et al.* [35, 96] developed a GPU-based simulator for a specific solution to *SAT* with tissue P systems with cell division. Their implementation consists of five separate stages, as follows: *generation*, *exchange*, *synchronization*, *checking* and *output*. These phases are tightly coupled to the problem at hand and, due to their lack of generality, are not described here. The problem addressed by the simulator consists in simulating a P system family which solves the SAT problem (we refer to [153] for more details). They obtained an acceleration factor up to 10x by running their simulations on the aforementioned Linux server with two NVIDIA C1060 graphic cards. Furthermore, this work served as the basis of a broader objective, in order to study which P system features are managed better by the GPU than by the CPU. This study was conducted by comparing the simulator for the model solving the SAT problem with P systems with active membranes and this tissue simulator. Results show that the simulator for the model with active membranes runs faster on the GPU (63x vs 10x) due to the usage of charges and non–cooperative rules.

6.6.3.6 Enzymatic Numerical P systems

A sequential Java-based simulator for Numerical P systems (namely *SNUPS*) was developed by Arsene *et al.* [115]. On the other hand, García–Quismondo *et al.* [43, 47, 48] developed a GPU–based simulator (namely *ENPSCUDA*) and a C++ one (namely *ENPSC++*) in order to conduct the performance gain analysis. The CUDA simulator was run on an *NVIDIA GeForce GTX 460M*, with authors reporting a maximum speed–up factor of about 90x achieved on the comparison between ENPSCUDA and SNUPS, for a dummy model. However, the maximum speed–up factor obtained between ENPSCUDA and ENPSC++ was only 6.5x. The maximum speed–up factor on the simulation for another model was about 49x on the SNUPS vs ENPSCUDA comparison and about 10x on the ENPSC++ vs ENPSCUDA comparison.

6.6.3.7 Probabilistic Guarded P systems

Probabilistic Guarded P systems (*PGP systems*, for short) are intended to simulate real–life phenomena, specifically attached to ecological processes. In [43], author introduces a GPU–based simulator (namely *PGPCUDA*) and a C++ one (namely *PGPC++*) in order to conduct the performance gain analysis. The simulators are used to study a model of the *Pieris napi oleracea* [73]. In all scenarios, the model was simulated for 10 years varying the number of simulations per instance. Contrarily to what it would be expected, PGPC++ outperformed PGPCUDA from the very beginning, so the author concluded that further research remained to be conducted.

6.6.4 The PMCGPU project

PMCGPU project, a result of the work object of [94], was born to provide P system simulators working on HPC platforms, concretely on the *GPU*. PMCGPU project consists of several subprojects, aiming to simulate different P systems models:

- *PCUDA*: the first P system simulator based on CUDA. It supports P systems with active membranes, with sequential and parallel versions.
- *PCUDASAT*: a branch of PCUDA, aiming to a fast simulation of a family of P systems with active membranes solving `SAT`. It is an *ad-hoc* (non-flexible) platform, so that it behaves as a `SAT` solver by means of P systems.
- *TSPCUDASAT*: a branch of PCUDASAT. A family of tissue P systems with cell division solving `SAT` is simulated in a fast and scalable way.
- *ABCD-GPU*: a project on the simulation of PDP systems. It includes C++ based simulators running on both multicore (by the OpenMP [203] library) and manycore (by CUDA) platforms.

6.7 Hardware specifications

All simulations reported in this work have been performed on a laptop with a NVIDIA GT 520M card and an Intel i5 as CPU processor, as shown in Table 6.1.

Feature	**Value**
CPU Processor	Intel i5
CPU RAM Memory	4 GB DDR3
CPU Cache Memory	3 MB
CPU Clock Frequency	2.4 GHz
GPU Model	NVIDIA GT 520M
Number of GPU Cores	48
GPU Clock Frequency	1.48 GHz
GPU Memory	3 GB DDR3, shared
GPU Memory Clock Frequency	1.6 GHz

Table 6.1: Hardware specifications of the laptop in which the simulations in this work have been carried out

Part II

Contributions

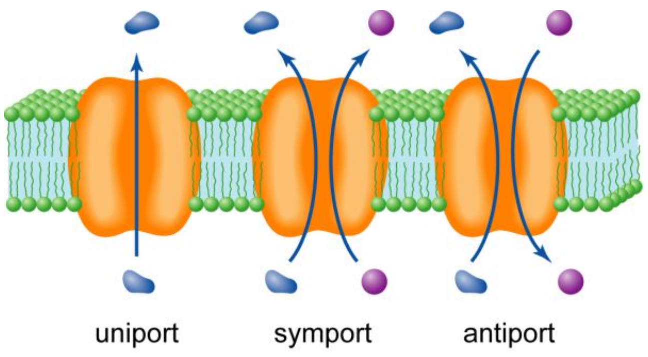
7 New frontiers of the efficiency

In Chapter 3, the computational efficiency of different models in membrane computing has been presented and new techniques and tools have been developed to tackle the **P** versus **NP** problem. For that, two framework has been considered: cell-like P systems with active membranes (with or without using electrical charges) and tissue-like P systems with cell division or cell separation. In both cases, the communication rules are different. In the case of cell-like P systems, evolution rules, send-in and send-out rules and dissolution rules are considered. In the case of tissue-like P systems, communication rules have been implemented by using symport/antiport rules.

In this chapter, cell-like P systems with symport/antiport rules are considered and their computational efficiency is studied when membrane division rules or membrane separation rules are allowed.

7.1 Introduction

Cell membranes contain a variety of transport proteins, each of which is responsible for transferring solute (inorganic ions, small organic molecules, or both) across the biological membranes. There are two classes of membrane transport proteins: *transporters* and *channels*. If a solute is present at a higher concentration outside the cell than inside and an appropriate channel or transporter is present in the plasma membrane, the solute will move spontaneously accross the membrane down its concentration gradient into the cell by passive transport, without expenditure of energy by its membrane transport protein. Active transport of solutes against their electrochemical gradient is essential to maintain the intracellular ionic composition of cells and to import molecules that are at a lower concentration outside the cell than inside. *Transporters* are integral membrane proteins which is involved in movement of two or more different types of solutes at the same time across biological membranes. If the transporter moves solutes in the same direction across the membrane, it is called a *symporter* (Figure 12–16). If it moves them in opposite directions, it is called an *antiporter*. A transporter that ferries only one type of solute across the membrane (and is therefore not a coupled transporter) is called a *uniporter* [1].



Channels form aqueous pores across the lipid bilayer through which solutes can diffuse. Whereas solute transfer carried out by transporters can be active or passive, transport by channels is always passive. Most channels are selective ion channels, which allow inorganic ions of appropriate size and charge to cross the membrane down their electrochemical gradients.

These kinds of transports have been incorporated in the framework of membrane computing by means of rewritting rules called *symport/antiport* rules. Specifically, cell-like *P systems with symport/antiport rules* were introduced in [145], aiming to abstract the biological phenomenon of trans-membrane transport of couples of chemical substances, in the same or in opposite directions, and the computational completeness of these models has been established (five membranes are enough if at most two objects are used in the rules). On the other hand, networks of membranes which compute by communication only, using symport/antiport rules were considered in [146] (*tissue P systems with symport/antiport rules*). These networks aim to abstract networks of elementary cells such that some of them are linked by "communication channels" abstracting the trans-membrane transport of chemical substances, in the same or opposite directions. Such rules are used both for communication with the environment and for direct communication between different membranes. It is worth noting that, in these systems, the environment plays an active role because we cannot only send objects outside the system, but we can also bring in objects from the environment. In [146] networks with a small number of membranes were proved to be computationally universal.

Two relevant processes take place in eukaryotic cells take place: *mitosis* and *membrane fission*. The first one is a nuclear division process during which one cell gives place to two genetically identical children cells. *Membrane fission* occurs when a membrane gives place to two separated membranes, that is, whenever a vesicle is produced or a larger subcellular compartment is divided into smaller discrete units. These processes have been a source of inspiration to incorporate new syntactical ingredients in membrane computing in order to be able to produce exponential workspace in polynomial – often linear – time. Specifically, inspired by the mitosis process, *membrane division* rules were defined in the framework of cell-like P systems providing computing devices called *P systems with active membranes* [150]). By applying this kind of rules, the object triggering them is replaced by other objects into the new membranes and the remaining objects are *duplicated*. The biological phenomenon of membrane fission was incorporated in *Membrane Computing* through a new kind of rules, called *membrane separation rules*, in the framework of polarizationless P systems with active membranes [2]. These rules were originally associated to (eventually) different subsets of the working alphabet. Nevertheless, in [118] a new definition of separation rules in the framework of P systems with active membranes was introduced. In the new definition there exists a partition of the working alphabet in two subsets such that each separation rule is associated to that given partition. By applying this kind of rules, the object triggering them is consumed and the remaining objects are *distributed* both in the created membranes.

7.1.1 P Systems with Symport/Antiport Rules

In this section, we introduce a kind of cell-like P systems that use communication rules capturing the biological phenomenon of trans-membrane transports of several chemical substances. Specifically, two processes have been considered. The first one allows a multiset of chemical substances to pass through a membrane in the same direction. In the second one, two multisets of chemical substances (located in different biological membranes) only pass with the help of each other (an *exchange* of objects between both membranes). In what follows, we introduce an abstraction of these operations in the framework of P systems with symport/antiport rules following [146]. In these models, the membranes are not polarized.

7.1.1.1 Basic P Systems with Symport/Antiport Rules

Definition 7.1. *A P system with symport/antiport rules of degree* $q \geq 1$ *is a tuple* $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$*, where:*

1. Γ *is a finite* alphabet*;*
2. $\mathcal{E} \subsetneq \Gamma$*;*
3. μ *is a membrane structure (a rooted tree) whose nodes are injectively labelled with* $1, 2 \dots, q$ *(the root of the tree is labelled by* 1*);*
4. $\mathcal{M}_1, \dots, \mathcal{M}_q$ *are finite multises over* Γ*;*
5. $\mathcal{R}_1, \cdots, \mathcal{R}_q$ *are finite set of communication rules of the following forms:*
   - $\star$ Symport rules*:* (u, out) *or* (u, in)*, where* u *is a finite multiset over* Γ *such that* $|u| > 0$*;*
   - $\star$ Antiport rules*:* $(u, out; v, in)$*, where* u, v *are finite multisets over* Γ *such that* $|u| > 0$ *and* $|v| > 0$*;*
6. $i_{out} \in \{0, 1, \dots, q\}$.

A P system with symport/antiport rules of degree q

$$\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$$

can be viewed as a set of q membranes, labelled by $1, \dots, q$, arranged in a hierarchical structure μ given by a rooted tree whose root is called the *skin membrane*, such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ represent the finite multisets of objects initially placed in the q membranes of the system; (b) $\mathcal{E}$ is the set of objects initially located in the environment of the system, all of them available in an arbitrary number of copies; (c) $\mathcal{R}_1, \cdots, \mathcal{R}_q$ are finite sets of communication rules over Γ (the set $\mathcal{R}_i$ is associated with the membrane i of μ); and (d) i_{out} represents a distinguished *zone* which will encode the output of the system. We use the term *zone* i $(0 \leq i \leq q)$ to refer to membrane i in the case $1 \leq i \leq q$ and to refer to the environment in the case $i = 0$. The length of rule (u, out) or (u, in) (resp. $(u, out; v, in)$) is defined as $|u|$ (resp. $|u| + |v|$).

For each membrane $i \in \{2, \dots, q\}$ (different from the skin membrane) we denote by $p(i)$ the parent of membrane i in the rooted tree μ. We define $p(1) = 0$, that is, by convention the "parent" of the skin membrane is the environment.

If the alphabet of the environment is an empty set then we say that the P system is *without environment*. This term means that there is not any objects initially located in the environment of the system available in an arbitrary number of copies, that is, in P systems without environment there is an environment (labelled by 0 as usual) but in any moment each object in it has a *finite multiplicity*.

An *instantaneous description* or a *configuration* at an instant t of a P system with symport/antiport rules is described by the membrane structure at instant t, all multisets of objects over Γ associated with all the membranes present in the system, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ associated with the environment at that moment. Recall that there are infinite copies of objects from $\mathcal{E}$ in the environment, and hence this set is not properly changed along the computation. The *initial configuration* of the system is $(\mu, \mathcal{M}_1, \cdots, \mathcal{M}_q; \emptyset)$.

A symport rule $(u, out) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in such membrane. When applying a rule $(u, out) \in \mathcal{R}_i$, the objects specified by u are sent out of membrane i into the region immediately outside (the parent $p(i)$ of i), this can be the environment in the case of the skin membrane.

A symport rule $(u, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in the parent of i. When applying a rule $(u, in) \in \mathcal{R}_i$, the multiset of objects u goes out from the parent membrane of i and enters into the region defined by the membrane i.

An antiport rule $(u, out; v, in) \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if membrane i is in $\mathcal{C}_t$ and multiset u is contained in such membrane, and multiset v is contained in the parent of i. When applying a rule $(u, out; v, in) \in \mathcal{R}_i$, the objects specified by u are sent out of membrane i into the parent of i and, at the same time, bringing the objects specified by v into membrane i.

The rules of a P system with symport/antiport rules are applied in a nondeterministic maximally parallel manner: at each step we apply a multiset of rules which is maximal, no further applicable rule can be added.

Let us fix a P system with symport/antiport rules Π. We say that configuration $\mathcal{C}^1$ yields configuration $\mathcal{C}^2$ in one *transition step*, denoted by $\mathcal{C}^1 \Rightarrow_\Pi \mathcal{C}^2$, if we can pass from $\mathcal{C}^1$ to $\mathcal{C}^2$ by applying the rules from $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_q$ following the previous remarks. A *computation* of Π is a (finite or infinite) sequence of configurations such that: (a) the first term of the sequence is the initial configuration of the system; (b) each non-initial configuration of the sequence is obtained from the previous configuration by applying rules of the system in a maximally parallel manner with the restrictions previously mentioned; and (c) if the sequence is finite (called *halting computation*) then the last term of the sequence is a *halting configuration* (a configuration where no rule of the system is applicable to it).

All computations start from an initial configuration and proceed as stated above; only halting computations give a result, which is encoded by the objects present in the output zone i_{out} in the halting configuration. If $\mathcal{C} = \{\mathcal{C}_t\}_{t \leq r}$ ($r \in \mathbb{N}$) is a halting computation of Π, then r is the *length of* $\mathcal{C}$, denoted by $|\mathcal{C}|$, that is, $|\mathcal{C}|$ is the number of non-initial configurations which appear in the finite sequence $\mathcal{C}$. We denote by $\mathcal{C}_t(i), 1 \leq i \leq q$, the multiset of objects over Γ contained in the membrane labelled by i at configuration $\mathcal{C}_t$. We also denote by $\mathcal{C}_t(0)$ the multiset of objects over $\Gamma \setminus \mathcal{E}$ contained in the environment at configuration $\mathcal{C}_t$.

7.1.1.2 P Systems with Symport/Antiport Rules and Division Rules

Definition 7.2. *A P system with symport/antiport rules and membrane division of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{out})$, where:*

1. *$(\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$ is a P system with symport/antiport rules.*

2. *$\mathcal{R}_i$, $1 \leq i \leq q$, are finite sets of rules over Γ of the following forms:*

   (a) Communication rules*:*

   (a.1) Symport rules*: (u, out) or (u, in), where u is a finite multiset over Γ such that $|u| > 0$;*

   (a.2) Antiport rules*: $(u, out; v, in)$, where u, v are finite multisets over Γ such that $|u| > 0$ and $|v| > 0$;*

   (b) Division rules*: $[\,a\,]_i \to [\,b\,]_i [\,c\,]_i$, where $a, b, c \in \Gamma$, $i \in \{2, \dots, q\}$, $i \neq i_{out}$, and i is the label of a leaf of the tree μ;*

3. *$i_{out} \in \{0, 1, \dots, q\}$.*

A P system with symport/antiport rules and membrane division is a P system with symport/antiport rules where division rules of cells are allowed.

A division rule $[a]_i \rightarrow [b]_i[c]_i \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if the following holds: (a) there exists a membrane labeled with i in $\mathcal{C}_t$; (b) object a is contained in such membrane; and (c) that membrane is neither the skin membrane nor the output membrane (if $i_{out} \in \{1, \dots, q\}$). When applying a division rule $[a]_i \rightarrow [b]_i[c]_i$ to a membrane labeled with i, under the influence of object a, the membrane is divided into two new ones with the same label; in the first new membrane, object a is replaced by object b, in the second one, object a is replaced by object c; all the other objects residing in such membrane are replicated and a copy of them is placed in each one of the two new membranes.

With respect to the semantics of P systems with symport/antiport rules and membrane division, rules are applied in a non-deterministic maximally parallel manner with the following important remark: when a membrane i is affected by a division rule at a computation step, this is the only rule from $\mathcal{R}_i$ which can be applied to such membrane at that step. The new membranes resulting from division could participate in the interaction with other membranes or the environment by means of communication rules at the next step – providing that they are not divided once again. The label of a membrane precisely identifies the rules which can be applied to it.

7.1.1.3 P Systems with Symport/Antiport Rules and Separation Rules

Definition 7.3. *A P system with symport/antiport rules and membrane separation of degree $q \geq 1$ is a tuple*

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$$

where:

1. $\Pi = (\Gamma, \mathcal{E}, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{out})$ *is a P system with symport/antiport rules.*

2. $\{\Gamma_0, \Gamma_1\}$ *is a partition of* Γ, *that is,* $\Gamma = \Gamma_0 \cup \Gamma_1$, $\Gamma_0, \Gamma_1 \neq \emptyset$, $\Gamma_0 \cap \Gamma_1 = \emptyset$*;*

3. $\mathcal{R}_1, \cdots, \mathcal{R}_q$ *are finite set of rules of the form:*

   (a) Communication rules*:*

   $(a.1)$ Symport rules*:* (u, out) *or* (u, in)*, where* u *is a finite multiset over* Γ *such that* $|u| > 0$*;*

$(a.2)$ Antiport rules*:* $(u, out; v, in)$*, where* u, v *are finite multisets over* Γ *such that* $|u| > 0$ *and* $|v| > 0$*;*

(b) Separation rules*:* $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i$*, where* $i \notin \{1, i_{out}\}$ *and* $a \in \Gamma$*.*

A separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i \in \mathcal{R}_i$ is *applicable* to a configuration $\mathcal{C}_t$ at an instant t if the following holds: (a) there exists a membrane labelled with i in $\mathcal{C}_t$; (b) object a is contained in such membrane; and (c) that membrane i is neither the skin membrane nor the output membrane (if $i_{out} \in \{1, \ldots, q\}$). When applying a separation rule $[a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i \in \mathcal{R}_i$, in reaction with an object a, the membrane i is separated into two membranes with the same label; at the same time, object a is consumed; the objects from Γ_0 are placed in the first membrane, those from Γ_1 are placed in the second membrane.

With respect to the semantics of these variants, the rules of such P systems are applied in a non-deterministic maximally parallel manner (at each step we apply a multiset of rules which is maximal, no further applicable rule can be added), with the following important remark: when a membrane i is separated, the separation rule is the only one from $\mathcal{R}_i$ which is applied for that membrane at that step (however, some rules can be applied in a daughter membrane). The new membranes resulting from separation could participate in the interaction with other membranes or the environment by means of communication rules at the next step – providing that they are not divided (resp. separated) once again. The label of a membrane precisely identify the rules which can be applied to it.

The concept of recognizer membrane systems and the concept of efficient solvability by means of families of membrane systems are extended to P systems with symport/antiport rules and membrane division or membrane separation, in a natural way. We denote by $\mathbf{CDC}(k)$ (resp. $\mathbf{CSC}(k)$) the class of recognizer P systems with membrane division (resp. membrane separation) such that the communication rules (symport/antiport rules) of the system have length at most k. We also denote by $\widehat{\mathbf{CDC}}(k)$ (resp. $\widehat{\mathbf{CSC}}(k)$) the class of recognizer P systems with membrane division (resp. membrane separation) and <u>without environment</u> such that the communication rules of the system have length at most k. If $\mathcal{R}$ is a class of recognizer P system, we denote by $\mathbf{PMC}_{\mathcal{R}}$ the set of all decision problems which can be solved in polynomial time (and in a uniform way) by means of membrane systems from $\mathcal{R}$.

7.2 Feasibility of Cell-like P Systems with Symport/Antiport

In this section, we investigate the limitation on the computational efficiency (ability so solve hard problems in polynomial time) in P systems with symport rules and with membrane division or membrane separation.

7.2.1 A characterization of P by using families from CDC(1)

First, we study the feasibility of P systems with membrane division and with symport rules with no cooperation, that is, by using communication rules involving only one object. Specifically, we show that the polynomial complexity class associated with the class of recognizer P systems from $\mathbf{CDC(1)}$ is equal to the class $\mathbf{P}$, that is, $\mathbf{P} = \mathbf{PMC_{CDC(1)}}$

Let $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ be a recognizer P system from $\mathbf{CDC(1)}$. We denote by $\mathcal{M}_j^*$, $1 \le j \le q$, the multiset over $\Gamma \times \{j\}$ obtained from $\mathcal{M}_j$ by replacing $a \in \Gamma$ with (a, j). In addition, for each finite multiset m over Σ, we denote m^* the multiset over $\Sigma \times \{i_{in}\}$ obtained from $\mathcal{M}_j$ by replacing $a \in \Sigma$ with (a, i_{in}).

The rules from $\mathcal{R}_1 \cup \cdots \cup \mathcal{R}_q$ are of the following form: (a, out), (b, in) and $[a]_i \to [b]_i\ [c]_i$. These rules can be considered, in a certain sense, as a *dependency* between the object triggering the rule and the object produced by its application.

- The rules in $\mathcal{R}_i$ of type (a, out) can be described as the pair (a, i) produces the pair $(a, p(i))$.
- The rules in $\mathcal{R}_i$ of type (b, in) can be described as the pair $(b, p(i))$ produces the pair (b, i).
- The rules in $\mathcal{R}_i$ of type $[a]_i \to [b]_i\ [c]_i$ can be described as the pair (a, i) produces the pairs (b, i) and (c, i).

We formalize these ideas in the following definition.

Definition 7.4. *Let* $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ *be a recognizer P system from* $\mathbf{CDC(1)}$*. The dependency graph associated with* Π *is the directed graph* $G_\Pi = (V_\Pi, E_\Pi)$ *defined as follows:*

- *The set of vertices is* $V_\Pi = \{s\} \cup VL_\Pi \cup VR_\Pi$, *where:*

$$\begin{aligned} VL_\Pi = \ & \{(a,i) \in \Gamma \times \{0,\dots,q\} \mid [(a, out) \in \mathcal{R}_i] \vee [\exists j \in ch(i)((a,in) \in \mathcal{R}_j)] \vee \\ & [\exists b, c \in \Gamma \ ([a]_i \to [b]_i[c]_i \in \mathcal{R}_i])\}; \end{aligned}$$

$$\begin{aligned} VR_\Pi = \ & \{(a,i) \in \Gamma \times \{0,\dots,q\} \mid [(a, in) \in \mathcal{R}_i] \vee [\exists j \in ch(i)((a,out) \in \mathcal{R}_j)] \vee \\ & [\exists b, c \in \Gamma([b]_i \to [a]_i[c]_i \in \mathcal{R}_i)]\}. \end{aligned}$$

- *The set of edges is:*

$$\begin{aligned} E_\Pi = \ & \{(s,(a,j)) \mid 1 \le j \le q \wedge (a,j) \in \mathcal{M}_j^*\} \cup \\ & \{((a,i),(b,j)) \in V_\Pi \times V_\Pi \mid [a = b] \wedge [j = p(i) \wedge (a, out) \in \mathcal{R}_i] \vee \\ & [a = b] \wedge [i = p(j) \wedge (a, in) \in \mathcal{R}_j] \vee \\ & [i = j] \wedge [\exists c \in \Gamma \ ([a]_i \to [b]_i[c]_i \in \mathcal{R}_i)]\}. \end{aligned}$$

In what follows, we show that the dependency graph associated with a P system from **CDC(1)**, can be constructed by a single deterministic Turing machine working in polynomial time.

Proposition 7.1. *Let* $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ *be a recognizer P system from* **CDC(1)**. *There exists a Turing machine that constructs the dependency graph,* G_Π, *associated with* Π, *in polynomial time (that is, in a time bounded by a polynomial function depending on the total number of rules and the maximum length of the rules).*

Proof. Given a recognizer P system Π from **CDC(1)** whose set of rules is $\mathcal{R} = \mathcal{R}_1 \cup \dots \cup \mathcal{R}_q$, a deterministic algorithm that constructs the corresponding dependency graph is the following:

```
Input: $(\Pi, \mathcal{R})$
$V_\Pi \leftarrow \{s\}$; $E_\Pi \leftarrow \emptyset$
for $j = 1$ to $q$ do
  for each pair $(a, j) \in \mathcal{M}_j^*$ do
    $E_\Pi \leftarrow E_\Pi \cup \{(s, (a, j))\}$
  end for
end for
for each rule $r \in \mathcal{R}$ of $\Pi$ do
  if $r = (a, in) \in \mathcal{R}_i$ then
    $V_\Pi \leftarrow V_\Pi \cup \{(a, p(i)), (a, i)\}$; $E_\Pi \leftarrow E_\Pi \cup \{((a, p(i)), (a, i))\}$
  end if
  if $r = (a, out) \in \mathcal{R}_i$ then
```

```
    $V_\Pi \leftarrow V_\Pi \cup \{(a,i),(a,p(i))\}$; $E_\Pi \leftarrow E_\Pi \cup \{((a,i),(a,p(i)))\}$
  end if
  if $r = [a]_i \rightarrow [b]_i[c]_i \in \mathcal{R}_i$ then
    $V_\Pi \leftarrow V_\Pi \cup \{(a,i),(b,i),(c,i)\}$;
    $E_\Pi \leftarrow E_\Pi \cup \{((a,i)),(b,i))\} \cup \{((a,i),(c,i))\}$
  end if
end for
```

The running time of this algorithm is bounded by $O(|\mathcal{R}|) \subset O(q \cdot |\Gamma|^3)$. □

Proposition 7.2. *Let* $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ *be a recognizer confluent P system from* **CDC(1)**. *The following assertions are equivalent:*

(1) There exists an accepting computation of Π.

(2) There exists a path (with length greater than or equal to 2) from s *to* $(\texttt{yes}, 0)$ *in the dependency graph associated with* Π.

Proof. (1) ⇒ (2). First, by induction on the length n of $\mathcal{C}$, we show that for each accepting computation $\mathcal{C}$ of Π there exists a path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with Π.

Let $n = 1$ and $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1)$ be an accepting computation of Π with length 1. Then, a rule of the form $(\texttt{yes}, out) \in \mathcal{R}_1$, with $a \in \Gamma$, has been applied at initial configuration $\mathcal{C}_0$. Then, $\texttt{yes} \in \mathcal{C}_0(1)$, so $(\texttt{yes}, 1) \in \mathcal{M}_1^*$. Hence, $(s, (\texttt{yes}, 1), (\texttt{yes}, 0))$ is a path from s to $(\texttt{yes}, 0)$ with length 2, in the dependency graph associated with Π.

Let us suppose that the result holds for n. Let $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_n, \mathcal{C}_{n+1})$ be an accepting computation of Π with length $n + 1$. In this situation, $\mathcal{C}' = (C_1, \dots, C_n, C_{n+1})$ is an accepting computation of the system $\Pi' = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}'_1, \dots, \mathcal{M}'_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$, with $\mathcal{M}'_j = \{(a,i) \in \Gamma \times \{0, \dots, q\} \mid C_1(j) = a\}$ being the "content" of membrane j in configuration $\mathcal{C}_1$, for $1 \leq j \leq q$. By induction hypothesis there exists a path $\gamma_{\mathcal{C}'} = (s, (b_1, i_1), \dots, (\texttt{yes}, 0))$ from s to $(\texttt{yes}, 0)$ in the dependency graph associated with Π' (with length greater than or equal to 2). We distinguish two cases.

- If $b_1 \in \mathcal{C}_0(i_1)$ (meaning that in the first step of computation $\mathcal{C}$, a division rule has been applied to membrane i_1 such that object b_1 does not appear in the rule), then $\gamma_{\mathcal{C}} = (s, (b_1, i_1), \dots, (\texttt{yes}, 0))$ is a path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with Π, and the result holds.

- Otherwise, there is an element $b_0 \in \mathcal{C}_0(i_0)$ producing (b_1, i_1) at the first step of computation $\mathcal{C}$. Hence, $\gamma_{\mathcal{C}} = (s, (b_0, i_0), (b_1, i_1), \ldots, (\texttt{yes}, 0))$ is a path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with Π, and the result holds.

(2) $\Rightarrow$ (1). By induction on the length k of the path, let us show that for each path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with Π, with length $k \geq 2$, there exists an accepting computation of Π.

Let $k = 2$ and $(s, (a_0, i_0), (\texttt{yes}, 0))$. Then, $i_0 = 1$ is the label of the skin membrane, $(a_0, out) \in \mathcal{R}_1$, $a_0 = \texttt{yes}$, and the computation $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1)$ is an accepting computation of Π, where the rule $(a_0, out) \in \mathcal{R}_1$ belongs to the multiset of rules that yields configuration C_1 from C_0.

Let us suppose that the result holds for $k \geq 2$, and let $(s, (a_0, i_0), (a_1, i_1), \ldots, (a_{k-1}, i_{k-1}), (\texttt{yes}, 0))$ be a path from s to $(\texttt{yes}, 0)$ in the dependency graph of length $k + 1$. If $(a_0, i_0) = (a_1, i_1)$, then the result holds by induction hypothesis. Otherwise, let $\mathcal{C}_1$ be a configuration of Π reached from $\mathcal{C}_0$ by the application of a multiset of rules containing a rule that yields (a_1, i_1) from (a_0, i_0). Then $(s, (a_1, i_1), \ldots (a_{k-1}, i_{k-1}), (\texttt{yes}, 0))$ is a path from s to $(\texttt{yes}, 0)$ of length k in the dependency graph of associated with the system $\Pi' = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}'_1, \ldots, \mathcal{M}'_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$, where $\mathcal{M}'_j = \{(a, i) \mid (a, i) \in \mathcal{C}_1(j)\}$ is the content of membrane j in configuration $\mathcal{C}_1$, for $1 \leq j \leq q$. By induction hypothesis, there exists an accepting computation $\mathcal{C}' = (\mathcal{C}_1, \ldots, \mathcal{C}_t)$ of Π'. Hence, $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_t)$ is an accepting computation of Π. □

Corollary 7.1. *Let $X = (I_X, \theta_X)$ be a decision problem. Let $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ be a family of recognizer P systems from* $\mathbf{CDC(1)}$ *solving X, according to Definition 2.14. Let (cod, s) be the polynomial encoding associated with that solution. Then, for each instance u of the problem X the following assertions are equivalent:*

(a) $\theta_X(u) = 1$, that is, the answer to the problem is yes *for u.*

(b) There exists a path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with the system $\Pi(s(u))$ with input multiset $cod(u)$.

Proof. Let $u \in I_X$. Then $\theta_X(u) = 1$ if and only if there exists an accepting computation of the system $\Pi(s(u)) + cod(u)$. Bearing in mind that $\Pi(s(u)) + cod(u)$ is a confluent system, from Proposition 7.2, we deduce that $\theta_X(u) = 1$ if and only if there exists a path from s to $(\texttt{yes}, 0)$ in the dependency graph associated with the system $\Pi(s(u)) + cod(u)$. □

Theorem 7.1. $\mathbf{P} = \mathbf{PMC}_{\mathbf{CDC(1)}}$.

Proof. We have $\mathbf{P} \subseteq \mathbf{PMC}_{\mathbf{CDC(1)}}$ because $\mathbf{PMC}_{\mathbf{CDC(1)}}$ is a nonempty class closed under polynomial–time reduction. Next, we show that $\mathbf{PMC}_{\mathbf{CDC(1)}} \subseteq \mathbf{P}$. For that, let $X \in \mathbf{PMC}_{\mathbf{CDC(1)}}$ and let $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbf{N}}$ be a family of recognizer P systems from $\mathbf{CDC(1)}$ solving X, according to Definition 2.14. Let (cod, s) be the polynomial encoding associated with that solution. We consider the following deterministic algorithm:

```
Input:  An instance $u$ of $X$
  - Construct the system $\Pi(s(u)) + cod(u)$
  - Construct the dependency graph $G_{\Pi(s(u))+cod(u)}$
  - Reachability $(G_{\Pi(s(u))+cod(u)}, s, (\texttt{yes}, 0))$
```

Obviously, this algorithm is polynomial in the size $|u|$ of the input. □

7.2.2 A characterization of P by using families from CSC(2)

In this section, we study the limitations on the computational efficiency of P systems with membrane separation which use symport/antiport rules with minimal cooperation. Specifically, we show that only problems in class **P** can be efficiently solved in polynomial time by means of families of recognizer P systems with membrane separation that use symport/antiport rules involving at most two objects. Hence, we prove that $\mathbf{P} = \mathbf{PMC}_{\mathbf{CSC(2)}}$.

Let us first introduce a new representation for the membrane structure of recognizer P systems with membrane separation. Let

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$$

be a recognizer P system of degree $q \geq 1$ from $\mathbf{CSC}(2)$. In order to identify the membranes created by the application of a separation rule, we introduce a new concept related to the labels of the new membranes in the following recursive manner:

- The label of a membrane will be a pair (i, σ), where $1 \leq i \leq q$ and σ is a string over $\{0, 1\}$. At the initial configuration, the labels of the membranes are $(1, \lambda), \dots, (q, \lambda)$.
- If a separation rule from $\mathcal{R}_i$ is applied to a membrane labelled by (i, σ), then the new created membranes will be labelled by $(i, \sigma 0)$ and $(i, \sigma 1)$,

respectively. Membrane $(i, \sigma 0)$ will only contain the objects of membrane (i, σ) which belong to Γ_0, and membrane $(i, \sigma 1)$ will only contain the objects of membrane (i, σ) which belong to Γ_1. The skin membrane cannot be separated, so the label of the skin membrane, $(1, \lambda)$, is not changed along any computation. Note that we can consider a lexicographical order over the set of labels of cells in the system along any computation.

If a membrane labelled by (i, σ) is engaged by a communication rule, then, after the application of the rule, the membrane keeps its label.

A configuration at an instant t of a P system from $\mathbf{CSC}(2)$ is described by the current membrane structure, the multisets of objects over Γ contained in each membrane, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ currently in the environment. Hence, a configuration of Π can be described by a multiset of labelled objects

$$\{(a, i, \sigma) \mid a \in \Gamma \cup \{\lambda\}, 1 \leq i \leq q, \sigma \in \{0,1\}^*\} \cup \{(a, 0) \mid a \in \Gamma \setminus \mathcal{E}\}.$$

Let us notice that the number of labels we need to identify all membranes appearing along any computation of a P system from $\mathbf{CSC}(2)$ is quadratic in the size of the initial configuration of the system and the length of the computation.

Let $r = (ab, out) \in \mathcal{R}_i$, $2 \leq i \leq q$, be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, i, \sigma)^n (b, i, \sigma)^n$, and we denote by $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ the multiset $(a, p(i), \tau)^n (b, p(i), \tau)^n$. In a similar way, $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ and $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ are defined when r is of the form $(a, out) \in \mathcal{R}_i$. Note that, at a given instant of the computation, for each membrane (i, σ) there is a unique parent membrane $(p(i), \tau)$, according to the current membrane structure.

Let $r = (ab, out) \in \mathcal{R}_1$ be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (1, \lambda), 0)$ the multiset of objects $(a, 1, \lambda)^n (b, 1, \lambda)^n$. We denote by $n \cdot RHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a, 0)^n (b, 0)^n, & \text{if } a, b \in \Gamma \setminus \mathcal{E}; \\ (a, 0)^n, & \text{if } a \in \Gamma \setminus \mathcal{E} \text{ and } b \in \mathcal{E}; \\ (b, 0)^n, & \text{if } b \in \Gamma \setminus \mathcal{E} \text{ and } a \in \mathcal{E}; \\ \emptyset, & \text{if } a, b \in \mathcal{E}. \end{cases}$$

In a similar way, $n \cdot LHS(r, (1, \lambda), 0)$ and $n \cdot RHS(r, (1, \lambda), 0)$ are defined when r is of the form $(a, out) \in \mathcal{R}_1$.

Let $r = (ab, in) \in \mathcal{R}_i$, $2 \leq i \leq q$, be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, p(i), \tau)^n (b, p(i), \tau)^n$. We denote by $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, i, \sigma)^n (b, i, \sigma)^n$. Similarly, $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ and $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ are defined when r is of the form $(a, in) \in \mathcal{R}_i$.

Let $r = (ab, in) \in \mathcal{R}_1$ be a symport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a,0)^n(b,0)^n\,, & \text{if } a, b \in \Gamma \setminus \mathcal{E}; \\ (a,0)^n\,, & \text{if } a \in \Gamma \setminus \mathcal{E} \text{ and } b \in \mathcal{E}; \\ (b,0)^n\,, & \text{if } b \in \Gamma \setminus \mathcal{E} \text{ and } a \in \mathcal{E}; \\ \emptyset\,, & \text{if } a, b \in \mathcal{E}. \end{cases}$$

We denote by $n \cdot RHS(r, (1, \lambda), 0)$ the multiset of objects $(a, 1, \lambda)^n (b, 1, \lambda)^n$. In a similar way, $n \cdot LHS(r, (1, \lambda), 0)$ and $n \cdot RHS(r, (1, \lambda), 0)$ are defined when r is of the form $(a, in) \in \mathcal{R}_1$.

Let $r = (a, out; b, in) \in \mathcal{R}_i$, $2 \leq i \leq q$, be an antiport rule of Π and $n \in \mathbb{N}$. We denote by $n \cdot LHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, i, \sigma)^n (b, p(i), \tau)^n$. Similarly, we denote by $n \cdot RHS(r, (i, \sigma), (p(i), \tau))$ the multiset of objects $(a, p(i), \tau)^n (b, i, \sigma)^n$.

Let $r = (a, out; b, in) \in \mathcal{R}_1$ be an antiport rule of Π. We denote by $n \cdot LHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a,1,\lambda)^n(b,0)^n\,, & \text{if } b \in \Gamma \setminus \mathcal{E}; \\ (a,1,\lambda)^n\,, & \text{if } b \in \mathcal{E}. \end{cases}$$

Similarly, we denote by $n \cdot RHS(r, (1, \lambda), 0)$ the following multiset of objects:

$$\begin{cases} (a,0)^n(b,1,\lambda)^n\,, & \text{if } a \in \Gamma \setminus \mathcal{E}; \\ (b,1,\lambda)^n\,, & \text{if } a \in \mathcal{E}. \end{cases}$$

If $\mathcal{C}_t$ is a configuration of Π, then we denote by $\mathcal{C}_t + \{(x, i, \sigma)/\sigma'\}$ the multiset obtained by replacing in $\mathcal{C}_t$ every occurrence of (x, i, σ) by (x, i, σ'). Besides, $\mathcal{C}_t + m$ (resp., $\mathcal{C}_t \setminus m$) is used to denote that a multiset m of labelled objects is added (resp., removed) to the configuration.

In order to show that only tractable problems can be solved efficiently by using families of P systems from $\mathbf{CSC}(2)$, we first state a technical result concerning recognizer P systems from $\mathbf{CSC}(2)$ (see [89] for more details).

Lemma 7.1. *Let* $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$ *be a recognizer P system of degree* $q \geq 1$ *from* $\mathbf{CSC}(2)$. *Let* $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$ *and let* $\mathcal{C} = \{\mathcal{C}_0, \ldots, \mathcal{C}_r\}$ *be a computation of* Π. *Then, we have*

(1) $|\mathcal{C}_0^*| = M$, *and for each* t, $0 \leq t < r$, $\mathcal{C}_{t+1}^* \cap (\Gamma \setminus \mathcal{E}) \subseteq \mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E})$*;*

(2) for each t, $0 \leq t \leq r$, $\mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E}) \subseteq (\mathcal{M}_1 + \cdots + \mathcal{M}_q) \cap (\Gamma \setminus \mathcal{E})$, *and* $|\mathcal{C}_t^* \cap (\Gamma \setminus \mathcal{E})| \leq M$*;*

(3) for each t, $0 \leq t < r$, $|\mathcal{C}_{t+1}^*| \leq |\mathcal{C}_t^*| + M$*;*

(4) for each t, $0 \leq t \leq r$, $|\mathcal{C}_t^*| \leq M \cdot (1 + t)$*;*

(5) the number of membranes created along computation $\mathcal{C}$ *by the application of separation rules is bounded by* $2M \cdot (1 + r)$*.*

Next, we present a deterministic algorithm $\mathcal{A}$ working in polynomial time that receives as an input a P system Π from $\mathbf{CSC}(2)$ and an input multiset m of Π, in such manner that algorithm $\mathcal{A}$ reproduces the behaviour of a computation of $\Pi + m$. In particular, if Π is confluent, then algorithm $\mathcal{A}$ will provide the same answer of the system Π.

The pseudocode of the algorithm $\mathcal{A}$ is described as follows:

```
Input: A P system $\Pi$ from $\mathbf{CSC}(2)$ and an input multiset $m$
  Initialization phase: $\mathcal{C}_0$ is the initial configuration of $\Pi + m$
  $t \leftarrow 0$
  while $\mathcal{C}_t$ is a non halting configuration do
    Selection phase: Input $\mathcal{C}_t$, Output $(\mathcal{C}'_t, A)$
    Execution phase: Input $(\mathcal{C}'_t, A)$, Output $\mathcal{C}_{t+1}$
    $t \leftarrow t + 1$
  end while
Output: Yes if object yes appears in the environment associated
with the halting configuration $\mathcal{C}_t$, No otherwise
```

The algorithm $\mathcal{A}$ receives a recognizer P system

$$\Pi = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$$

from $\mathbf{CSC}(2)$ and an input multiset m. Let $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$, $p \in \mathbb{N}$ be a natural number such that any computation of $\Pi + m$ performs, at most, p transition steps. Hence, from Lemma 7.1, we know that the number of membranes in the system along any computation is bounded by $2M(1+p)+q$.

A transition step of a recognizer P system $\Pi + m$ is performed by the selection and the execution phases. Specifically, the selection phase receives as an input a configuration $\mathcal{C}_t$ of $\Pi + m$ at an instant t. The output of this phase is a pair $(\mathcal{C}'_t, A)$, where A encodes a multiset of rules selected to be applied to $\mathcal{C}_t$, and $\mathcal{C}'_t$ is the configuration obtained from $\mathcal{C}_t$ once the labelled objects corresponding to the left-hand side of the rules from A have been consumed. The execution phase receives as an input the pair $(\mathcal{C}'_t, A)$, and the output of this phase is the next configuration $\mathcal{C}_{t+1}$ of $\mathcal{C}_t$. More precisely, configuration $\mathcal{C}_{t+1}$ is obtained from $\mathcal{C}'_t$ by adding the labelled objects produced by the application of rules from A; that is, the labelled objects corresponding to the right-hand side of the rules from A.

Selection phase.

```
  Input: A configuration $\mathcal{C}_t$ of $\Pi + m$ at instant $t$
$\mathcal{C}'_t \leftarrow \mathcal{C}_t$; $A \leftarrow \emptyset$; $B \leftarrow \emptyset$
for $r = (u, out; v, in) \in \mathcal{R}_i, 2 \leq i \leq q$ according to the order
   chosen do
   for each membrane $(i, \sigma)$ of $\mathcal{C}'_t$ according to the lexicographical
      order do
      $n_r \leftarrow$ maximum number of times that $r$ is applicable to $(i, \sigma)$
      if $n_r > 0$ then
         $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus n_r \cdot LHS(r, (i, \sigma), (p(i), \tau))$
         $A \leftarrow A \cup \{(r, n_r, (i, \sigma), (p(i), \tau))\}$
         $B \leftarrow B \cup \{(i, \sigma), (p(i), \tau)\}$
      end if
   end for
end for
for $r = (u, out; v, in) \in \mathcal{R}_1$ according to the order chosen do
   $n_r \leftarrow$ maximum number of times that $r$ is applicable to $(1, \lambda)$
   if $n_r > 0$ then
      $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus n_r \cdot LHS(r, (1, \lambda), 0)$
      $A \leftarrow A \cup \{(r, n_r, (1, \lambda), 0)\}$
   end if
end for
for $r = [\,a\,]_i \rightarrow [\,\Gamma_0\,]_i[\,\Gamma_1\,]_i \in \mathcal{R}_i$ $(i \neq 1)$ according to the
   order chosen do
   for each $(a, i, \sigma) \in \mathcal{C}'_t$ according to the lexicographical
      order, and such that $(i, \sigma) \notin B$ do
      $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus \{(a, i, \sigma)\}$
      $A \leftarrow A \cup \{(r, 1, (i, \sigma))\}$
      $B \leftarrow B \cup \{(i, \sigma)\}$
   end for
  end for
```

This algorithm is deterministic and works in polynomial time. Indeed, the running time of the previous algorithm is polynomial in the size of Π because: the number of cycles of the first main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q)$; the number of cycles of the second main loop **for** is of order $O(|\mathcal{R}|)$; and the number of cycles of the third main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q \cdot |\Gamma|)$.

Execution phase.

```
 Input: The output $(\mathcal{C}'_t, A)$ of the selection phase
for each $(r, n_r, (i, \sigma), (p(i), \tau)) \in A$ do
  $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (i, \sigma), (p(i), \tau))$
end for
for each $(r, n_r, (1, \lambda), 0) \in A$ do
  $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (1, \lambda), 0)$
end for
for each $(r, 1, (i, \sigma)) \in A$ do
  $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma)/\sigma 0\}$
  $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma 1)\}$
  for each $(x, i, \sigma) \in C'_t$ according to the lexicographical
    order do
    if $x \in \Gamma_0$ then
      $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 0\}$
    else
      $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 1\}$
    end if
  end for
end for
$\mathcal{C}_{t+1} \leftarrow \mathcal{C}'_t$
```

This algorithm is deterministic and works in polynomial time. Indeed, the running time of the previous algorithm is polynomial in the size of Π because: the number of cycles of the first main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q)$; the number of cycles of the second main loop **for** is of order $O(|\mathcal{R}|)$; and the number of cycles of the third main loop **for** is of order $O(|\mathcal{R}| \cdot M \cdot p \cdot q \cdot |\Gamma|)$.

Theorem 7.2. $\mathbf{P} = \mathbf{PMC}_{\mathbf{CSC}(2)}$.

Proof. It suffices to show that $\mathbf{PMC}_{\mathbf{CSC}(2)} \subseteq \mathbf{P}$. Let $X \in \mathbf{PMC}_{\mathbf{CSC}(2)}$ and let $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ be a family of recognizer P systems from $\mathbf{CSC}(2)$ solving X, according to Definition 2.14. Let (cod, s) be a polynomial encoding associated with that solution. If $u \in I_X$ is an instance of the problem X, then u will be processed by the system $\Pi(s(u)) + cod(u)$.

Let us consider the following deterministic algorithm $\mathcal{A}'$:

```
Input:  an instance $u$ of the problem $X$
   Construct the system $\Pi(s(u)) + cod(u)$
   Run algorithm $\mathcal{A}$ with input $\Pi(s(u)) + cod(u)$
 Output:  $Yes$ if algorithm $\mathcal{A}$ returns $Yes$,
          $No$ otherwise.
```

The algorithm $\mathcal{A}'$ receives as an input an instance u of the decision problem $X = (I_X, \theta_X)$ and works in polynomial time with respect to the size of the input. The following assertions are equivalent:

- $\theta_X(u) = 1$; that is, the answer of problem X to instance u is affirmative.
- Every computation of $\Pi(s(u)) + cod(u)$ is an accepting computation.
- The output of algorithm $\mathcal{A}'$ with input u is *Yes*.

Hence, $X \in \mathbf{P}$.

□

7.2.3 A characterization of P by using families from $\widehat{\mathbf{CSC}}$

Let $\Pi = (\Gamma, \Gamma_0, \Gamma_1, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ be a recognizer P system from $\widehat{\mathbf{CSC}}$ of degree $q \geq 1$.

- We denote by $f(i)$ (resp., $ch(i)$) the label of the father (resp., a child) of the membrane labelled by i, the father of the skin membrane is the environment (we write $f(1) = 0$). We denote by $\mathcal{R}_C$ (resp., $\mathcal{R}_S$) the set of communication rules (resp., separation rules) of Π. We will fix total orders in $\mathcal{R}_C$ and $\mathcal{R}_S$.
- Let $\mathcal{C}$ be a computation of Π, and $\mathcal{C}_t$ a configuration of $\mathcal{C}$. The application of a communication rule keeps the multiset of objects of the whole system unchanged because only movement of objects between the cells of the system is produced. On the other hand, the application of a separation rule causes that an object is removed from the system, and since there is no objects replication, the rest remain unchanged. Thus, the multiset of objects of the system in any configuration $\mathcal{C}_t$ is contained in $\mathcal{M}_0 + \dots + \mathcal{M}_q$. Moreover, if $M = |\mathcal{M}_0 + \dots + \mathcal{M}_q|$ then the total number of copies

of membrane i, $1 \leq i \leq q$, at configuration $\mathcal{C}$ is, at most, M because the copies can only be produced by the application of a separation rule, and each application of this kind of rule consumes one object. Consequently, $M \cdot q$ is an upper bound of the number of membranes at any configuration of the system.

- In order to identify the membranes created by the application of a separation rule, we modify the labels of the new membranes in the following recursive manner:
  - The label of a membrane will be a pair (i, σ) where $0 \leq i \leq q$ and $\sigma \in \{0, 1\}^*$. At the initial configuration, the labels of the membranes are $(1, \lambda), \dots, (q, \lambda)$. The label of the environment is denoted by $(0, \lambda)$.
  - If a separation rule is applied to a membrane labelled by (i, σ), then the new created membranes will be labelled by $(i, \sigma 0)$ and $(i, \sigma 1)$, respectively. Membrane $(i, \sigma 0)$ will only contain the objects of membrane (i, σ) which belong to Γ_0, and membrane $(i, \sigma 1)$ will only contain the objects of membrane (i, σ) which belong to Γ_1 (we only consider separation rules for elementary membranes). Only elementary membranes can be separated, so if a membrane i is non elementary then we denote it by the label (i, λ).
- If a membrane labelled by (i, σ) is engaged by a communication rule, then, after the application of the rule, the membrane keeps its label.
- A configuration $\mathcal{C}_t$ of a P system from $\widehat{\mathbf{CSC}}$ is described by the current membrane structure and the multisets of labelled objects over objects

  $$\{(a, i, \sigma) \mid a \in \Gamma \cup \{\lambda\}, 0 \leq i \leq q, \sigma \in \{0, 1\}^*\}$$

  The expression $(a, i, \sigma) \in \mathcal{C}_t$ means that object a belongs to membrane labelled by (i, σ). Let us notice that the number of labels we need to identify all membranes appearing along any computation of a P system from $\mathbf{CSC}(2)$ is quadratic in the size of the initial configuration of the system and the length of the computation.
- Let $r = (a_1, \dots, a_s\,, out\,; b_1, \dots, b_{s'}\,, in) \in \mathcal{R}_i$ be an antiport rule of Π. We denote by $n \cdot LHS(r, (i, \sigma))$, $n \in \mathbb{N}$, the following multiset of objects $(a_1, i, \sigma)^n \cdots (a_s, i, \sigma)^n (b_1, f(i), \tau)^n \cdots (b_{s'}, f(i), \tau)^n$, where $(f(i), \tau)$ is the father of membrane (i, σ). Similarly, $n \cdot RHS(r, (i, \sigma))$ denotes the

multiset of labelled objects produced by applying n times rule r over membrane (i, σ). That is, $n \cdot RHS(r, (i, \sigma))$ is the following multiset $(a_1, f(i), \tau)^n \cdots (a_s, f(i), \tau)^n (b_1, i, \sigma)^n \cdots (b_{s'}, i, \sigma)^n$.

- Let $r = (a_1, \dots, a_s\, , out) \in \mathcal{R}_i$ be a symport rule of Π (let us recall that $\mathcal{R}_1 = \emptyset$). We denote by $n \cdot LHS(r, (i, \sigma))$, $n \in \mathbb{N}$, the following multiset of labelled objects $(a_1, i, \sigma)^n \cdots (a_s, i, \sigma)^n$. Similarly, $n \cdot RHS(r, (i, \sigma))$ denotes the multiset of labelled objects produced by applying n times rule r over membrane (i, σ). That is, $n \cdot RHS(r, (i, \sigma))$ is the following multiset $(a_1, f(i), \tau)^n \cdots (a_s, f(i), \tau)^n$, where $(f(i), \tau)$ is the father of membrane (i, σ).

- Let $r = (a_1, \dots, a_s\, , in) \in \mathcal{R}_i$ be a symport rule of Π. We denote by $n \cdot LHS(r, (i, \sigma))$, $n \in \mathbb{N}$, the following multiset of labelled objects $(a_1, f(i), \tau)^n \cdots (a_s, f(i), \tau)^n$, where $(f(i), \tau)$ is the father of membrane (i, σ). Similarly, $n \cdot RHS(r, (i, \sigma))$ denotes the multiset of labelled objects produced by applying n times rule r over membrane (i, σ). That is, $n \cdot RHS(r, (i, \sigma))$ is the following multiset $(a_1, i, \sigma)^n \cdots (a_s, i, \sigma)^n$.

- Let $\mathcal{C}_t$ is a configuration of Π, we denote by $\mathcal{C}_t + \{(x, i, \sigma)/\sigma'\}$ the multiset obtained by replacing in $\mathcal{C}_t$ every occurrence of (x, i, σ) by (x, i, σ'). Besides, $\mathcal{C}_t + m$ (resp., $\mathcal{C}_t \setminus m$) is used to denote that a multiset m of labelled objects is added (resp., removed) to the configuration.

Next, we provide a deterministic algorithm $\mathcal{A}$ working in polynomial time that receives as input a recognizer tissue P system Π from $\widehat{\mathbf{CSC}}$ together with an input multiset m of Π. Then algorithm $\mathcal{A}$ reproduces the behaviour of a single computation of such system.

The pseudocode of the algorithm $\mathcal{A}$ is described as follows:

```
Input: A P system $\Pi$ from $\widehat{\mathbf{CSC}}$ and an input multiset $m$ of $\Pi$
    Initialization stage: the initial configuration $\mathcal{C}_0$ of $\Pi+m$
    $t \leftarrow 0$
    while $\mathcal{C}_t$ is a non halting configuration do
        Selection stage: Input $\mathcal{C}_t$, Output $(\mathcal{C}'_t, A)$
        Execution stage: Input $(\mathcal{C}'_t, A)$, Output $\mathcal{C}_{t+1}$
        $t \leftarrow t+1$
    end while
Output: Yes if $\mathcal{C}_t$ is an accepting configuration, No otherwise
```

The selection stage and the execution stage implement a transition step of a recognizer P system Π. Specifically, the selection stage receives as input a configuration $\mathcal{C}_t$ of Π at an instant t. The output of this stage is a pair $(\mathcal{C}'_t, A)$, where A encodes a multiset of rules selected to be applied to $\mathcal{C}_t$, and $\mathcal{C}'_t$ is the configuration obtained from $\mathcal{C}_t$ once the labelled objects corresponding to the application of rules from A have been consumed. The execution stage receives as input the output $(\mathcal{C}'_t, A)$ of the selection stage, and the output is the next configuration $\mathcal{C}_{t+1}$ of $\mathcal{C}_t$. Specifically, at this stage, the configuration $\mathcal{C}_{t+1}$ is obtained from $\mathcal{C}'_t$ by adding the labelled objects produced by the application of rules from A.

Next, selection stage and execution stage are described in detail.

Selection stage.

```
Input: A configuration $\mathcal{C}_t$ of $\Pi$ at instant $t$
  $\mathcal{C}'_t \leftarrow \mathcal{C}_t$;  $A \leftarrow \emptyset$;   $B \leftarrow \emptyset$
  for each $r \in \mathcal{R}_C \cap \mathcal{R}_i$ according to the order chosen do
    for each membrane $(i,\sigma)$ of $\mathcal{C}'_t$ according to the lexicographical order do
      $n_r \leftarrow$ maximum number of times that $r$ is applicable to $(i,\sigma)$
      if $n_r > 0$ then
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus n_r \cdot LHS(r,(i,\sigma))$
        $A \leftarrow A \cup \{(r, n_r, (i,\sigma))\}$
        $B \leftarrow B \cup \{(i,\sigma)\}$
      end if
    end for
  end for
  for $r \equiv [a]_i \rightarrow [\Gamma_0]_i[\Gamma_1]_i \in \mathcal{R}_i$ according to the order chosen do
    if $(i,\sigma) \notin B$ then
      for each $(a,i,\sigma) \in \mathcal{C}'_t$ according to the lexicographical order
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t \setminus \{(a,i,\sigma)\}$
        $A \leftarrow A \cup \{(r, 1, (i,\sigma))\}$
        $B \leftarrow B \cup \{(i,\sigma)\}$
      end for
    end if
  end for
```

This algorithm is deterministic and works in polynomial time. Indeed, the cost in time of the previous algorithm is polynomial in the size of Π because the number of cycles of the first main loop **for** is of order $O(|R| \cdot M \cdot q)$, and the number of cycles of the second main loop **for** is of order $O(|R| \cdot |\Gamma| \cdot M \cdot q)$. Besides, the last loop includes a membership test of order $O(M \cdot q)$.

In order to complete the simulation of a computation step of the system Π, the execution stage takes care of the effects of applying the rules selected in the previous stage: updating the objects according to the RHS of the rules.

Execution stage.

```
Input: The output $\mathcal{C}'_t$ and $A$ of the selection stage
    for each $(r, n_r, (i, \sigma)) \in A$ do
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + n_r \cdot RHS(r, (i, \sigma))$
    end for
    for each $(r, 1, (i, \sigma)) \in A$ do
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma)/\sigma 0\}$
        $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(\lambda, i, \sigma 1)\}$
        for each $(x, i, \sigma) \in \mathcal{C}'_t$ according to the lexicographical order do
            if $x \in \Gamma_0$ then
                $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 0\}$
            else
                $\mathcal{C}'_t \leftarrow \mathcal{C}'_t + \{(x, i, \sigma)/\sigma 1\}$
            end if
        end for
    end for
    $\mathcal{C}_{t+1} \leftarrow \mathcal{C}'_t$
```

This algorithm is deterministic and works in polynomial time. Indeed, the cost in time of the previous algorithm is polynomial in the size of Π because the number of cycles of the first main loop **for** is of order $O(|R|)$, and the number of cycles of the second main loop **for** is of order $O(|R| \cdot |\Gamma| \cdot M \cdot q)$. Besides, inside the body of the last loop there is a membership test of order $O(|\Gamma|)$.

Theorem 7.3. $\mathbf{P} = \mathbf{PMC}_{\widehat{\mathbf{CSC}}}$.

Proof. It suffices to prove that $\mathbf{PMC}_{\widehat{\mathbf{CSC}}} \subseteq \mathbf{P}$. Let $k \in \mathbb{N}$ such that $X \in \mathbf{PMC}_{\widehat{\mathbf{CSC}}(k)}$ and let $\{\Pi(n) : n \in \mathbb{N}\}$ be a family of P systems from $\widehat{\mathbf{CSC}}(k)$ solving X according to Definition 2.14. Let (cod, s) be a polinomial encoding associated with that solution. Let us recall that instance $u \in I_X$ of the problem X is processed by the system $\Pi(s(u)) + cod(u)$.

Let us consider the following algorithm $\mathcal{A}'$:

```
Input: an instance $u$ of the decision problem $X = (I_X, \theta_X)$.
    Construct the system $\Pi(s(u)) + cod(u)$
    Run algorithm $\mathcal{A}$ with input $\Pi(s(u)) + cod(u)$
Output: Yes if $\Pi(s(u)) + cod(u)$ has an accepting computation, No otherwise
```

The algorithm $\mathcal{A}'$ receives as input an instance u of the decision problem $X = (I_X, \theta_X)$. The following assertions are equivalent:

1. $\theta_X(u) = 1$, that is, the answer of problem X to instance u is affirmative.

2. Every computation of $\Pi(s(u)) + cod(u)$ is an accepting computation.

3. The output of the algorithm with input u is *Yes*.

Therefore, algorithm $\mathcal{A}'$ provide a solution of the decision problem X. Bearing in mind that $\mathcal{A}'$ works in polynomial time, we finally deduce that $X \in \mathbf{P}$. $\square$

7.3 Computational Efficiency of Cell-like P Systems with Symport/Antiport

In this section we study the ability to solve **NP**–complete problems in an efficient way by means of families of recognizer P systems with membrane division or membrane separation. Specifically, we show that minimal cooperation in communication rules is enough when division rules are allowed. Nevertheless, by using membranes separation the computational efficiency is reached when the length of communication rules is at most three.

7.3.1 Computational Efficiency of Systems in CDC(2)

In this section we give a polynomial time solution to `HAM-CYCLE` problem, a well known **NP**-complete problem [49], by means of a family of recognizer P systems with membrane division which use symport/antiport rules involving minimal cooperation.

Let us recall that `HAM-CYCLE` problem is the following: *Given a directed graph, determine whether or not there exists a Hamiltonian cycle in the graph.*

7.3.1.1 A polynomial time solution of `HAM-CYCLE` problem in CDC(2)

For each $n, m \in \mathbb{N}$, we consider the recognizer P system with symport/antiport rules and membrane division of degree $11 + 2n + n^3$

$$\begin{aligned}\Pi(\langle n, m\rangle) \;=\; & (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_r\,(1 \le r \le 11)\,,\, \mathcal{M}_{a_{1,j}}\,(1 \le j \le n),\ \mathcal{M}_{a_{2,j}}\,(1 \le j \le n)\,,\\ & \mathcal{M}_{e_{i,j,k}}(1 \le i, j, k \le n)\,,\, \mathcal{R}_r\,(1 \le r \le 11)\,,\, \mathcal{R}_{a_{1,j}}\,(1 \le j \le n)\,,\\ & \mathcal{R}_{a_{2,j}}\,(1 \le j \le n)\ \mathcal{R}_{e_{i,j,k}}(1 \le i, j, k \le n))\end{aligned}$$

defined as follows:

(1) Working alphabet:

$$\begin{aligned}\Gamma \;=\; & \Sigma \cup \mathcal{E} \cup \{\beta_r \mid 0 \le r \le n^3+7\} \cup \{b'_r, b''_r, b'''_r, c'_r, c''_r, c'''_r, c''''_r \mid 1 \le r \le n^3\} \cup \\ & \{(i,j)'_k\,,\,(i,j)''_k \mid 1 \le i,j,k \le n\} \cup \{(i,j)''_{k,r} \mid 1 \le i,j,k \le n \wedge 1 \le r \le n^3\} \cup \\ & \{\alpha_0, a, a', a'', b, b', b'', b''', c, c', c'', c''', c'''', \mathtt{yes}, \mathtt{no}\},\end{aligned}$$

where the input alphabet is $\Sigma = \{(i,j)_k \mid 1 \le i,j,k \le n\}$, and the alphabet of the environment is $\mathcal{E} = \{\alpha_r \mid 1 \le r \le n^3+6\}$

(2) Membrane structure μ: the root is labelled by 1, and the remaining nodes are children of the root, being labelled by

$$2, 3, \ldots, 11, a_{1,j}\ (1 \le j \le n), a_{2,j}\ (1 \le j \le n), e_{i,j,k}\ (1 \le i,j,k \le n),$$

respectively.

(3) Initial multisets:

$\mathcal{M}_1 = \{\alpha_0\} \cup \{\beta_r \mid 1 \le r \le n^3+7\} \cup \{b'_r, b''_r, b'''_r, c'_r, c''_r, c'''_r, c''''_r \mid 1 \le r \le n^3-1\}$;
$\mathcal{M}_2 = \{a^n, b, c\}$;
$\mathcal{M}_3 = \{b'_{n^3}\}$; $\mathcal{M}_4 = \{b''_{n^3}\}$; $\mathcal{M}_5 = \{b'''_{n^3}\}$;
$\mathcal{M}_6 = \{c'_{n^3}\}$; $\mathcal{M}_7 = \{c''_{n^3}\}$; $\mathcal{M}_8 = \{c'''_{n^3}\}$; $\mathcal{M}_9 = \{c''''_{n^3}\}$;
$\mathcal{M}_{10} = \{\mathtt{yes}\}$; $\mathcal{M}_{11} = \{\mathtt{no}, \beta_0\}$;
$\mathcal{M}_{a_{1,j}} = \{a'_{n^3}\}$, $\mathcal{M}_{a_{2,j}} = \{a''_{n^3}\}, 1 \le j \le n$;
$\mathcal{M}_{e_{i,j,k}} = \{(i,j)''_{k,n^3}\}, 1 \le i,j,k \le n.$

(4) Rules of the system:

- Rules in $\mathcal{R}_1$:

1.1 Rules to control the output of the computations by counters of type α_r.

$$(\alpha_r\,,\,out\,;\,\alpha_{r+1}\,,\,in)\ ,\ 0 \le r \le n^3+5.$$

Rules 1.2 and 1.3 produce the output of the computations:

1.2 $(\mathtt{yes}\,,\,out)$

1.3 $(\mathtt{no}\,\alpha_{n^3+6}\,,\,out)$

- Rules in $\mathcal{R}_2$:

2.1 Rules to produce all possible subsets of A'_G in membranes labelled by 2 at configuration $\mathcal{C}_{n^3+1}$:

$$[\,(i,j)_k\,]_2 \to [\,(i,j)'_k\,]_2\ [\,\#\,]_2,\ 1 \le i,j,k \le n.$$

Rules 2.2, 2.3, 2.4 and 2.5 allow to introduce objects a', a'', b', b'', c''', c', c'', c''' and c'''' in membranes labelled by 2 at configurations $\mathcal{C}_{n^3+2}$, $\mathcal{C}_{n^3+3}$, $\mathcal{C}_{n^3+4}$ and $\mathcal{C}_{n^3+5}$, respectively:

2.2 $(a\,,\,out\,;\,a'\,,\,in)$; $(a'\,,\,out\,;\,a''\,,\,in)$;

2.3 $(b\,,\,out\,;\,b'\,,\,in)$; $(b'\,,\,out\,;\,b''\,,\,in)$; $(b''\,,\,out\,;\,b'''\,,\,in)$;

2.4 $(c\,,\,out\,;\,c'\,,\,in)$; $(c'\,,\,out\,;\,c''\,,\,in)$; $(c''\,,\,out\,;\,c'''\,,\,in)$; $(c'''\,,\,out\,;\,c''''\,,\,in)$;

2.5 $(a''\,b'''\,,\,out)$; $(b'''\,c''''\,,\,out)$.

2.6 Rules to produce in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ a subset of A''_G from a subset of A'_G at configuration $\mathcal{C}_{n^3+1}$:

$$((i,j)'_k\,,\,out\,;\,(i,j)''_k\,,\,in)\ ,\ 1 \leq i,j,k \leq n.$$

2.7 Rules to generate in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ a subset of A''_G encoding a possible Hamiltonian cycle.
$((i,j)''_k\ (i,j')''_{k'}\ ,\, out)$, $1 \leq i,i',j,j',k,k' \leq n$;
$((i,j)''_k\ (i',j)''_{k'}\ ,\, out)$, $1 \leq i,i',j,j',k,k' \leq n$;
$((i,j)''_k\ (i',j')''_{k+1}\ ,\, out)$, $1 \leq i,i',j,j',k,k' \leq n,\ j \neq i'$;
$((i,j)''_k\ (i',j')''_k\ ,\, out)$, $1 \leq i,i',j,j',k,k' \leq n$.

2.8 Rules to check if the subset represented by each membrane with label 2 at configuration $\mathcal{C}_{n^3+3}$ encodes a Hamiltonian cycle of the input graph:

$$(a''\,(i,j)''_k\,,\,out),\ 1 \leq i,j,k \leq n.$$

• Rules in $\mathcal{R}_3$:

Rules to produce $2^{n \cdot p}$ copies of objects b' in the skin membrane of configuration $\mathcal{C}_{n^3+1}$:

3.1 $(b'_r\,,\,out\,;\,b'_{r-1}\,,\,in)$, $n \cdot m + 1 \leq r \leq n^3$;

3.2 $[\,b'_r\,]_3 \to [\,b'_{r-1}\,]_3\ [\,b'_{r-1}\,]_3$, $2 \leq r \leq n \cdot m$;

3.3 $[\,b'_1\,]_3 \to [\,b'\,]_3\ [\,b'\,]_3$;

3.4 $(b'\ ,\,out)$.

• Rules in $\mathcal{R}_4$:

Rules to produce $2^{n \cdot p}$ copies of objects b'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

4.1 $(b''_r\,,\,out\,;\,b''_{r-1}\,,\,in)$, $n \cdot m + 1 \leq r \leq n^3$;

4.2 $[\,b''_r\,]_4 \rightarrow [\,b''_{r-1}\,]_4\ b''_{r-1}\,]_4,\ 2 \leq r \leq n \cdot m;$

4.3 $[\,b''_1\,]_4 \rightarrow [\,b''\,]_4\ [\,b''\,]_4;$

4.4 $(b''\ ,\ out).$

- Rules in $\mathcal{R}_5$:

Rules to produce $2^{n \cdot p}$ copies of objects b''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

5.1 $(b'''_r\ ,\ out\,;\ b'''_{r-1}\ ,\ in),\ n \cdot m + 1 \leq r \leq n^3;$

5.2 $[\,b'''_r\,]_5 \rightarrow [\,b'''_{r-1}\,]_5\ [\,b'''_{r-1}\,]_5,\ 2 \leq r \leq n \cdot m;$

5.3 $[\,b'''_1\,]_5 \rightarrow [\,b'''\,]_5\ [\,b'''\,]_5;$

5.4 $(b'''\ ,\ out).$

- Rules in $\mathcal{R}_6$:

Rules to produce $2^{n \cdot p}$ copies of objects c' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

6.1 $(c'_r\ ,\ out\,;\ c'_{r-1}\ ,\ in),\ n \cdot m + 1 \leq r \leq n^3;$

6.2 $[\,c'_r\,]_6 \rightarrow [\,c'_{r-1}\,]_6\ [\,c'_{r-1}\,]_6,\ 2 \leq r \leq n \cdot m;$

6.3 $[\,c'_1\,]_6 \rightarrow [\,c'\,]_6\ [\,c'\,]_6;$

6.4 $(c'\ ,\ out).$

- Rules in $\mathcal{R}_7$:

Rules to produce $2^{n \cdot p}$ copies of objects c'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

7.1 $(c''_r\ ,\ out\,;\ c''_{r-1}\ ,\ in),\ n \cdot m + 1 \leq r \leq n^3;$

7.2 $[\,c''_r\,]_7 \rightarrow [\,c''_{r-1}\,]_7\ [\,c''_{r-1}\,]_7,\ 2 \leq r \leq n \cdot m;$

7.3 $[\,c''_1\,]_7 \rightarrow [\,c''\,]_7\ [\,c''\,]_7;$

7.4 $(c''\ ,\ out).$

- Rules in $\mathcal{R}_8$:

Rules to produce $2^{n \cdot p}$ copies of objects c''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

8.1 $(c'''_r\ ,\ out\,;\ c'''_{r-1}\ ,\ in),\ n \cdot m + 1 \leq r \leq n^3;$

8.2 $[\, c'''_r \,]_8 \rightarrow [\, c'''_{r-1} \,]_8 \; [\, c'''_{r-1} \,]_8, \; 2 \leq r \leq n \cdot m;$

8.3 $[\, c'''_1 \,]_8 \rightarrow [\, c''' \,]_8 \; [\, c''' \,]_8;$

8.4 $(c''' \, , \, out).$

• Rules in $\mathcal{R}_9$:

Rules to produce $2^{n \cdot p}$ copies of objects c'''' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

9.1 $(c''''_r \, , \, out \, ; \, c''''_{r-1} \, , \, in), \; n \cdot m + 1 \leq r \leq n^3;$

9.2 $[\, c''''_r \,]_9 \rightarrow [\, c''''_{r-1} \,]_9 \; [\, c''''_{r-1} \,]_9, \; 2 \leq r \leq n \cdot m;$

9.3 $[\, c''''_1 \,]_9 \rightarrow [\, c'''' \,]_9 \; [\, c'''' \,]_9;$

9.4 $(c'''' \, , \, out).$

• Rules in $\mathcal{R}_{10}$:

Rules to produce an affirmative answer:

10.1 $(\alpha_{n^3+6} \, c'''' \, , \, in) \; ; \; (c'''' \, \texttt{yes} \, , \, out)$

• Rules in $\mathcal{R}_{11}$:

Rules to control the negative answer of the computations by counters β_r:

11.1 $(\beta_r \, out \, ; \, \beta_{r+1} \, , \, in), \; 0 \leq r \leq n^3 + 6;$

11.2 $(\beta_{n^3+7} \, \texttt{no} \, , \, out).$

• Rules in $\mathcal{R}_{a_{1,j}}, \;\; 1 \leq j \leq n$:

Rules to produce 2^{n^3} copies of objects a' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

$\mathbf{a_{1,j}.1}$ $[\, a'_r \,]_{a_{1,j}} \rightarrow [\, a'_{r-1} \,]_{a_{1,j}} \; [\, a'_{r-1} \,]_{a_{1,j}}, \; 2 \leq r \leq n^3;$

$\mathbf{a_{1,j}.2}$ $[\, a'_1 \,]_{a_{1,j}} \rightarrow [\, a' \,]_{a_{1,j}} \; [\, a' \,]_{a_{1,j}};$

$\mathbf{a_{1,j}.3}$ $(a' \, , \, out).$

• Rules in $\mathcal{R}_{a_{2,j}}, \;\; 1 \leq j \leq n$:

Rules to produce 2^{n^3} copies of objects a'' in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

$\mathbf{a_{2,j}.1}$ $[\, a''_r \,]_{a_{2,j}} \to [\, a''_{r-1} \,]_{a_{2,j}} \; [\, a''_{r-1} \,]_{a_{2,j}}, \; 2 \leq r \leq n^3;$

$\mathbf{a_{2,j}.2}$ $[\, a''_1 \,]_{a_{2,j}} \to [\, a'' \,]_{a_{2,j}} \; [\, a'' \,]_{a_{2,j}};$

$\mathbf{a_{2,j}.3}$ $(a''\,,\, out).$

- Rules in $\mathcal{R}_{e_{i,j,k}}, \; 1 \leq i, j, k \leq n$:

  Rules to produce 2^{n^3} copies of objects $(i,j)''_k$ in the skin membrane at configuration $\mathcal{C}_{n^3+1}$:

$\mathbf{e_{i,j,k}.1}$ $[\, (i,j)''_{k,r} \,]_{e_{i,j,k}} \to [\, (i,j)''_{k,r-1} \,]_{e_{i,j,k}} \; [\, (i,j)''_{k,r-1} \,]_{e_{i,j,k}}, \; 2 \leq r \leq n^3;$

$\mathbf{e_{i,j,k}.2}$ $[\, (i,j)''_{k,1} \,]_{e_{i,j,k}} \to [\, (i,j)''_{k} \,]_{e_{i,j,k}} \; [\, (i,j)''_{k} \,]_{e_{i,j,k}};$

$\mathbf{e_{i,j,k}.3}$ $((i,j)''_k\,,\, out).$

(5) The input membrane is the membrane labelled by 2 and the output zone is the environment of the system (labelled by 0).

7.3.1.2 An overview of the computations

Now we briefly show how each system $\Pi(\langle n, m \rangle)$ works in order to process any directed graph with n nodes and m arcs.

We consider the ensuing polynomial encoding (cod, s) from HAM-CYCLE in $\mathbf{\Pi}$: for each instance $G = (V, E)$ of HAM-CYCLE problem, with $V = \{1, \ldots, n\}$ and $E = \{(i_1, j_1), \ldots, (i_m, j_m)\}$, we define $s(G) = \langle n, m \rangle$ and $cod(G) = \{(i,j)_k \mid (i,j) \in E, 1 \leq k \leq n\}$. The expression $(i,j)_k$ in $cod(G)$ can be interpreted as follows: arc (i,j) is "placed" in "position k" in a potential path. According to this polynomial encoding, graph G will be processed by system $\Pi(s(G))$ with input multiset $cod(G)$. In what follows, we informally describe how system $\Pi(s(G)) + cod(G)$ works. The solution is structured in the following stages:

- *Generation Stage*: All possible combinations of arcs from the input graph, including a code of their position in potential paths, are generated by using cell division in an adequate way.

- *Checking Stage*: It is checked whether or not the different combinations of arcs generated in the previous stage encode Hamiltonian cycles of the input graph.

- *Output Stage*: The system sends the right answer to the environment according to the results obtained in the previous stage.

Generation stage

At this stage, the system generates all the possible subsets of arcs of the graph (in fact, subsets of A'_G) which contain their potential positions in a path according to the notations introduced in Subsection 1.2. In this way, by applying rules of type 2.1 at configuration $\mathcal{C}_{2^{n\cdot m}}$, there will be $2^{n\cdot m}$ membranes labelled by 2 such that each of them encodes a different combination of arcs from the input graph. Simultaneously, by applying rules of types 1, 2 and 3 from $\mathcal{R}_3$, $\mathcal{R}_4$, $\mathcal{R}_5$, $\mathcal{R}_6$, $\mathcal{R}_7$, $\mathcal{R}_8$ and $\mathcal{R}_9$, $2^{n\cdot m}$ copies of objects $b', b'', b''', c', c'', c'''$ and c'''' are produced in membranes labelled by $3, 4, 5, 6, 7, 8, 9$, respectively, and 2^{n^3} copies of objects a', a'' and $(i,j)''_k$ are produced in membranes labelled by $a_{1,j}, a_{2,j}$, and $e_{i,j,k}$, respectively. The generation stage takes n^3 steps.

Checking stage

At this stage, the system checks whether or not there exists a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ containing a subset of A''_G that encodes a Hamiltonian cycle of G. This is done in 4 steps.

At step $n^3 + 1$, the contents of membranes labelled by $3, 4, 5, 6, 7, 8, 9$, $a_{1,j}$ $(1 \le j \le n), a_{2,j}$ $(1 \le j \le n)$ and $e_{i,j,k}$ $(1 \le i, j, k \le n)$ are sent to the skin membrane by applying rules 3.4, 4.4, 5.4, 6.4, 7.4, 8.4, 9.4, $a_{1,j}$.2, $a_{2,j}$.2, $e_{i,j,k}$.3. From this moment on, none of these membranes will participate in the evolution of the configurations.

At step $n^3 + 2$, objects a, b, c in membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ are replaced by objects a', b', c' from the skin membrane by applying rules 2.2, 2.3, and 2.4. Simultaneously, by applying rules 2.6, each subset of A'_G contained in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+1}$ produces the "corresponding" subset of A''_G. Besides, $\mathcal{C}_{n^3+2}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+2}(11) = \{\beta_{n^3+2}\,, \texttt{no}\}$.

At step $n^3 + 3$, by applying rules 2.3 and 2.4, objects a', b', c' in membranes labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ are replaced by objects a'', b'', c'' from the skin membrane. Simultaneously, by applying rules of type 2.7, each subset contained in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+2}$ is transformed into a subset encoding each possible path in the input graph. This way, according to Proposition 1.1, we have that the input graph (with n nodes and m arcs) has a Hamiltonian cycle if and only if at configuration $\mathcal{C}_{n^3+3}$ there exists some membrane labelled by 2 at configuration $\mathcal{C}_{n^3+3}$ such that the subset of A''_G contained in it has size equal to n. Besides, $\mathcal{C}_{n^3+3}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+3}(11) = \{\beta_{n^3+3}\,, \texttt{no}\}$.

At step $n^3 + 4$, by applying rules 2.3 and 2.4, objects b'', c'' in membranes labelled by 2 are substituted by objects b''', c''' from the skin membrane. Simultaneously, by applying rules 2.8, each object contained in the subset associated with each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+3}$ is sent to the skin membrane cooperating with an object a''. Therefore, the number of copies of object a'' appearing in a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+4}$ is equal to $n-\gamma$, where γ is the size of the path in the input graph encoded by that membrane. Then, the input graph (with n nodes and m arcs) has a Hamiltonian cycle if and only if there exists a membrane labelled by 2 at configuration $\mathcal{C}_{n^3+4}$ such that it does not contain any object a''.

At step n^3+5, by applying rules of type 2.5, objects a'' and b''' in membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ are sent to the skin membrane. Simultaneously, rule $(c''', out; c'''', in)$ produces an object c'''' in each membrane labelled by 2 at configuration $\mathcal{C}_{n^3+5}$.

Output stage

Finally, the output stage takes 4 steps. Only membranes labelled by 2 at configuration $\mathcal{C}_{n^3+5}$ containing some object b''' (i.e., membrane encoding a Hamiltonian cycle) can evolve, and only rule $(c''', out; c'''', in) \in \mathcal{R}_2$ is applicable to that membrane. In this case, an object c'''' will appear in each membrane labelled by 2 at that configuration. Besides, if a membrane with label 2 at the mentioned configuration does not encode a Hamiltonian cycle of the input graph, then it contains objects b'', so rule $(a'' b''', out) \in \mathcal{R}_2$ will be applied. That is, the input graph has a Hamiltonian cycle if and only if some object c'''' appears in the skin membrane at configuration $\mathcal{C}_{n^3+6}$. Besides, $\mathcal{C}_{n^3+6}(10) = \{\texttt{yes}\}$ and $\mathcal{C}_{n^3+6}(11) = \{\beta_{n^3+6}, \texttt{no}\}$.

If the input graph has a Hamiltonian cycle, then only rules $(\alpha_{n^3+6}\, c'''', in) \in \mathcal{R}_{10}$ and $(\beta_{n^3+6}, out; \beta_{n^3+7}, in) \in \mathcal{R}_{11}$ are applicable to configuration $\mathcal{C}_{n^3+6}$. Otherwise, only rule $(\beta_{n^3+6}\, out; \beta_{n^3+7}, in)$ is applicable to that configuration. Therefore, the answer of the problem is affirmative if and only if $\mathcal{C}_{n^3+7}(10) = \{\alpha_{n^3+6}\, c'''', \texttt{yes}\}$. Besides, in any case, $\mathcal{C}_{n^3+7}(11) = \{\beta_{n^3+7}, \texttt{no}\}$. Then, if there exists a Hamiltonian path, then rules $(c''''\, \texttt{yes}, out) \in \mathcal{R}_{10}$ and $(\beta_{n^3+7}\, \texttt{no}, out) \in \mathcal{R}_{11}$ are applicable to configuration $\mathcal{C}_{n^3+7}$. Otherwise, only rule $(\beta_{n^3+7}\, \texttt{no}, out) \in \mathcal{R}_{11}$ is applicable to that configuration. Hence, the answer of the problem is affirmative if and only if the skin membrane at configuration $\mathcal{C}_{n^3+8}$ contains object yes (together with objects $c'''', \beta_{n^3+7}, \texttt{no}$), but no object α_{n^3+6}. Otherwise, the skin membrane at configuration $\mathcal{C}_{n^3+8}$ contains objects $\beta_{n^3+7}, \texttt{no}, \alpha_{n^3+6}$, but no object yes.

At the last step, in cases when an affirmative answer results, rule $(\texttt{yes}, out)$

is applied to configuration $\mathcal{C}_{n^3+8}$, producing an object `yes` in the environment, and the computation halts. Otherwise, rule $(\mathtt{no}\,\alpha_{n^3+6}\,, \mathit{out})$ is applied to that configuration, thus producing a negative answer.

7.3.1.3 Main result

Theorem 7.4. `HAM-CYCLE` $\in \mathbf{PMC_{CDC(2)}}$.

Proof. The family of P systems with symport/antiport rules and membrane division constructed in Subsection 4.3.1.1 verifies the following:

(a) Every system of the family $\mathbf{\Pi}$ is a recognizer P system with membrane division and symport/antiport rules of length at most 2.

(b) The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines because, for each $n, m \in \mathbb{N}$, the rules of $\Pi(\langle n, m\rangle)$ of the family are recursively defined from $n, m \in \mathbb{N}$, and the amount of resources needed to build an element of the family is of a polynomial order in n, as shown below:

- Size of the alphabet: $n^6 + 12n^3 + 29 \in \Theta(n^6)$;
- Initial number of membranes: $n^3 + 2n + 11 \in \Theta(n^3)$;
- Initial number of objects: $9n^3 + 3n + 13 \in \Theta(n^3)$;
- Number of rules: $n^6 + 4n^5 + n^4 + 13n^3 + 2n + 30 \in \Theta(n^6)$;
- Maximal length of a rule: $2 \in \Theta(1)$.

(c) The pair (cod, s) of polynomial–time computable functions defined in Subsection 7.3.1.2 is a polynomial encoding from $\mathtt{HAM} - \mathtt{CYCLE}$ to $\mathbf{\Pi}$.

(d) The family $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regard to $(\mathtt{HAM\text{-}CYCLE}, \mathit{cod}, s)$ (see Subsection 7.3.1.2).

Therefore, according to Definition 2.14, the family $\mathbf{\Pi}$ from $\mathbf{CDC(2)}$ solves `HAM-CYCLE` problem in polynomial time with respect to the number of nodes.

□

Corollary 7.2. $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC_{CDC(2)}}$.

Proof. It suffices to notice that `HAM-CYCLE` problem is an **NP**-complete problem, `HAM-CYCLE`$\in \mathbf{PMC_{CDC(2)}}$, and the complexity class $\mathbf{PMC_{CDC(2)}}$ is closed under polynomial-time reduction and under complement.

□

7.3.2 Computational Efficiency of Systems in CSC(3)

The limitations on the efficiency of P systems with membrane separation whose symport/antiport rules involve at most two objects, have been established [89]. Specifically, it has been proved that only tractable problems can be efficiently solved by using families of P systems with membrane separation which make use of symport/antiport rules with length at most 2. In this Section we analyze the computational efficiency of familes of P systems from **CSC**(**3**), and it is given a polynomial time solution to `SAT` problem by means of a family of such P systems, in a uniform way, according to Definition 2.14.

7.3.2.1 A polynomial time solution to `SAT` problem in CSC(3)

We consider a family $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$ of recognizer P system from **CSC**(3), such that each system $\Pi(t)$, with $t = \langle n, m\rangle$, will process all instances of `SAT` problem (an instance is a Boolean formula φ in conjunctive normal form with n variables and m clauses) provided that the appropriate input multiset $cod(\varphi)$ is supplied to the system.

For each $n, m \in \mathbb{N}$, we consider the recognizer P system from **CSC**(3)

$$\Pi(\langle n, m\rangle) = (\Gamma, \Gamma_0, \Gamma_1, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \cdots, \mathcal{R}_q, i_{in}, i_{out})$$

defined as follows:

(1) Working alphabet:

$$\begin{aligned}\Gamma \;=\; & \Sigma \cup \mathcal{E} \cup \{\alpha_{i,0,k}, \alpha'_{i,0,k} \mid 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1\} \cup \\ & \{A_1, B_1, b_1, b'_1, c_1, c'_1, v_1, q_{1,1}, \beta_0, \beta'_0, \beta''_0, \gamma_0, \gamma'_0, \gamma''_0, \gamma'''_0, f_0, \texttt{yes}, \texttt{no}\} \cup \\ & \{f'_i \mid 0 \leq i \leq 3n+2m+1\}, \cup \{\rho_{i,0}, \tau_{i,0} \mid 1 \leq i \leq n\}, \cup \{\delta_{j,0} \mid 0 \leq j \leq m\} \cup\end{aligned}$$

where the input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\}$, and the alphabet of the environment is:

$$\begin{array}{rcl}\mathcal{E} & = & \{\alpha_{i,j,k}, \alpha'_{i,j,k} \mid 1 \leq i \leq n-1 \wedge 1 \leq j \leq 3(n-1) \wedge 0 \leq k \leq 1\} \cup \\ & & \{\beta_j, \beta'_j, \beta''_j, \gamma_j, \gamma'_j, \gamma''_j, \gamma'''_j \mid 1 \leq j \leq 3(n-1)\} \cup \\ & & \{\rho_{i,j}, \tau_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq 3n-1\} \cup \\ & & \{T_{i,j}, T'_{i,j}, F_{i,j}, F'_{i,j} \mid 1 \leq i < j \wedge 1 \leq j \leq n\} \cup \\ & & \{T_{i,i}, F'_{i,i}, T_i, F_i \mid 1 \leq i \leq n\} \cup \{A_i, A'_i, B_i, B'_i \mid 2 \leq i \leq n+1\} \cup \\ & & \{b_i, b'_i, c_i, c'_i \mid 2 \leq i \leq n\} \cup \{v_i \mid 2 \leq i \leq n-1\} \cup \\ & & \{y_i, a_i, w_i \mid 1 \leq i \leq n-1\} \cup \{z_i \mid 1 \leq i \leq n-2\} \cup \\ & & \{q_{i,j} \mid 1 \leq i \leq j \wedge 2 \leq j \leq n-1\} \cup \{u_{i,j} \mid 1 \leq i \leq j \wedge 1 \leq j \leq n-2\} \cup \\ & & \{t_{i,j}, f_{i,j}, r_{i,j}, s_{i,j} \mid 1 \leq i \leq j \wedge 1 \leq j \leq n-1\} \cup \\ & & \{d_{i,j,k}, \overline{d}_{i,j,k} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-1\} \cup \\ & & \{f_r \mid 1 \leq r \leq 3n+2m\} \cup \{e_{i,j}, \overline{e}_{i,j} \mid 1 \leq i \leq n \wedge 1 \leq j \leq m\} \cup \\ & & \{\delta_{j,r} \mid 0 \leq j \leq m \wedge 1 \leq r \leq 3n\} \cup \{E_j \mid 0 \leq j \leq m\} \cup \{S\}\end{array}$$

(2) The partition is $\{\Gamma_0, \Gamma_1\}$, where $\Gamma_0 = \Gamma \setminus \Gamma_1$ and

$$\begin{array}{rcl}\Gamma_1 & = & \{T'_{i,j}\, F'_{i,j} \mid 1 \leq i < j \wedge 1 \leq j \leq n\} \cup \{F'_{i,i} \mid 1 \leq i \leq n\} \cup \\ & & \{A'_i, B'_i \mid 2 \leq i \leq n+1\}\end{array}$$

(3) Membrane structure: $\mu = [\ [\ \ ]_2\ [\ \ ]_3]_1$. The input membrane is the membrane labelled with 1.

(4) Initial multisets:

$$\begin{array}{rcl}\mathcal{M}_1 & = & \{\alpha_{i,0,k}, \alpha'_{i,0,k} \mid 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1\} \cup \{\rho_{i,0}, \tau_{i,0} \mid 1 \leq i \leq n\} \cup \\ & & \{\beta_0, \beta'_0, \beta''_0, \gamma_0, \gamma'_0, \gamma''_0, \gamma'''_0, c_1, c'_1, b_1, b'_1, v_1, q_{1,1}, f_0, \texttt{yes}\} \cup \\ & & \{\delta_{j,0} \mid 0 \leq j \leq m\} \cup \{f'_p \mid 1 \leq p \leq 3n+2m+1\} \\ \mathcal{M}_2 & = & \{A_1, B_1\} \\ \mathcal{M}_3 & = & \{f'_0, \texttt{no}\}\end{array}$$

(5) $\boxed{\text{Rules in } \mathcal{R}_1}$:

1.1 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 1, \ldots, n-1$) the objects $T^{2^{p-1}}_{i,p+1}, T'^{2^{p-1}}_{i,p+1}, F^{2^{p-1}}_{i,p+1}, F'^{2^{p-1}}_{i,p+1}$:

$$\left.\begin{array}{l}(\alpha_{i,0,k}, out; \alpha_{i,1,k}, in) \\ (\alpha'_{i,0,k}, out; \alpha'_{i,1,k}, in) \\ (\alpha_{i,1,k}, out; \alpha_{i,2,k}, in) \\ (\alpha'_{i,1,k}, out; \alpha'_{i,2,k}, in) \\ (\alpha_{i,2,k}, out; \alpha_{i,3,k}, in) \\ (\alpha'_{i,2,k}, out; \alpha'_{i,3,k}, in)\end{array}\right\} 1 \leq i \leq n-1 \wedge 0 \leq k \leq 1$$

$$\begin{array}{rl}
(\alpha_{i,3p,k}\,, out; \alpha_{i,3p+1,k}\ \Delta^{k}_{i,p+1}\,, in): & 1 \le i \le p \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha'_{i,3p,k}\,, out; \alpha'_{i,3p+1,k}\ \Delta'^{k}_{i,p+1}\,, in): & 1 \le i \le p \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha_{i,3p,k}\,, out; \alpha_{i,3p+1,k}\,, in): & p+1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha'_{i,3p,k}\,, out; \alpha'_{i,3p+1,k}\,, in): & p+1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha_{i,3p+1,k}\,, out; \alpha_{i,3p+2,k}\,, in): & 1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha'_{i,3p+1,k}\,, out; \alpha'_{i,3p+2,k}\,, in): & 1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha_{i,3p+2,k}\,, out; \alpha^{2}_{i,3p+3,k}\,, in): & 1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha'_{i,3p+2,k}\,, out; \alpha'^{2}_{i,3p+3,k}\,, in): & 1 \le i \le n-1 \wedge 1 \le p \le n-2 \wedge 0 \le k \le 1 \\
(\alpha_{i,3(n-1),k}\,, out;\ \Delta^{k}_{i,n}\,, in): & 1 \le i \le n-1 \wedge 0 \le k \le 1 \\
(\alpha'_{i,3(n-1),k}\,, out;\ \Delta'^{k}_{i,n}\,, in): & 1 \le i \le n-1 \wedge 0 \le k \le 1
\end{array}$$

where $\Delta^0_{i,j} = F_{i,j}$, $\Delta'^0_{i,j} = F'_{i,j}$, $\Delta^1_{i,j} = T_{i,j}$, $\Delta'^1_{i,j} = T'_{i,j}$.

1.2 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 0, 1, \ldots, n-1$) the objects $B^{2^p}_{p+2}, B'^{2^p}_{p+2}, S^{2^p}$:

$$\left.\begin{array}{r}
(\beta_{3p}\,, out; \beta_{3p+1}\ B_{p+2}\,, in) \\
(\beta'_{3p}\,, out; \beta'_{3p+1}\ B'_{p+2}\,, in) \\
(\beta''_{3p}\,, out; \beta''_{3p+1}\ S\,, in) \\
(\beta_{3p+1}\,, out; \beta_{3p+2}\,, in) \\
(\beta'_{3p+1}\,, out; \beta'_{3p+2}\,, in) \\
(\beta''_{3p+1}\,, out; \beta''_{3p+2}\,, in) \\
(\beta_{3p+2}\,, out; \beta^{2}_{3p+3}\,, in) \\
(\beta'_{3p+2}\,, out; \beta'^{2}_{3p+3}\,, in) \\
(\beta''_{3p+2}\,, out; \beta''^{2}_{3p+3}\,, in)
\end{array}\right\} 0 \le p \le n-3$$

$$\left.\begin{array}{r}
(\beta_{3(n-2)}\,, out; \beta_{3(n-2)+1}\ B_{n}\,, in) \\
(\beta'_{3(n-2)}\,, out; \beta'_{3(n-2)+1}\ B'_{n}\,, in) \\
(\beta''_{3(n-2)}\,, out; \beta''_{3(n-2)+1}\ S\,, in) \\
(\beta_{3(n-2)+1}\,, out; \beta_{3(n-2)+2}\,, in) \\
(\beta'_{3(n-2)+1}\,, out; \beta'_{3(n-2)+2}\,, in) \\
(\beta''_{3(n-2)+1}\,, out; \beta''_{3(n-2)+2}\,, in) \\
(\beta_{3(n-2)+2}\,, out; \beta^{2}_{3(n-2)+3}\,, in) \\
(\beta'_{3(n-2)+2}\,, out; \beta'^{2}_{3(n-2)+3}\,, in) \\
(\beta''_{3(n-2)+2}\,, out; \beta''^{2}_{3(n-2)+3}\,, in)
\end{array}\right\}$$

$$\left.\begin{array}{r}
(\beta_{3(n-1)}\,, out; B_{n+1}\,, in) \\
(\beta'_{3(n-1)}\,, out; B'_{n+1}\,, in) \\
(\beta''_{3(n-1)}\,, out; S\,, in)
\end{array}\right\}$$

1.3 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3p+1}$ ($p = 0, 1, \dots, n-1$) the objects $T^{2^p}_{p+1,p+1}, T'^{2^p}_{p+1,p+1}, A^{2^p}_{p+2}, A'^{2^p}_{p+2}$:

$$\left.\begin{array}{r}
(\gamma_{3p}\,, out; \gamma_{3p+1}\ T_{p+1,p+1}\,, in) \\
(\gamma'_{3p}\,, out; \gamma'_{3p+1}\ F'_{p+1,p+1}\,, in) \\
(\gamma''_{3p}\,, out; \gamma''_{3p+1}\ A_{p+2}\,, in) \\
(\gamma'''_{3p}\,, out; \gamma'''_{3p+1}\ A'_{p+2}\,, in) \\
(\gamma_{3p+1}\,, out; \gamma_{3p+2}\,, in) \\
(\gamma'_{3p+1}\,, out; \gamma'_{3p+2}\,, in) \\
(\gamma''_{3p+1}\,, out; \gamma''_{3p+2}\,, in) \\
(\gamma'''_{3p+1}\,, out; \gamma'''_{3p+2}\,, in) \\
(\gamma_{3p+2}\,, out; \gamma^2_{3p+3}\,, in) \\
(\gamma'_{3p+2}\,, out; \gamma'^2_{3p+3}\,, in) \\
(\gamma''_{3p+2}\,, out; \gamma''^2_{3p+3}\,, in) \\
(\gamma'''_{3p+2}\,, out; \gamma'''^2_{3p+3}\,, in)
\end{array}\right\} 0 \le p \le n-3$$

$$\left.\begin{array}{r}
(\gamma_{3(n-2)}\,, out; \gamma_{3(n-2)+1}\ T_{n-1,n-1}\,, in) \\
(\gamma'_{3(n-2)}\,, out; \gamma'_{3(n-2)+1}\ F'_{n-1,n-1}\,, in) \\
(\gamma''_{3(n-2)}\,, out; \gamma''_{3(n-2)+1}\ A_{n}\,, in) \\
(\gamma'''_{3(n-2)}\,, out; \gamma''_{3(n-2)+1}\ A'_{n}\,, in) \\
(\gamma_{3(n-2)+1}\,, out; \gamma_{3(n-2)+2}\ in) \\
(\gamma'_{3(n-2)+1}\,, out; \gamma'_{3(n-2)+2}\ in) \\
(\gamma''_{3(n-2)+1}\,, out; \gamma''_{3(n-2)+2}\ in) \\
(\gamma'''_{3(n-2)+1}\,, out; \gamma'''_{3(n-2)+2}\ in) \\
(\gamma_{3(n-2)+2}\,, out; \gamma^2_{3(n-2)+3}\ in) \\
(\gamma'_{3(n-2)+2}\,, out; \gamma'^2_{3(n-2)+3}\ in) \\
(\gamma''_{3(n-2)+2}\,, out; \gamma''^2_{3(n-2)+3}\ in) \\
(\gamma'''_{3(n-2)+2}\,, out; \gamma'''^2_{3(n-2)+3}\ in)
\end{array}\right\}$$

$$\left.\begin{array}{r}
(\gamma_{3(n-1)}\,, out;\ T_{n,n}\,, in) \\
(\gamma'_{3(n-1)}\,, out;\ F'_{n,n}\,, in) \\
(\gamma''_{3(n-1)}\,, out;\ A_{n+1}\,, in) \\
(\gamma'''_{3(n-1)}\,, out;\ A'_{n+1}\,, in)
\end{array}\right\}$$

1.4 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3n}$ the objects $T_i^{2^{n-1}}, F_i^{2^{n-1}}$ ($1 \le i \le n$):

$$\left.\begin{array}{r}(\rho_{i,0}\,,out;\rho_{i,1}\,,in)\\(\tau_{i,0}\,,out;\tau_{i,1}\,,in)\\(\rho_{i,1}\,,out;\rho_{i,2}\,,in)\\(\tau_{i,1}\,,out;\tau_{i,2}\,,in)\\(\rho_{i,2}\,,out;\rho_{i,3}\,,in)\\(\tau_{i,2}\,,out;\tau_{i,3}\,,in)\end{array}\right\} 1 \le i \le n$$

$$\left.\begin{array}{r}(\rho_{i,3p}\,,out;\rho_{i,3p+1}\,,in)\\(\tau_{i,3p}\,,out;\tau_{i,3p+1}\,,in)\\(\rho_{i,3p+1}\,,out;\rho^2_{i,3p+2}\,,in)\\(\tau_{i,3p+1}\,,out;\tau^2_{i,3p+2}\,,in)\\(\rho_{i,3p+2}\,,out;\rho_{i,3p+3}\,,in)\\(\tau_{i,3p+2}\,,out;\tau_{i,3p+3}\,,in)\end{array}\right\} 1 \le i \le n \wedge 1 \le p \le n-2$$

$$\left.\begin{array}{r}(\rho_{i,3(n-1)}\,,out;\rho_{i,3(n-1)+1}\,,in)\\(\tau_{i,3(n-1)}\,,out;\tau_{i,3(n-1)+1}\,,in)\\(\rho_{i,3(n-1)+1}\,,out;\rho^2_{i,3(n-1)+2}\,,in)\\(\tau_{i,3(n-1)+1}\,,out;\tau^2_{i,3(n-1)+2}\,,in)\\(\rho_{i,3(n-1)+2}\,,out;T_i\,,in)\\(\tau_{i,3(n-1)+2}\,,out;F_i\,,in)\end{array}\right\} 1 \le i \le n$$

$$\left.\begin{array}{r}(A_i\,,out;a_i\,,in)\\(A'_i\,,out;a_i\,,in)\\(B_i\,,out;a_i\,,in)\\(B'_i\,,out;a_i\,,in)\end{array}\right\} 1 \le i \le n-1$$

$$\left.\begin{array}{rcl}(y_i\,,out;z_i\,w_i\,,in) & : & 1 \le i \le n-2\\(y_{n-1}\,,out;w_{n-1}\,,in) & : & \end{array}\right\}$$

$$\left.\begin{array}{rcl}(w_i\,,out;c_{i+1}\,c'_{i+1}\,,in) & : & 1 \le i \le n-1\\(z_i\,,out;v_{i+1}\,,in) & : & 1 \le i \le n-2\end{array}\right\}$$

$$\left.\begin{array}{r}(v_i\,,out;y_i^2\,,in)\\(a_i\,,out;b_{i+1}\,b'_{i+1}\,,in)\end{array}\right\} 1 \le i \le n-1$$

$$\left.\begin{array}{rcl}(q_{1,1}\,,out;r_{1,1}\,,in) & & \\(q_{i,j}\,,out;r^2_{i,j}\,,in) & : & 1 \le i \le n-1 \wedge i \le j \le n-1\end{array}\right\}$$

$$\left.\begin{array}{rcc}(r_{i,j}\,,out;s_{i,j}\,u_{i,j}\,,in) & : & 1\leq i\leq n-2\wedge i\leq j\leq n-2\\ (r_{i,n-1}\,,out;s_{i,n-1}\,,in) & : & 1\leq i\leq n-1\end{array}\right\}$$

$$(s_{i,j}\,,out;t_{i,j}\,f_{i,j}\,,in)\;:\;1\leq i\leq n-1\wedge i\leq j\leq n-1$$

$$\left.\begin{array}{rcc}(u_{1,j}\,,out;q_{1,j+1}\,q_{2,j+1}\,,in) & : & 1\leq j\leq n-2\\ (u_{i,j}\,,out;q_{i+1,j+1}\,,in) & : & 2\leq i\leq j\wedge 2\leq j\leq n-2\end{array}\right\}$$

$$\left.\begin{array}{c}(T_{i,j}\,t_{i,j},out)\\ (T'_{i,j}\,t_{i,j},out)\\ (F_{i,j}\,f_{i,j},out)\\ (F'_{i,j}\,f_{i,j},out)\end{array}\right\}1\leq i\leq j\wedge 1\leq j\leq n$$

1.5 Rules allowing that each object $x_{i,j}$ (meaning that $x_i\in C_j$) and $\overline{x}_{i,j}$ (meaning that $\neg x_i\in C_j$) results in the corresponding $e_{i,j}$ and $\overline{e}_{i,j}$ objects with multiplicity 2^{n-1} in membrane 1 of configuration $\mathcal{C}_{n+1}$.

$$\left.\begin{array}{c}(x_{i,j}\,,out;d^2_{i,j,1}\,;in)\\ (\overline{x}_{i,j}\,,out;\overline{d}^2_{i,j,1}\,;in)\end{array}\right\}1\leq i\leq n\wedge 1\leq j\leq m$$

$$\left.\begin{array}{c}(d_{i,j,k}\,,out;d^2_{i,j,k+1}\,,in)\\ (\overline{d}_{i,j,k}\,,out;\overline{d}^2_{i,j,k+1}\,,in)\end{array}\right\}1\leq i\leq n\wedge 1\leq j\leq m\wedge 1\leq k\leq n-2$$

$$\left.\begin{array}{c}(d_{i,j,n-1}\,,out;e_{i,j}\,,in)\\ (\overline{d}_{i,j,n-1}\,,out;\overline{e}_{i,j}\,,in)\end{array}\right\}1\leq i\leq n\wedge 1\leq j\leq m$$

1.6 Output rule with **affirmative** answer: $(E_0\,f_{3n+2m}\,\texttt{yes}\,;out)$.

1.7 Output rule with **negative** answer: $(f_{3n+2m}\,\texttt{no}\,;out)$.

1.8 Rules to generate in the membrane 1 of configuration $\mathcal{C}_{3n}$ the objects $E_1^{2^n}$, and in the membrane 1 of configuration $\mathcal{C}_{3n+1}$ the objects $E_0^{2^n},E_2^{2^n},\ldots,E_m^{2^n}$:

$$\left.\begin{array}{c}(\delta_{j,3p}\,,out;\delta_{j,3p+1},in)\\ (\delta_{j,3p+1}\,,out;\delta^2_{j,3p+2},in)\end{array}\right\}0\leq j\leq m\wedge 0\leq p\leq n-1$$

$$(\delta_{j,3p+2}\,, out; \delta_{j,3p+3}, in)\ 0 \le j \le m \wedge 0 \le p \le n-2$$

$$(\delta_{1,3(n-1)+2}\,, out; E_1, in)$$

$$\left.\begin{array}{r}(\delta_{j,3(n-1)+2}\,, out; \delta_{j,3(n-1)+3}, in)\\ (\delta_{j,3n}\,, out; E_j, in)\end{array}\right\} 0 \le j \le m \wedge j \neq 1$$

$$(f_p\,, out; f_{p+1}\,; in)\ 0 \le p \le 3n+2m-1$$

1.9 Rules to remove a part of the garbage:

$$\left.\begin{array}{r}(t_{i,k}\ T_{i,k}\,, out)\\ (t_{i,k}\ T'_{i,k}\,, out)\\ (f_{i,k}\ F_{i,k}\,, out)\\ (f_{i,k}\ F'_{i,k}\,, out)\end{array}\right\} 1 \le i < k \wedge 2 \le k \le n$$

$$\left.\begin{array}{r}(t_{i,i}\ T_{i,i}\,, out)\\ (f_{i,i}\ F'_{i,i}\,, out)\end{array}\right\} 1 \le i \le n$$

$$\left.\begin{array}{r}(b_k\ B_{k+1}\,, out)\\ (b'_k\ B'_{k+1}\,, out)\\ (c_k\ A_{k+1}\,, out)\\ (c'_k\ A'_{k+1}\,, out)\end{array}\right\} n-1 \le k \le n$$

(6) $\boxed{\text{Rules in } \mathcal{R}_2}$:

2.1 Separation rule: $[\,S\,]_2 \rightarrow [\,\Gamma_0\,]_2\ [\,\Gamma_1\,]_2$.

2.2 Rules to produce objects $T_{i,i}, A_{i+1}, F'_{i,i}, A'_{i+1}$ in each membrane 2:

$$\left.\begin{array}{r}(A_i, out; c_i\,c'_i, in)\\ (A'_i, out; c_i\,c'_i, in)\\ (B_i, out; b_i\,b'_i, in)\\ (B'_i, out; b_i\,b'_i, in)\\ (b_i, out; B_{i+1}\,S, in)\\ (b'_i, out; B'_{i+1}, in)\\ (c_i, out; T_{i,i}\,A_{i+1}, in)\\ (c'_i, out; F'_{i,i}\,A'_{i+1}, in)\end{array}\right\} 1 \le i \le n$$

2.3 Rules to produce an object E_1 in each membrane 2 of configuration $\mathcal{C}_{3n+1}$ and an object E_0 in each membrane 2 of configuration $\mathcal{C}_{3n+2}$:

$$\begin{array}{c}(B_{n+1}, out; E_1, in)\\(B'_{n+1}, out; E_1, in)\\(A_{n+1}, out; E_0, in)\\(A'_{n+1}, out; E_0, in)\end{array}$$

2.4 Rules to produce a truth assignment in each membrane 2 of configuration $\mathcal{C}_{3n+1}$:

$$\left.\begin{array}{c}(T_{i,j}, out; t_{i,j}, in)\\(T'_{i,j}, out; t_{i,j}, in)\\(F_{i,j}, out; f_{i,j}, in)\\(F'_{i,j}, out; f_{i,j}, in)\end{array}\right\} 1 \leq i \leq j \wedge 1 \leq j \leq n$$

$$\left.\begin{array}{c}(t_{i,j}, out; T_{i,j+1}\, T'_{i,j+1}, in)\\(f_{i,j}, out; F_{i,j+1}\, F'_{i,j+1}, in)\end{array}\right\} 1 \leq i \leq j \wedge 1 \leq j \leq n-1$$

$$\left.\begin{array}{c}(T_{i,n}, out; T_i, in)\\(T'_{i,n}, out; T_i, in)\\(F_{i,n}, out; F_i, in)\\(F'_{i,n}, out; F_i, in)\end{array}\right\} 1 \leq i \leq n$$

2.5 Rules to check clause C_j through the truth assignment encoded by a membrane 2:

$$\left.\begin{array}{c}(E_j\, T_i, out; e_{i,j}, in)\\(E_j\, F_i, out; \overline{e}_{i,j}, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m$$

2.6 Rules to restore the truth assignment encoded by a membrane 2 which makes clause C_j true:

$$\left.\begin{array}{c}(e_{i,j}, out, E_{j+1}\, T_i, in)\\(\overline{e}_{i,j}, out, E_{j+1}\, F_i, in)\end{array}\right\} 1 \leq i \leq n \wedge 1 \leq j \leq m-1$$

2.7 Rules to send an object E_0 to membrane 1 of configuration $\mathcal{C}_{3n+2m+1}$, meaning that some truth assignment encoded by a membrane labelled with 2 makes the input formula φ true:

$$\left.\begin{array}{c}(e_{i,m}\, E_0\, ; out)\\(\overline{e}_{i,m}\, E_0\, ; out)\end{array}\right\} 1 \leq i \leq n$$

(7) $\boxed{\text{Rules in } \mathcal{R}_3}$:

3.1 Rules to produce objects $f'_{3n+2m+1}$ and $\texttt{no}$ in the membrane 1 of configuration $\mathcal{C}_{3n+2m+2}$.

$$(f'_p, out; f'_{p+1}, in)\ 0 \leq p \leq 3n+2m$$
$$(f'_{3n+2m+1}\,\texttt{no}\,; out)$$

7.3.2.2 An overview of the computations

A family of recognizer P systems with symport/antiport rules and membrane separation is constructed above. For an instance of $\texttt{SAT}$ problem $\varphi = C_1 \wedge \cdots \wedge C_m$, consisting of m clauses $C_j = l_{j,1} \vee \cdots \vee l_{j,r_j}$, $1 \leq j \leq m$, where $Var(\varphi) = \{x_1, \cdots, x_n\}$, and $l_{j,k} \in \{x_i, \neg x_i \mid 1 \leq i \leq n\}$, $1 \leq j \leq m, 1 \leq k \leq r_j$. Let us assume that the number of variables, n, and the number of clauses, m, of the input formula φ, are greater or equal to 2.

The *size* mapping on the set of instances is defined as $s(\varphi) = \langle m, n \rangle$, for each $\varphi \in I_{\texttt{SAT}}$, and the encoding of the instance φ is the multiset

$$cod(\varphi) = \{x_{i,j} :\ x_i \in C_j\} \ \cup\ \{\overline{x}_{i,j} :\ \neg x_i \in C_j\}$$

That is, $x_{i,j}$ (respectively, $\overline{x}_{i,j}$) denotes variable x_i (respectively, $\neg x_i$) belonging to clause C_j. Then, the Boolean formula φ will be processed by the system $\Pi(s(\varphi))$ with input multiset $cod(\varphi)$.

Next, we informally describe how the system $\Pi(s(\varphi)) + cod(\varphi)$ works, in order to process the instance φ of $\texttt{SAT}$ problem. The solution proposed follows a brute force algorithm in the framework of recognizer P systems with symport/antiport rules and membrane separation, and it consists of the following phases:

- *Generation phase*: using separation rules, all truth assignments for the variables associated with the Boolean formula $\varphi(x_1, \ldots, x_n)$ are produced. This phase exactly takes $3n+1$ computation steps.

- *Checking phase*: checking whether or not the input formula φ is satisfied by some truth assignment generated in the previous phase. This phase takes, exactly, $3m+1$ steps, being m the number of clauses of the formula φ.

- *Output phase*: the system sends the right answer to the environment depending on the results of the previous phase. This phase takes, exactly, 1 step if the answer affirmative, and 2 steps if the answer is negative.

Generation phase

In this phase, all truth assignments for the variables associated with the Boolean formula $\varphi(x_1, \dots, x_n)$ are generated, by applying separation rules in membranes labelled with 2. This way, after completing the phase, there will exist 2^n membranes labelled with 2 such that each of them encodes a different truth assignment of the variables $\{x_1, \dots, x_n\}$.

This phase consists in a loop with n iterations and one additional final step. Each iteration of the loop takes three steps and, consequently, this phase takes $3n + 1$ steps.

To do this, in the configurations of the kind $\mathcal{C}_{3p+2}$ $(0 \leq p \leq n-1)$ there exist 2^p membranes labelled with 2 containing objects

$$A_{p+2},\ A'_{p+2},\ B_{p+2},\ B'_{p+2},\ T_{p+1,p+1},\ F'_{p+1,p+1},\ S$$

along with $2p$–tuples of objects $(\pi_{1,p+1}, \pi'_{1,p+1}, \dots, \pi_{p,p+1}, \pi'_{p,p+1})$, with $\pi \in \{T, F\}$, in such a way that the corresponding tuples are all different in the different membranes.

Thus, a separation rule can be applied to each membrane labelled with 2. As a consequence, in configuration $\mathcal{C}_{3p+3}$ $(0 \leq p \leq n-2)$ there will exist 2^{p+1} membranes labelled with 2. 2^p of them will contain objects A_{p+2} and B_{p+2}, as well as $(p+1)$–tuples $(\pi_{1,p+1}, \dots, \pi_{p+1,p+1})$, with $\pi \in \{T, F\}$, in such a way that $\pi_{p+1,p+1} = T_{p+1,p+1}$, and the corresponding tuples of these membranes are all different. The other 2^p membranes labelled with 2 contain the objects A'_{p+2} and B'_{p+2}, as well as $(p+1)$–tuples $(\pi'_{1,p+1}, \dots, \pi'_{p+1,p+1})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{p+1,p+1} = F'_{p+1,p+1}$ and the corresponding tuples of these membranes are all different.

Finally, in configuration $\mathcal{C}_{3n}$ there exist 2^n membranes labelled with 2. 2^{n-1} of them contain the objects A_{n+1} and B_{n+1}, as well as n–tuples $(\pi_{1,n}, \dots, \pi_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi_{n,n} = T_{n,n}$ and the corresponding tuples of these membranes are all different. The other 2^{n-1} membranes labelled with 2 contain the objects A'_{n+1} and B'_{n+1}, as well as n–tuples $(\pi'_{1,n}, \dots, \pi'_{n,n})$ with $\pi \in \{T, F\}$, in such a way that $\pi'_{n,n} = F'_{n,n}$ and the corresponding tuples of these membranes are all different.

This phase ends in the step $3n+1$, where configuration $\mathcal{C}_{3n+1}$ contains 2^n membranes labelled with 2, each one of them containing the objects A_{n+1} and E_1, as well as n–tuples $(\pi_1, \dots, \pi_n)$ with $\pi \in \{T, F\}$, and the corresponding tuples of these membranes are all different.

Simultaneously, during the generation phase, from the input multiset placed initially in the skin membrane, 2^{n-1} copies of each object of that multiset are generated in that membrane, corresponding to configuration $\mathcal{C}_n$. Due to technical reasons, we will change variables $x_{i,j}$ and $\overline{x}_{i,j}$ by $e_{i,j}$ and $\overline{e}_{i,j}$, respectively. This is accomplished by using the following rules from $\mathcal{R}_1$:

$$\left.\begin{array}{r}
(x_{i,j}\,, out; d^2_{i,j,1}\,; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(\overline{x}_{i,j}\,, out; \overline{d}^2_{i,j,1}\,; in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(d_{i,j,k}\,, out; d^2_{i,j,k+1}\,, in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2 \\
(\overline{d}_{i,j,k}\,, out; \overline{d}^2_{i,j,k+1}\,, in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \wedge 1 \leq k \leq n-2 \\
(d_{i,j,n-1}\,, out; e_{i,j}\,, in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m \\
(\overline{d}_{i,j,n-1}\,, out; \overline{e}_{i,j}\,, in) \ : \ 1 \leq i \leq n \wedge 1 \leq j \leq m
\end{array}\right\}$$

The cited multiset that codifies the input formula will be denoted by $(cod(\varphi))_e^{2^{n-1}}$.

Checking phase

This phase begins at computation step $3n+2$ and consists in a loop with m iterations, taking each of them 2 steps. Hence, the checking phase takes $2m$ steps.

In the configuration $\mathcal{C}_{3n+1}$, the presence of an object E_1 in each membrane labelled with 2, along with the code of a truth assignment, marks the beginning of this phase. In the first iteration of the loop, the truth assignments making clause C_1 of φ true are found. To do this, the following rules of $\mathcal{R}_2$ are applied:

$$\left.\begin{array}{l}
(E_1\, T_i, out; e_{i,1}, in) \\
(E_1\, F_i, out; \overline{e}_{i,1}, in)
\end{array}\right\} 1 \leq i \leq n$$

Simultaneously, in the computation step $(3n+1)+2$, the object E_0 is incorporated to each of the membranes labelled with 2 by means of the application of the following rules of $\mathcal{R}_2$: (A_{n+1}, out, E_0, in) and (A'_{n+1}, out, E_0, in).

At this point, the presence of an object $e_{i,1}$ or an object $\overline{e}_{i,1}$ in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+1}$ indicates that this membrane codifies a truth assignment making the first clause true.

In the next computation step, those membranes will incorporate an object E_2 coming from the skin by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}(e_{i,1}, out, E_2\,T_i, in)\\(\overline{e}_{i,1}, out, E_2\,F_i, in)\end{array}\right\} 1 \le i \le n$$

This way, the presence of an object E_2 in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+2}$ indicates that this membrane codifies a truth assignment making true the first clause and that is ready to check the second clause of the formula. That is, from this moment, the membranes labelled with 2 not making true the first clause will not evolve.

In the j-th iteration $(2 \le j \le m)$ of the aforementioned loop, the truth assignments making true the clause C_j of the formula are checked, taking into account that only the truth assignments containing the object E_j will be checked, since only these membranes make clauses $C_1, \dots, C_{j-1}$ of φ true. This is accomplished by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}(E_j\,T_i, out; e_{i,j}, in)\\(E_j\,F_i, out; \overline{e}_{i,j}, in)\end{array}\right\} 1 \le i \le n \wedge 1 \le j \le m$$

Then, the presence of an object $e_{i,j}$ or an object $\overline{e}_{i,j}$ in a membrane 2 of the configuration $\mathcal{C}_{(3n+1)+2\cdot(j-1)+1}$ indicates that this membrane codifies a truth assignment making clauses $C_1, \dots, C_j$ of φ true. Following this, those membranes will incorporate an object E_{j+1} coming from the skin by applying the following rules from $\mathcal{R}_2$:

$$\left.\begin{array}{l}(e_{i,j}, out, E_{j+1}\,T_i, in)\\(\overline{e}_{i,j}, out, E_{j+1}\,F_i, in)\end{array}\right\} 1 \le i \le n \wedge 1 \le j \le m-1$$

If the input formula φ is satisfiable, then in some membrane labelled with 2 of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ there will exist an object $e_{i,m}$ or an object $\overline{e}_{i,m}$. This indicates that the truth assignment that this membrane codifies makes true all the clauses from φ and, consequently, makes true the input formula. In this case, the checking phase ends up by applying a rule from $\mathcal{R}_2$ of the kind $(e_{i,m}\,E_0\,; out)$ or $(\overline{e}_{i,m}\,E_0\,; out)$ making an object E_0 go to the skin membrane of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+2}$, where also the object f_{3n+2m} has been produced.

If the input formula φ is not satisfiable, the no membrane labelled with 2 of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$ contains an object $e_{i,m}$ neither an object $\overline{e}_{i,m}$. In this case, the checking phase ends up by applying the rule

$(f'_{3n+2m}, out; f'_{3n+2m+1}, in) \in \mathcal{R}_3$ (in fact, this is the only rule applicable to the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1}$).

The checking phase ends at step $(3n + 1) + 2(m - 1) + 2 = 3n + 2m + 1$.

Output phase

If the input formula φ is satisfiable, then objects E_0 and f_{3n+2m} will appear in the input membrane of the configuration $\mathcal{C}_{3n+2m+1}$. Then, by applying the rule $(E_0\, f_{3n+2m}\, \texttt{yes}\, ; out)$ in the skin membrane, the object $\texttt{yes}$ is released into the environment, providing and affirmative answer at computation step $(3n + 1) + 2m + 1 = 3n + 2m + 2$.

If the input formula φ is not satisfiable, then objects f_{3n+2m} and $\texttt{yes}$ are present in the skin membrane of the configuration $\mathcal{C}_{(3n+1)+2(m-1)+1} = \mathcal{C}_{3n+2m}$, but not the object E_0. In this case, the only applicable rule in the system is $(f'_{3n+2m}, out; f'_{3n+2m+1}, in)$ in the membrane 3 and in the next computation step only the rule $(f'_{3n+2m+1}\, \texttt{no}\, ; out) \in \mathcal{R}_3$ is applicable. Consequently, objects $f_{3n+2m}, \texttt{yes}, f'_{3n+2m+1}$ and $\texttt{no}$ appear in the skin membrane of the configuration $\mathcal{C}_{3n+2m+2}$. Then, by applying the rule $(f_{3n+2m}\, \texttt{no}\, ; out)$ in the skin membrane, an object $\texttt{no}$ will be released into the environment, providing a negative answer in the step $3n + 2m + 3$.

Hence, the output phase takes 1 computation step in the case of an affirmative answer, and 2 computation steps in the case of a negative answer.

7.3.2.3 Main result

Theorem 7.5. $\texttt{SAT} \in \mathbf{PMC_{CSC(3)}}$.

Proof. The family of P systems with symport/antiport rules and membrane division constructed in Subsection 4.3.2.1 verifies the following:

(a) Every system of the family $\mathbf{\Pi}$ is a recognizer P system with membrane separation and symport/antiport rules of length at most 3.

(b) The family $\mathbf{\Pi}$ is polynomially uniform by Turing machines because, for each $n, m \in \mathbb{N}$, the rules of $\Pi(\langle n, m\rangle)$ of the family are recursively defined from $n, m \in \mathbb{N}$, and the amount of resources needed to build an element of the family is of a polynomial order in n, as shown below:

- The size of the working alphabet is of the order $\Theta(n^2 \cdot m)$.
- The initial number of cells is $3 \in \Theta(1)$.

- The initial number of objects in membranes is $9n + 3m + 17 \in \Theta(\max\{n, m\})$.
- The total number of rules is of order $\Theta(n^2 \cdot m)$.
- The maximum length of a rule is $3 \in \Theta(1)$.

(c) The pair (cod, s) of polynomial–time computable functions defined in Subsection 7.3.2.2 is a polynomial encoding from `SAT` to $\mathbf{\Pi}$.

(d) The family $\mathbf{\Pi}$ is polynomially bounded, sound and complete with regard to $(\texttt{SAT}, cod, s)$ (see Subsection 7.3.2.2).

Therefore, according to Definition 2.14, the family $\mathbf{\Pi}$ from $\mathbf{CSC(3)}$ solves `SAT` problem in polynomial time with respect to the maximum of number of variables and number of clauses.

□

Corollary 7.3. $\mathbf{NP} \cup \mathbf{co\text{-}NP} \subseteq \mathbf{PMC_{CSC(3)}}$.

Proof. It suffices to notice that `SAT` problem is an **NP**-complete problem, `SAT`$\in$ $\mathbf{PMC_{CSC(3)}}$, and the complexity class $\mathbf{PMC_{CSC(3)}}$ is closed under polynomial-time reduction and under complement.

□

7.3.3 P systems with symport/antiport rules and with cell division but without environment

In this subsection we show that the environment play an irrelevant role with respect the complexity classes associated with P systems with symport/antiport rules and with cell division. That is, the following holds: for each $k \in \mathbb{N}$ we have $\mathbf{PMC_{CDC(\mathit{k}+1)}} = \mathbf{PMC_{\widehat{CDC}(\mathit{k}+1)}}$.

Firs, we prove two technical results concerning recognizer P systems.

Proposition 7.3. *Let* $\Pi = (\Gamma, \mathcal{E}, \Sigma, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$ *be a recognizer P systems with symport/antiport rules with length at most* k, $k \geq 2$, *and without membrane division. Let* $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$ *and let* $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_m)$ *be a computation of* Π *Then,* $|\mathcal{C}_0^*| = M$, *and for each* t, $0 \leq t < m$, *we have* $|\mathcal{C}_{t+1}^*| \leq |\mathcal{C}_t^*| \cdot k$, *and* $|\mathcal{C}_{t+1}^*| \leq M \cdot k^t$.

Proof: Obviously, $|\mathcal{C}_0^*| = |\mathcal{C}_0(0) + \mathcal{C}_0(1) + \cdots + \mathcal{C}_0(q)| = |\mathcal{M}_1 + \cdots + \mathcal{M}_q| = M$. Let t, $0 \leq t < m$, and let us compute $\mathcal{C}_{t+1}^* = \mathcal{C}_{t+1}(0) + \mathcal{C}_{t+1}(1) + \cdots + \mathcal{C}_{t+1}(q)$.

Bearing in mind that only the skin membrane can send and receive objects from the environment, we have

$$\mathcal{C}_{t+1}(0) + \mathcal{C}_{t+1}(2) + \mathcal{C}_{t+1}(3) + \cdots + \mathcal{C}_{t+1}(q) \subseteq \mathcal{C}_t(0) + \mathcal{C}_t(1) + \cdots + \mathcal{C}_t(q)$$

Next, let us see what are the objects that membrane 1 can receive at step $t+1$.

- On the one hand, can receive objects from $\mathcal{C}_t(0)$.
- On the one hand, can receive objects from $\mathcal{E}$ by means of rules in the skin membrane of the type:
  - $(a\ e_{i_1} \ldots\ e_{i_r}, in)$ with $a \in \mathcal{C}_t(0)$ and $e_{i_1}, \ldots, e_{i_r} \in \mathcal{E}$, $r \leq k-1$.
  - $(a, out; e_{i_1} \ldots\ e_{i_r}, in)$ with $a \in \mathcal{C}_t(1)$ and $e_{i_1}, \ldots, e_{i_r} \in \mathcal{E}$, $r \leq k-1$.

Then, $|\mathcal{C}_{t+1}(1)| \leq |\mathcal{C}_t(0) + \mathcal{C}_t(1)| \cdot (k-1)$. So, we have

$$\begin{array}{rcl} |\mathcal{C}^*_{t+1}| & = & |\mathcal{C}_{t+1}(0) + \mathcal{C}_{t+1}(2) + \mathcal{C}_{t+1}(3) + \cdots + \mathcal{C}_{t+1}(q)| + |\mathcal{C}_{t+1}(1)| \\ & \leq & |\mathcal{C}_t(0) + \mathcal{C}_t(1) + \cdots + \mathcal{C}_t(q)| + |\mathcal{C}_t(0) + \mathcal{C}_t(1)| \cdot (k-1) \\ & \leq & |\mathcal{C}^*_t| + |\mathcal{C}^*_t| \cdot (k-1) \leq |\mathcal{C}^*_t| \cdot k \end{array}$$

Finally, let us see that $|\mathcal{C}^*_{t+1}| \leq M \cdot k^t$ by induction on t. For $t = 1$ the result is trivial because of $|\mathcal{C}^*_1| \leq (|\mathcal{C}^*_0| + M) \cdot (k-1) = 2M \cdot (k-1)$.

Let t be such that $1 < t < m$ and the result holds for t. Then,

$$|\mathcal{C}^*_{t+1}| \leq |\mathcal{C}^*_t| \cdot k \overset{h.i}{\leq} M \cdot k^{t-1} \cdot k \leq M \cdot k^t$$

$\square$

Proposition 7.4. *Let* $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ *a family of recognizer P systems from* $\mathbf{CDC}(k)$*, where* $k \geq 2$*, solving a decision problem* $X = (I_X, \theta_X)$ *in polynomial time according to Definition 2.14. Let* (cod, s) *be a polynomial encoding associated with that solution. There exists a polynomial function* $r(n)$ *such that for each instance* $u \in I_X$*,* $2^{r(|u|)}$ *is an upper bound of the number of objects in all membranes of the system* $\Pi(s(u)) + cod(u)$ *along any computation.*

Proof: Let $p(n)$ be a polynomial function such that for each $u \in I_X$ every computation of $\Pi(s(u)) + cod(u)$ is halting and it performs at most $p(|u|)$ steps.

Let $u \in I_X$ be an instance of X and

$$\Pi(s(u)) + cod(u) = (\Gamma, \mathcal{E}, \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}_1, \ldots, \mathcal{R}_q, i_{in}, i_{out})$$

Let $M = |\mathcal{M}_1 + \cdots + \mathcal{M}_q|$. Let $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_m)$, $0 \leq m \leq p(|u|)$, be a computation of Π.

First, let us suppose that we apply only communication rules at m consecutive transition steps. From Proposition 7.3 we deduce that $|\mathcal{C}_0^*| = M$ and $|\mathcal{C}_{t+1}^*| \leq M \cdot k^t$, for each t, $0 \leq t < m$.

Thus, if we apply in a consecutive way the maximum possible number of communication rules (without applying any division rules) to the system $\Pi(s(u)) + cod(u)$, in any instant of any computation of the system, $M \cdot k^{p(|u|)}$ is an upper bound of the number of objects in the whole system.

Now, let us consider the effect of applying in a consecutive way the maximum possible number of division rules (without applying any communication rules) to the system $\Pi(s(u)) + cod(u)$ when the initial configuration has $M \cdot k^{p(|u|)}$ objects. After that, an upper bound of the number of objects in the whole system by any computation is $M \cdot k^{p(|u|)} \cdot 2^{p(|u|)} \cdot p(|u|)$. Then, we consider a polynomial function $r(n)$ such that $r(|u|) \geq \log(M) + p(|u|) \cdot (1 + \log\, k) + \log(p(|u|))$, for each instance $u \in I_X$. The polynomial function $r(n)$ fulfills the property required.

$\square$

Corollary 7.4. *Let* $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ *a family of recognizer P systems with symport/antiport rules and membrane division, solving a decision problem* $X = (I_X, \theta_X)$ *in polynomial time according to Definition 2.14. Let* (cod, s) *a polynomial encoding associated with that solution. Then, there exists a polynomial function* $r(n)$ *such that for each instance* $u \in I_X$, $2^{r(|u|)}$ *is an upper bound of the number of objects from* $\mathcal{E}$ *which are moved from the environment to all membranes of the system* $\Pi(s(u)) + cod(u)$ *along any computation.*

Proof: It suffices to note that from Proposition 7.4 there exists a polynomial function $r(n)$ such that for each instance $u \in I_X$, $2^{r(|u|)}$ is an upper bound of the number of objects in all membranes of the system $\Pi(s(u)) + cod(u)$.

$\square$

7.3.3.1 Simulating systems from $\mathbf{CDC}(k)$ by means of systems from $\widehat{\mathbf{CDC}}(k)$

The goal of this section is to show that any recognizer P system with symport/antiport rules and membrane division can be simulated by a recognizer P system symport/antiport rules, membrane division and without environment, in an efficient way.

First of all, we define the meaning of efficient simulations in the framework of recognizer P systems with symport/antiport rules.

Definition 7.5. *Let* Π *and* Π' *be recognizer P systems with symport/antiport rules. We say that* Π' simulates Π in an efficient way *if the following holds:*

1. Π' *can be constructed from* Π *by a deterministic Turing machine working in polynomial time.*
2. *There exists an injective function,* f*, from the set* $\mathbf{Comp}(\Pi)$ *of computations of* Π *onto the set* $\mathbf{Comp}(\Pi')$ *of computations of* Π' *such that:*
   - ⋆ *There exists a deterministic Turing machine that constructs computation* $f(\mathcal{C})$ *from computation* $\mathcal{C}$ *in polynomial time.*
   - ⋆ *A computation* $\mathcal{C} \in \mathbf{Comp}(\Pi)$ *is an accepting computation if and only if* $f(\mathcal{C}) \in \mathbf{Comp}(\Pi')$ *is an accepting one.*
   - ⋆ *There exists a polynomial function* $p(n)$ *such that for each* $\mathcal{C} \in \mathbf{Comp}(\Pi)$ *we have* $|f(\mathcal{C})| \leq p(|\mathcal{C}|)$*.*

Now, for every family of recognizer P system with symport/antiport rules and membrane division solving a decision problem, we design a family of recognizer P systems with symport/antiport rules, membrane division and *without environment* efficiently simulating it, according to Definition 7.5.

In what follows throghout this Section, let $\mathbf{\Pi} = \{\Pi(n) \mid n \in \mathbb{N}\}$ a family of recognizer P systems with symport/antiport rules and membrane division solving a decision problem $X = (I_X, \theta_X)$ in polynomial time according to Definition 2.14, and let $r(n)$ be a polynomial function such that for each instance $u \in I_X$, $2^{r(|u|)}$ is an upper bound of the number of objects from $\mathcal{E}$ which are moved from the environment to all membranes of the system by any computation of $\Pi(s(u)) + cod(u)$.

Definition 7.6. *For each* $n \in \mathbb{N}$*, let*

$$\Pi(n) = (\Gamma, \mathcal{E}, \Sigma, \mu, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_1, \dots, \mathcal{R}_q, i_{in}, i_{out})$$

an element of the previous family, and for the sake of simplicity we denote r *instead of* $r(n)$ *and* 1 *is the label of the skin membrane. Let us consider the recognizer P system with symport/antiport rules of degree* $q_1 = 1+q\cdot(r+2)+|\mathcal{E}|$*, with membrane division and without environment*

$$\mathbf{S}(\Pi(n)) = (\Gamma', \Sigma', \mu', \mathcal{M}'_0, \mathcal{M}'_1, \dots, \mathcal{M}'_{q_1}, \mathcal{R}'_0, \mathcal{R}'_1, \dots, \mathcal{R}'_{q_1}, i'_{in}, i'_{out})$$

defined as follows:

- $\Gamma' = \Gamma \ \cup \ \{\alpha_i : \ 0 \leq i \leq r-1\}$.

- $\Sigma' = \Sigma$.
- *Each membrane* $i \in \{1, \dots, q\}$ *of* Π *provides a membrane of* $\mathbf{S}(\Pi(n))$ *with the same label. In addition,* $\mathbf{S}(\Pi(n))$ *has:*
  - ⋆ $r+1$ *new membranes, labelled by* $(i,0), (i,1), \dots, (i,r)$*, respectively, for each* $i \in \{1, \dots, q\}$.
  - ⋆ *A distinguished membrane labelled by 0.*
  - ⋆ *A new membrane, labelled by* l_b*, for each* $b \in \mathcal{E}$.
- μ' *is the rooted tree obtained from* μ *as follows:*
  - ⋆ *Membrane* 0 *is the root of* μ' *and it is the father of the root of* μ.
  - ⋆ *For each* $b \in \mathcal{E}$*, membrane* 0 *is the father of membrane* l_b.
  - ⋆ *We consider a linear structure whose nodes are* $(i,0), (i,1), \dots, (i,r)$ *and such that* (i,j) *is the father of* $(i, j-1)$*, for each* $1 \le i \le q$ *and* $1 \le j \le r$.
  - ⋆ *For each membrane* i *of* μ *we add the previous linear structure being membrane* i *the father of membrane* (i,r).
- *Initial multisets:* $\mathcal{M}'_0 = \emptyset$, $\mathcal{M}'_{l_b} = \{\alpha_0\}$*, for each* $b \in \mathcal{E}$*, and*

$$\left.\begin{array}{lcl} \mathcal{M}'_{(i,0)} & = & \mathcal{M}_i \\ \mathcal{M}'_{(i,1)} & = & \emptyset \\ \dots\dots & & \dots \\ \mathcal{M}'_{(i,r)} & = & \emptyset \\ \mathcal{M}'_i & = & \emptyset \end{array}\right\} (1 \le i \le q)$$

- *Set of rules:*

$$\mathcal{R}'_0 \cup \mathcal{R'}_1 \cup \dots \cup \mathcal{R'}_q \cup \{\mathcal{R}'_{(i,j)} : 1 \le i \le q, 0 \le j \le r\} \cup \{\mathcal{R}'_{l_b} : b \in \mathcal{E}\}$$

where $\mathcal{R}'_0 = \emptyset$, $\mathcal{R}'_i = \mathcal{R}_i$ *for* $1 \le i \le q$*, and*

$$\begin{array}{lcl} \mathcal{R}'_{(i,j)} & = & \{(a, out; \lambda, in) : \ a \in \Gamma\}, \ for\ 1 \le i \le q \ \wedge \ 0 \le j \le r\} \\ \mathcal{R}'_{l_b} & = & \{[\alpha_j]_{l_b} \to [\alpha_{j+1}]_{l_b}\ [\alpha_{j+1}]_{l_b} : \ 0 \le j \le r-2\} \ \cup \\ & & \{[\alpha_{r-1}]_{l_b} \to [b]_{l_b}\ [b]_{l_b}, (l_b, out; \lambda, in)\}, \ for\ b \in \mathcal{E} \end{array}$$

- $i'_{in} = (i_{in}, 0)$, *and* $i'_{out} = 0$.

Let us notice that $\mathbf{S}(\Pi(n))$ can be considered as an **extension** of $\Pi(n)$ **without environment**, in the following sense:

- $\star$ $\Gamma \subseteq \Gamma', \Sigma \subseteq \Sigma'$ and $\mathcal{E} = \emptyset$.
- $\star$ Each membrane in Π is also a membrane in $\mathbf{S}(\Pi(n))$.
- $\star$ There is a distinguished membrane in $\mathbf{S}(\Pi(n))$ labelled by 0 which plays the role of environment of $\Pi(n)$.
- $\star$ μ is a subtree of μ'.
- $\star$ $\mathcal{R} \subseteq \mathcal{R}'$, and now 0 is the label of a "ordinary membrane" in $\mathbf{S}(\Pi(n))$.

Next, we analyze the structure of the computations of system $\mathbf{S}(\Pi(n))$ and we compare them with the computations of $\Pi(n)$.

Lemma 7.2. *Let $\mathcal{C}' = (\mathcal{C}'_0, \mathcal{C}'_1, \dots)$ be a computation of $\mathbf{S}(\Pi(n))$. For each t $(1 \leq t \leq r)$ the following holds:*

- *$\mathcal{C}'_t(i) = \emptyset$, for $0 \leq i \leq q$.*
- *For each $1 \leq i \leq q$, and $0 \leq j \leq r$ we have:*

$$\mathcal{C}'_t(i,j) = \begin{cases} \mathcal{M}_i, & \textit{if} \quad j = t \\ \emptyset, & \textit{if} \quad j \neq t \end{cases}$$

- *For each $b \in \mathcal{E}$, there exist 2^t membranes labelled by l_b whose father is membrane 0 and their content is:*

$$\mathcal{C}'_t(l_b) = \begin{cases} \{\alpha_t\}, & \textit{if} \quad 1 \leq t \leq r-1 \\ \{b\}, & \textit{if} \quad t = r \end{cases}$$

Proof: By induction on t.

Let us start with the basic case $t = 1$. The initial configuration of system $\mathbf{S}(\Pi(n))$ is the following:

- $\mathcal{C}'_0(i) = \emptyset$, for $0 \leq i \leq q$.
- For each $1 \leq i \leq q$ we have $\mathcal{C}'_0(i,0) = \mathcal{M}_i$, and $\mathcal{C}'_0(i,j) = \emptyset$, for $1 \leq j \leq r$.
- For each $b \in \mathcal{E}$, there exists only one membrane labelled by l_b whose contents is $\{\alpha_0\}$.

At configuration $\mathcal{C}'_0$, only the following rules are applicable:

- $[\alpha_0]_{l_b} \to [\alpha_1]_{l_b}\ [\alpha_1]_{l_b}$, for each $b \in \mathcal{E}$.
- $\big(a, out; \lambda, in\big) \in \mathcal{R}_{(i,0)}$, for each $a \in supp(\mathcal{M}_i)$.

Thus,

(a) For each i $(1 \leq i \leq q)$ we have:

$$\begin{cases} \mathcal{C}'_1(i) & = \emptyset \\ \mathcal{C}'_1(0) & = \emptyset \\ \mathcal{C}'_1(i,0) & = \emptyset \\ \mathcal{C}'_1(i,1) & = \mathcal{M}_i \\ \mathcal{C}'_1(i,j) & = \emptyset, \text{ for } 2 \leq j \leq r \end{cases}$$

(b) For each $b \in \mathcal{E}$, there are 2 membranes labelled by l_b whose father is membrane 0 and their content is $\{\alpha_1\}$.

Hence, the result holds for $t = 1$.

By induction hypothesis, let t be such that $1 \leq t < r$, and let us suppose the result holds for t, that is,

- $\mathcal{C}'_t(i) = \emptyset$, for $0 \leq i \leq q$.
- For each $1 \leq i \leq q$, and $0 \leq j \leq r$ we have:

$$\mathcal{C}'_t(i,j) = \begin{cases} \mathcal{M}_i, & \text{if} \quad j = t \\ \emptyset, & \text{if} \quad j \neq t \end{cases}$$

- For each $b \in \mathcal{E}$, there exist 2^t membranes labelled by l_b whose father is membrane 0 and their content is $\mathcal{C}'_t(l_b) = \{\alpha_t\}$ (because $t \leq r - 1$).

Then, at configuration $\mathcal{C}'_t$ only the following rules are applicable:

(1) If $t \leq r - 2$, the rules $[\alpha_t]_{l_b} \to [\alpha_{t+1}]_{l_b}\ [\alpha_{t+1}]_{l_b}$, for each $b \in \mathcal{E}$.

(2) If $t = r - 1$, the rules $[\alpha_t]_{l_b} \to [b]_{l_b}\ [b]_{l_b}$, for each $b \in \mathcal{E}$.

(3) $\big(a, out; \lambda, in\big) \in \mathcal{R}_{(i,t)}$, for each $a \in supp(\mathcal{M}_i)$.

From the application of rules of types (1) or (2) at configuration $\mathcal{C}'_t$, we deduce that there are 2^{t+1} membranes labelled by l_b in $\mathcal{C}'_{t+1}$, for each $b \in \mathcal{E}$, whose father is membrane 0 and their content is $\{\alpha_{t+1}\}$, if $t \leq r-2$, or $\{b\}$, if $t = r-1$.

From the application of rules of type (3) at configuration $\mathcal{C}'_t$, we deduce that

$$\mathcal{C}'_{t+1}(i,j) = \begin{cases} \mathcal{M}_i, & \text{if} \quad j = t+1 \\ \emptyset, & \text{if} \quad 0 \leq j \leq r \ \wedge \ j \neq t+1 \end{cases}$$

Bearing in mind that no other rule of system $\mathbf{S}(\Pi(n))$ is applicable, we deduce that $\mathcal{C}'_{t+1}(i) = \emptyset$, for $0 \leq i \leq q$.

This completes the proof of this Lemma.

□

Lemma 7.3. *Let $\mathcal{C}' = (\mathcal{C}'_0, \mathcal{C}'_1, \dots)$ be a computation of the P system $\mathbf{S}(\Pi(n))$. Configuration $\mathcal{C}'_{r+1}$ is the following:*

(1) $\mathcal{C}'_{r+1}(0) = b_1^{2^r} \dots b_\alpha^{2^r}$, where $\mathcal{E} = \{b_1, \dots, b_\alpha\}$.

(2) $\mathcal{C}'_{r+1}(i) = \mathcal{M}_i = \mathcal{C}_0(i)$, for $1 \leq i \leq q$.

(3) $\mathcal{C}'_{r+1}(i,j) = \emptyset$, for $1 \leq i \leq q$, $0 \leq j \leq r$.

(4) For each $b \in \mathcal{E}$, there exist 2^r membranes labelled by l_b whose father is membrane 0 and their content is empty.

Proof: From Lemma 7.2, the configuration $\mathcal{C}'_r$ is the following:

- $\mathcal{C}'_r(i) = \emptyset$, for $0 \leq i \leq q$.
- For each i $(1 \leq i \leq q)$ we have

$$\mathcal{C}'_r(i,j) = \begin{cases} \mathcal{M}_i, & \text{if} \quad j = r \\ \emptyset, & \text{if} \quad j \neq r \end{cases}$$

- For each $b \in \mathcal{E}$, there exist 2^r membranes labelled by l_b whose father is membrane 0 and their content is $\{b\}$.

At configuration $\mathcal{C}'_r$ only the following rules are applicables:

- $\big(a, out; \lambda, in\big) \in \mathcal{R}_{(i,r)}$, for each $a \in \Gamma \cap supp(\mathcal{M}_i)$.
- $\big(b, out; \lambda, in\big) \in \mathcal{R}_{l_b}$, for each $b \in \mathcal{E}$.

Thus,

- $\mathcal{C}'_{r+1}(0) = b_1^{2^r} \dots b_\alpha^{2^r}$, where $\mathcal{E} = \{b_1, \dots, b_\alpha\}$.
- $\mathcal{C}'_{r+1}(i) = \mathcal{M}_i = \mathcal{C}_0(i)$, for $1 \leq i \leq q$.
- $\mathcal{C}'_{r+1}(i,j) = \emptyset$, for $1 \leq i \leq q$ and $0 \leq j \leq r$.
- For each $b \in \mathcal{E}$, there exist 2^r membranes labelled by l_b whose father is membrane 0 and their content is empty.

$\square$

Definition 7.7. *Let $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_m)$ be a halting computation of $\Pi(n)$. Then we define the computation $\mathbf{S}(\mathcal{C}) = (\mathcal{C}'_0, \mathcal{C}'_1, \dots, \mathcal{C}'_r, \mathcal{C}'_{r+1}, \dots, \mathcal{C}'_{r+1+m})$ of $\mathbf{S}(\Pi(n))$ as follows:*

(1) *The initial configuration is:*

$$\begin{cases} \mathcal{C}'_0(i) & = \emptyset, \textit{ for } 0 \leq i \leq q \\ \mathcal{C}'_0(i,0) & = \mathcal{C}_0(i), \textit{ for } 1 \leq i \leq q \\ \mathcal{C}'_0(i,j) & = \emptyset, \textit{ for } 1 \leq i \leq q \textit{ and } 1 \leq j \leq r \\ \mathcal{C}'_0(l_b) & = \alpha_0, \textit{ for each } b \in \mathcal{E} \end{cases}$$

(2) *The configuration $\mathcal{C}'_t$, for $1 \leq t \leq r$, is described by Lemma 7.2.*

(3) *The configuration $\mathcal{C}'_{r+1}$ is described by Lemma 7.3.*

(4) *The configuration $\mathcal{C}'_{r+1+s}$, for $0 \leq s \leq m$, coincides with the configuration $\mathcal{C}_s$ of Π, that is, $\mathcal{C}_s(i) = \mathcal{C}'_{r+1+s}(i)$, for $1 \leq i \leq q$. The content of the remaining membranes (excluding membrane 0) at configuration $\mathcal{C}'_{r+1+s}$ is equal to the content of that membrane at configuration $\mathcal{C}'_{r+1}$, that is, these membranes do not evolve after step $r+1$.*

That is, every computation $\mathcal{C}$ of $\Pi(n)$ can be "reproduced" by a computation $\mathbf{S}(\mathcal{C})$ of $\mathbf{S}(\Pi(n))$ with a delay: from step $r+1$ to step $r+1+m$, the computation $\mathbf{S}(\mathcal{C})$ restricted to membranes $1, \dots, q$ provides the computation $\mathcal{C}$ of $\Pi(n)$.

From Lemma 7.2 and Lemma 7.3 we deduce the following:

(a) $\mathbf{S}(\mathcal{C})$ is a computation of $\mathbf{S}(\Pi(n))$.

(b) $\mathbf{S}$ is an injective function from $\mathbf{Comp}(\Pi(n))$ onto $\mathbf{Comp}(S(\Pi(n)))$.

Proposition 7.5. *The P system* $\mathbf{S}(\Pi(n))$ *defined in 7.6 simulates* $\Pi(n)$ *in an efficient way.*

Proof. In order to show that $\mathbf{S}(\Pi(n))$ can be constructed from $\Pi(n)$ by a deterministic Turing machine working in polynomial time, it is enough to note that the amount of resources needed to construct $\mathbf{S}(\Pi(n))$ from $\Pi(n)$ is polynomial in the size of the initial resources of $\Pi(n)$. Indeed,

1. The size of the alphabet of $\mathbf{S}(\Pi(n))$ is $|\Gamma'| = |\Gamma| + r$.
2. The initial number of membranes of $\mathbf{S}(\Pi(n))$ is $1 + q \cdot (r+2) + |\mathcal{E}|$.
3. The initial number of objects of $\mathbf{S}(\Pi(n))$ is the initial number of objects of $\Pi(n)$ plus $|\mathcal{E}|$.
4. The number of rules of $\mathbf{S}(\Pi(n))$ is $|\mathcal{R}'| = |\mathcal{R}| + (r+1) \cdot |\mathcal{E}| + |\Gamma| \cdot q \cdot (r+1)$.
5. The maximal length of a communication rule of $\mathbf{S}(\Pi(n))$ is equal to the maximal length of a communication rule of $\Pi(n)$.

From Lemma 7.2 and Lemma 7.3 we deduce that:

(a) Every computation $\mathcal{C}'$ of $\mathbf{S}(\Pi(n))$ has associated a computation $\mathcal{C}$ of $\Pi(n)$ such that $\mathbf{S}(\mathcal{C}) = \mathcal{C}'$ in a natural way.

(b) The function $\mathbf{S}$ is injective.

(c) A computation $\mathcal{C}$ of $\Pi(n)$ is an accepting computation if and only if $\mathbf{S}(\mathcal{C})$ is an accepting computation of $\mathbf{S}(\Pi(n))$.

Finally, let us notice that if $\mathcal{C}$ is a computation of $\Pi(n)$ with length m, then $\mathbf{S}(\mathcal{C})$ is a computation of $\mathbf{S}(\Pi(n))$ with length $r + 1 + m$.

$\square$

7.3.3.2 Computational efficiency of P systems with membrane division and without environment

In this Section, we analyze the role of the environment in the efficiency of P systems with membrane division. That is, we study the ability of these P systems with respect to the computational efficiency when the alphabet of the environment is an empty set.

Theorem 7.6. *For each* $k \geq 1$ *we have* $\mathbf{PMC}_{\mathbf{CDC}(k)} = \mathbf{PMC}_{\widehat{\mathbf{CDC}}(k)}$.

Proof: Let us recall that $\mathbf{PMC}_{\mathbf{CDC}(1)} = \mathbf{P}$. Then,

$$\mathbf{P} \subseteq \mathbf{PMC}_{\widehat{\mathbf{CDC}}(1)} \subseteq \mathbf{PMC}_{\mathbf{CDC}(1)} = \mathbf{P}$$

Thus, the result holds for $k = 1$. Let us show the result for $k \geq 2$. Since $\widehat{\mathbf{CDC}}(k) \subseteq \mathbf{CDC}(k)$ it suffices to prove that $\mathbf{PMC}_{\mathbf{CDC}(k)} \subseteq \mathbf{PMC}_{\widehat{\mathbf{CDC}}(k)}$. For that, let $X \in \mathbf{PMC}_{\mathbf{CDC(k)}}$.

Let $\{\Pi(n) : n \in N\}$ be a family of P systems from $\mathbf{CDC}(k)$ solving X according to Definition 2.14. Let (cod, s) be a polinomial encoding associated with that solution. Let $u \in I_X$ be an instance of the problem X that will be processed by the system $\Pi(s(u)) + cod(u)$. According to Proposition 7.4, let $r(n)$ be a polynomial function such that $2^{r(|u|)}$ is an upper bound of the number of objects from $\mathcal{E}$ which are moved from the environment to all membranes of the system by any computation of

$$\Pi(s(u)) + cod(u) = (\Gamma, \mathcal{E}, \Sigma, \mathcal{M}_1, \ldots, \mathcal{M}_{i_{in}} + cod(u), \ldots, \mathcal{M}_{q_1}, \mathcal{R}, i_{in}, i_{out})$$

Then, we consider the P system without environment

$$\mathbf{S}(\Pi(s(u))) + cod(u) = (\Gamma', \Sigma', \mathcal{M}'_0, \mathcal{M}'_1, \ldots, \mathcal{M}'_{i_{in}} + cod(u), \ldots, \mathcal{M}'_{q_1}, \mathcal{R}', i'_{in}, i'_{out})$$

according to Definition 7.6, where $q_1 = 1 + q \cdot (r(|u|) + 2) + |\mathcal{E}|$.

Therefore, $\mathbf{S}(\Pi(s(u))) + cod(u)$ is a P system from $\widehat{\mathbf{CDC}}(k)$ such that verifies the following:

- A distinguished membrane labelled by 0 has been considered, which will play the role of the environment at the system $\Pi(s(u)) + cod(u)$.
- At the initial configuration, it has enough objects in membrane 0 in order to simulate the behaviour of the environment of the system $\Pi(s(u))) + cod(u)$.
- After $r(n) + 1$ step, computations of $\Pi(s(u)) + cod(u)$ are reproduced by the computations of $\mathbf{S}(\Pi(s(u))) + cod(u)$ exactly.

Let us suppose that $\mathcal{E} = \{b_1, \ldots, b_\alpha\}$. In order to simulate $\Pi(s(u)) + cod(u)$ by a P system without environment in an efficient way, we need to have enough objects in the membrane of $\mathbf{S}(\Pi(s(u))) + cod(u)$ labelled by 0 available. That is, $2^{r(n)}$ objects in that membrane are enough.

In order to start the simulation of any computation $\mathcal{C}$ of $\Pi(s(u)) + cod(u)$, it would be enough to have $2^{r(n)}$ copies of each object $b_j \in \mathcal{E}$ in the membrane of $\mathbf{S}(\Pi(s(u))) + cod(u)$ labelled by 0. For this purpose

- For each $b \in \mathcal{E}$ we consider a membrane in $\mathbf{S}(\Pi(s(u)))+cod(u)$ labelled by l_b which only contains object α_0 initially. We also consider the following rules:
  - $[\alpha_j]_{l_b} \to [\alpha_{j+1}]_{l_b}\ [\alpha_{j+1}]_{l_b}$, for $0 \leq j \leq r(|u|) - 2$.
  - $[\alpha_{p(n)-1}]_{l_b} \to [b]_{l_b}\ [b]_{l_b}$.
  - $(b, out) \in \mathcal{R}_{l_b}$.
- By applying the previous rules, after $r(|u|)$ transition steps we get $2^{r(|u|)}$ membranes labelled by l_b, for each $b \in \mathcal{E}$ in such a way that each of them contains only object b. Finally, by applying the third rule we get $2^{r(|u|)}$ copies of objects b in membrane 0, for each $b \in \mathcal{E}$.

Therefore, after the execution of $r(|u|) + 1$ transition steps in each computation of $\mathbf{S}(\Pi(s(u))) + cod(u)$ in membrane 0 of the corresponding configuration, we have $2^{r(|u|)}$ copies of each object $b \in \mathcal{E}$. This number of copies is enough to simulate any computation $\mathcal{C}$ of $\Pi(s(u)) + cod(u)$ through the system $\mathbf{S}(\Pi(s(u)) + cod(u))$.

From Proposition 7.5 we deduce that the family $\{\mathbf{S}(\Pi(n))|\ n \in N\}$ solves X in polynomial time according to Definition 2.14. Hence, $X \in \mathbf{PMC}_{\widehat{\mathbf{CDC}(k)}}$. $\square$

7.4 Frontiers of the Efficiency in P Systems with Symport/Antiport

Next, based on the results from previous sections, we present different frontiers of the efficiency in terms of syntactical ingredients of recognizer P systems with symport/antiport rules and membrane division or membranes separation (with or without environment).

1. The class $\mathbf{CDC}(1)$ is feasible and the class $\mathbf{CDC}(2)$ is presumably efficient. Hence, the length of communication rules in systems from $\mathbf{CDC}$ provides a borderline of the efficiency. Specifically, in the framework of recognizer P systems with membrane division, passing from non cooperation (length of communication rules equal to 1) to minimal cooperation (length of communication rules at most 2) amounts to passing from feasibility to presumable efficiency. An optimal frontier of the efficiency is obtained passing from 1 to 2.

2. The class $\mathbf{CSC}(2)$ is feasible and the class $\mathbf{CDC}(2)$ is presumably efficient. Hence, when only minimal cooperation is used in communication rules, allowing division rules instead separation rules amounts to passing from feasibility to presumable efficiency.

3. The class $\mathbf{CSC}(2)$ is feasible and the classe $\mathbf{CSC}(3)$ is presumably efficient. Hence, the length of communication rules in systems from **CSC** provides a borderline of the efficiency. Specifically, in the framework of recognizer P systems with membrane division, passing from minimal cooperation (length of communication rules at most 2) to allow communication rules with length at most 3, amounts to passing from feasibility to presumable efficiency. An optimal frontier of the efficiency is obtained passing from 2 to 3.

4. The class $\widehat{\mathbf{CSC}}$ is feasible. In particular, the class $\widehat{\mathbf{CSC}}(3)$ is feasible but the classe $\mathbf{CSC}(3)$ is presumably efficient. Hence, in the framework of recognizer P systems with membrane separation and communication rules with length at most 3, the use of objects with infty multiplicity provides a borderline between tractability and **NP**–hardness.

5. For each $k \geq 2$, the class $\widehat{\mathbf{CSC}}(k)$ is feasible and the class $\widehat{\mathbf{CDC}}(k)$ is presumably efficient. Thus, in the framework of recognizer P systems without environment whose communication rules have length at most $k \geq 2$, allowing division rules instead of separation rules amounts to passing from feasibility to presumably efficiency.

7.4.1 Does structure matter?

In this subsection we analyze the role played by the structure associated with different membrane systems which use symport/antiport rules as communication among processing units (*membranes* in the case of cell-like approach or *cells* in the case of tissue-like approach) from a computational complexity point of view. These structures are *rooted trees* (in cell-like P systems) an *undirected graphs* (in tissue-like P systems). It is clear that a rooted tree is a particular case of an undirected graph. Then, is there some qualitative advantage to consider tissue-like structure instead of cell-like structure?

According with the results presented in Section 3.4.3 and in Section 7.4 we have the following:

Feasible	Presumable Efficiency	
$\mathbf{CDC}(1)$	$\mathbf{CDC}(2)$	(*length of rules*)
$\mathbf{CSC}(2)$	$\mathbf{CDC}(2)$	(*kind of rules*)
$\mathbf{CSC}(2)$	$\mathbf{CSC}(3)$	(*length of rules*)
$\widehat{\mathbf{CSC}}(3)$	$\mathbf{CSC}(3)$	(*the environment*)
$\widehat{\mathbf{CSC}}(k)$	$\widehat{\mathbf{CDC}}(k)$	($k \geq 2$: *kind of rules*)

Table 7.1: Frontiers of the Presumable Efficiency

These results show that <u>the structure</u> associated with membrane systems (rooted trees in cell-like P systems versus undirected graphs in tissue-like P systems) <u>is not relevant</u> from a computational complexity point of view.

Feasible	Presumable Efficiency	
$\mathbf{TDC}(1)$	$\mathbf{TDC}(2)$	$(length\ of\ rules)$
$\mathbf{CDC}(1)$	$\mathbf{CDC}(2)$	$(length\ of\ rules)$
$\mathbf{TSC}(2)$	$\mathbf{TDC}(2)$	$(kind\ of\ rules)$
$\mathbf{CSC}(2)$	$\mathbf{CDC}(2)$	$(kind\ of\ rules)$
$\mathbf{TSC}(2)$	$\mathbf{TSC}(3)$	$(length\ of\ rules)$
$\mathbf{CSC}(2)$	$\mathbf{CSC}(3)$	$(length\ of\ rules)$
$\widehat{\mathbf{TSC}}(3)$	$\mathbf{TSC}(3)$	$(the\ environment)$
$\widehat{\mathbf{CSC}}(3)$	$\mathbf{CSC}(3)$	$(the\ environment)$
$\widehat{\mathbf{TSC}}(k)$	$\widehat{\mathbf{TDC}}(k)$	$(k \geq 2 : kind\ of\ rules)$
$\widehat{\mathbf{CSC}}(k)$	$\widehat{\mathbf{CDC}}(k)$	$(k \geq 2 : kind\ of\ rules)$

Table 7.2: Frontiers of the Presumable Efficiency

8 P–Lingua based software for Cell-like P systems with Symport/Antiport Rules

8.1 Introduction

In Chapter 7 an extensive study on how cell-like P systems with symport/antiport rules enable finding new boundaries of the efficiency, when tackling **NP**-complete problems. It has been shown, by means of constructive proofs, that **CDC(2)** and **CSC(3)** provide frontiers between efficiency and non-efficiency in terms of computational complexity. In this way, a P system family from **CDC**(2) has been shown to solve **HAM–CYCLE** in polynomial time, while a family from **CSC**(3) has been shown to solve **SAT** in polynomial time.

Due to the extraordinary hardness of the families designs, the aforementioned results could not have been achieved without the assistance of a newly developed simulator, which has been included into P–Lingua framework. In this way, each family design has been structured into different modules (a task associated with each of them), each one being checked by using the simulator with several relevant instances. Subsequently, the module has been incorporated into the complete design. With respect to the formal verification, the simulator has been used to check that the identified invariant formulas were corroborated in the corresponding configurations. At the same time, this process has allowed to check the functioning of the simulator, by performing virtual experiments on the solutions of both **HAM-CYCLE** and **SAT**. The same procedure was also conducted when addressing a characterization of **SAT** by means of a family of recognizer P systems in **CDC(3)**, as described in [90].

The developed simulator has been included within the newly released version 5.0 of pLinguaCore library. It is worth pointing out that previous versions of P–Lingua framework provided support for cell-like symport/antiport P systems with membrane division or membrane separation rules, but only allowing *finite* multiplicities of objects in the environment. As such, the new simulator fixes an open issue in the framework, since it provides full support for *arbitrary* number of object copies in the environment, in a similar way for existing simulators for tissue-like P systems.

In this Chapter, P–Lingua simulation of cell-like P systems with symport/antiport rules with either membrane division or membrane separation rules and allowing arbitrary object copies in the environment, that we will collectively abbreviate as SAMDS P systems, is discussed. Section 8.2 covers the P–Lingua syntax for defining models belonging to such variant. Section 8.3 shows full examples of the P–Lingua definition for concrete instances of the solutions of **HAM–CYCLE** and **SAT** in **CDC(2)** and **CSC(3)**, respectively, in order to illustrate the complexity of the families design. Section 8.4 introduces a simulation algorithm reproducing semantics of SACDS P systems, which has been implemented into pLinguaCore library. Finally, Section 8.5 covers performance results of the developed simulator in reference to concrete instances of the solutions of **SAT** in **CDC(3)** and **CSC(3)**, respectively.

8.2 P–Lingua syntax for SAMDS P systems

Taking the existing P–Lingua syntax for tissue-like P systems introduced in [131, 132] as a starting point, we now review the syntax for SAMDS P systems.

8.2.1 Reserved words

The set of reserved words has been updated by adding the following text string:

```
@msInfEnv
```

8.2.2 Model specification

Any P–Lingua file defining a SAMDS P system must begin with the following sentence:

```
@model<infEnv_symport_antiport>
```

8.2.3 Initial membrane structure

SAMDS P systems are a variant of cell-like P systems where the environment plays an active role. In order to specify the initial membrane structure, the same syntax for cell-like P systems from [131] applies, with one constraint: the environment must be defined with a virtual membrane labelled with the `environment` label. This virtual membrane will be placed as the outer-most membrane in the structure specification, in the following way:

```
@mu = [ ... ]'environment;
```

where `...` stands for the definition of the membrane structure as usual (starting with the skin membrane).

8.2.4 Definition of initial multisets

When defining SAMDS P systems, it is necessary to specify the objects initially placed both in the membranes and the environment.

Membranes

Syntax for specifying the initial multiset for a membrane labelled with `label` is the same from [131]. The following sentence is used:

```
@ms(label) = MULTISET_OF_OBJECTS;
```

where `MULTISET_OF_OBJECTS` is a comma-separated list of objects, which multiplicity can be expressed with the operator `*`. Character `#` represents the empty multiset. Example:

```
@ms(2) = a1*10,b;
```

Using the operator `=` enables specifying the multiset contents with a single sentence. Alternatively, the operator `+=` enables cumulative addition of objects to the corresponding multiset through a sequence of several sentences.

Environment

In order to specify the objects initially placed in environment (each appearing in an arbitrary number of copies), the following sentence must be written:

```
@msInfEnv = OBJECTS;
```

where `OBJECTS` is a comma-separated list of objects (no multiplicity is specified, as it is supposed to be arbitrary). Example:

```
@msInfEnv = a,b,c;
```

The operator `+=` can be used also, as described above.

8.2.5 Definition of the partition of the working alphabet

In order to define the partition Γ_1 and Γ_2 of the working alphabet Γ, associated with a SAMDS P system with membrane separation, the following pair of sentences is used:

```
@ms1 = OBJECTS1;
@ms2 = OBJECTS2;
```

where `OBJECTS1` and `OBJECTS2` are comma-separated lists of objects. Examples:

```
@ms1 = b,c;
@ms2 = e,f;
```

The operator `+=` can be used also, as described above.

8.2.6 Definition of rules

Four types of rules can be defined:

(1) *Symport rules*, that can be specified in the following ways:

- Rules of type $(u\,,\,in) \in \mathcal{R}_i$ are defined as

  ```
  u[]'i --> [u]'i;
  ```

- Rules of type $(u\,,\,out) \in \mathcal{R}_i$ are defined as

```
[u]'i --> u[]'i;
```

(2) *Antiport rules*, of the form $(u\,,\,out\,;\,v\,,\,in) \in \mathcal{R}_i$, are defined as

```
u[v]'i --> v[u]'i;
```

(3) *Membrane division rules*, of the form $[\,a\,]_i \rightarrow [\,b\,]_i\,[\,c\,]_i$, are defined as

```
[a]'i --> [b]'i[c]'i;
```

(4) *Membrane separation rules*, of the form $[\,a\,]_i \rightarrow [\,\Gamma_1\,]_i\,[\,\Gamma_2\,]_i$, are defined as

```
[a]'i --> []'i[]'i;
```

where i is a membrane label of the SAMDS P system described, u and v are multisets of objects and a and b are objects.

8.2.7 Two basic examples

Next, we show two basic models defined in P–Lingua to illustrate the described syntax.

The first one corresponds to a model in **CDC**.

```
@model<infEnv_symport_antiport>

def main()
{
   call module_init_conf();
   call module_rules();
}

def module_init_conf()
{
   @mu = [    [ [ []'2 ]'1]'0 ]'environment; /* defining membrane structure */
   @ms(1) = a,b,c; /* initial multiset for membrane 1 */
   @ms(2) = e,f;   /* initial multiset for membrane 2 */
   @msInfEnv = d;  /* environment objects with arbitrary number of copies */
}

def module_rules()
{
   [b]'1 --> b[]'1;        /* Communication rule */
   [a]'1 --> [b]'1[c]'1;   /* Division rule */
   [f]'2 --> [g]'2[h]'2;   /* Division rule */
}
```

The second one corresponds to a model in **CSC**.

```
def main()
{
   call module_init_conf();
   call module_rules();
}

def module_init_conf()
{
   @mu = [    [ [ []'2 ]'1]'0 ]'environment; /* defining membrane structure */
   @ms(1) = a,b,c,e,f;   /* initial multiset for membrane 1 */
   @ms(2) = g,h;         /* initial multiset for membrane 2 */
   @msInfEnv = d;        /* environment objects with arbitrary number of copies */
   @ms1 += b,c;          /* first partition */
   @ms2 += e,f;          /* second partition */
}

def module_rules()
{
   b[]'1 --> [b]'1;  /* Communication rule */
   [a]'1 --> []'1[]'1; /* Separation rule */
}
```

8.3 Full examples

In this Section we show full examples of the P–Lingua definition for concrete instances of the solutions of **HAM–CYCLE** and **SAT** in **CDC(2)** and **CSC(3)**, respectively, provided in Chapter 7, in order to illustrate the complexity of the families designed.

8.3.1 HAM–CYCLE in CDC(2)

The solution provided for **HAM–CYCLE** in **CDC(2)** requires an exponential amount of space to be executed. Specifically, the space is of order $O(2^{n \cdot p} + n^3 \cdot 2^{n^3})$, where n is the number of nodes and p is the number of egdes of the graph. Consequently, the simulator only runs for small instances.

Let us consider the instance of **HAM–CYCLE(3,3)** shown at Fig.8.1.

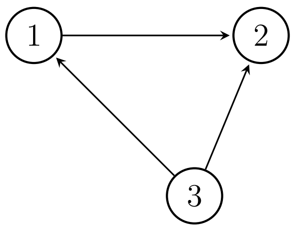


Figure 8.1: A simple instance of **HAM–CYCLE**

The following model in **CDC(2)** written in P–Lingua solves such instance:

```
@model<infEnv_symport_antiport>

def main()
{
let n = 3; /* nodes */
let p = 3; /* edges */

call module_init_conf(n);
call module_rules(n,p);
call module_input_instance();
}

def module_init_conf(n)
{

@mu = [[ []'2 []'3 []'4 []'5 []'6 []'7 []'8 []'9 []'10 []'11
        []'a{1,1} []'a{1,2} []'a{1,3} []'a{2,1} []'a{2,2} []'a{2,3}
        []'e{1,1,1} []'e{1,1,2} []'e{1,1,3} []'e{1,2,1} []'e{1,2,2}
        []'e{1,2,3} []'e{1,3,1} []'e{1,3,2} []'e{1,3,3}
        []'e{2,1,1} []'e{2,1,2} []'e{2,1,3} []'e{2,2,1} []'e{2,2,2}
        []'e{2,2,3} []'e{2,3,1} []'e{2,3,2} []'e{2,3,3}
        []'e{3,1,1} []'e{3,1,2} []'e{3,1,3} []'e{3,2,1} []'e{3,2,2}
        []'e{3,2,3} []'e{3,3,1} []'e{3,3,2} []'e{3,3,3}
]'1]'environment;

@msInfEnv += alpha{r} : 1 <= r <= n^3 + 6;

@ms(1) += alpha{0};
@ms(1) += beta{r} : 1 <= r <= n^3 + 7;
@ms(1) += bp{r},bpp{r},bppp{r} : 1 <= r <= n^3 - 1;
@ms(1) += cp{r},cpp{r},cppp{r},cpppp{r} : 1 <= r <= n^3 - 1;
@ms(2) += a*n,b,c;
@ms(3) += bp{n^3};
@ms(4) += bpp{n^3};
@ms(5) += bppp{n^3};
@ms(6) += cp{n^3};
@ms(7) += cpp{n^3};
@ms(8) += cppp{n^3};
@ms(9) += cpppp{n^3};
@ms(10) += yes;
@ms(11) += no,beta{0};
@ms(a{1,j}) = ap{n^3} : 1 <= j <= n;
@ms(a{2,j}) = app{n^3} : 1 <= j <= n;
@ms(e{i,j,k}) = epp{i,j,k,n^3} : 1<=i<=n, 1<=j<=n, 1<=k<=n;
}
```

```
def module_rules(n,p)
{

/* R1 */

alpha{r+1}[alpha{r}]'1 --> alpha{r}[alpha{r+1}]'1 : 0 <= r <= n^3+5;
[yes]'1 --> yes[]'1;
[no,alpha{n^3+6}]'1 --> no,alpha{n^3+6}[]'1;

/* R2 */

[e{i,j,k}]'2 --> [ep{i,j,k}]'2 [sharp]'2 :
1 <= i <= n, 1 <= j <= n, 1 <= k <= n;

ap[a]'2 --> a[ap]'2;
app[ap]'2 --> ap[app]'2;
bp[b]'2 --> b[bp]'2;
bpp[bp]'2 --> bp[bpp]'2;
bppp[bpp]'2 --> bpp[bppp]'2;
cp[c]'2 --> c[cp]'2;
cpp[cp]'2 --> cp[cpp]'2;
cppp[cpp]'2 --> cpp[cppp]'2;
cpppp[cppp]'2 --> cppp[cpppp]'2;
[app,bppp]'2 --> app,bppp[]'2;
[bppp,cpppp]'2 --> bppp,cpppp[]'2;

epp{i,j,k}[ep{i,j,k}]'2 --> ep{i,j,k}[epp{i,j,k}]'2 :
1 <= i <= n, 1 <= ip <= n, 1 <= j <= n, 1 <= jp <= n,
1 <= k <= n, 1 <= kp <= n;

[epp{i,j,k},epp{i,jp,kp}]'2 --> epp{i,j,k},epp{i,jp,kp}[]'2 :
1 <= i <= n, 1 <= ip <= n, 1 <= j <= n, 1 <= jp <= n,
1 <= k <= n, 1 <= kp <= n;

[epp{i,j,k},epp{ip,j,kp}]'2 --> epp{i,j,k},epp{ip,j,kp}[]'2 :
1 <= i <= n, 1 <= ip <= n, 1 <= j <= n, 1 <= jp <= n,
1 <= k <= n, 1 <= kp <= n;

[epp{i,j,k},epp{ip,jp,k+1}]'2 --> epp{i,j,k},epp{ip,jp,k+1}[]'2 :
1 <= i <= n, j <> ip, 1 <= ip <= n, 1 <= j <= n, 1 <= jp <= n,
1 <= k <= n, 1 <= kp <= n;

[epp{i,j,k},epp{ip,jp,k}]'2 --> epp{i,j,k},epp{ip,jp,k}[]'2 :
1 <= i <= n, 1 <= ip <= n, 1 <= j <= n, 1 <= jp <= n,
1 <= k <= n, 1 <= kp <= n;

[app,epp{i,j,k}]'2 --> app,epp{i,j,k}[]'2 : 1 <= i <= n,
1 <= j <= n, 1 <= k <= n;

/* R3 */

bp{r-1}[bp{r}]'3 --> bp{r}[bp{r-1}]'3 : n*p + 1 <= r <= n^3;
[bp{r}]'3 --> [bp{r-1}]'3 [bp{r-1}]'3 : 2 <= r <= n*p;
[bp{1}]'3 --> [bp]'3 [bp]'3;
[bp]'3 --> bp[]'3;

/* R4 */
```

```
bpp{r-1}[bpp{r}]'4 --> bpp{r}[bpp{r-1}]'4 : n*p + 1 <= r <= n^3;
[bpp{r}]'4 --> [bpp{r-1}]'4 [bpp{r-1}]'4 : 2 <= r <= n*p;
[bpp{1}]'4 --> [bpp]'4 [bpp]'4;
[bpp]'4 --> bpp[]'4;

/* R5 */

bppp{r-1}[bppp{r}]'5 --> bppp{r}[bppp{r-1}]'5 : n*p + 1 <= r <= n^3;
[bppp{r}]'5 --> [bppp{r-1}]'5 [bppp{r-1}]'5 : 2 <= r <= n*p;
[bppp{1}]'5 --> [bppp]'5 [bppp]'5;
[bppp]'5 --> bppp[]'5;

/* R6 */

cp{r-1}[cp{r}]'6 --> cp{r}[cp{r-1}]'6 : n*p + 1 <= r <= n^3;
[cp{r}]'6 --> [cp{r-1}]'6 [cp{r-1}]'6 : 2 <= r <= n*p;
[cp{1}]'6 --> [cp]'6 [cp]'6;
[cp]'6 --> cp[]'6;

/* R7 */

cpp{r-1}[cpp{r}]'7 --> cpp{r}[cpp{r-1}]'7 : n*p + 1 <= r <= n^3;
[cpp{r}]'7 --> [cpp{r-1}]'7 [cpp{r-1}]'7 : 2 <= r <= n*p;
[cpp{1}]'7 --> [cpp]'7 [cpp]'7;
[cpp]'7 --> cpp[]'7;

/* R8 */

cppp{r-1}[cppp{r}]'8 --> cppp{r}[cppp{r-1}]'8 : n*p + 1 <= r <= n^3;
[cppp{r}]'8 --> [cppp{r-1}]'8 [cppp{r-1}]'8 : 2 <= r <= n*p;
[cppp{1}]'8 --> [cppp]'8 [cppp]'8;
[cppp]'8 --> cppp[]'8;

/* R9 */

cpppp{r-1}[cpppp{r}]'9 --> cpppp{r}[cpppp{r-1}]'9 : n*p + 1 <= r <= n^3;
[cpppp{r}]'9 --> [cpppp{r-1}]'9 [cpppp{r-1}]'9 : 2 <= r <= n*p;
[cpppp{1}]'9 --> [cpppp]'9 [cpppp]'9;
[cpppp]'9 --> cpppp[]'9;

/* R10 */

alpha{n^3+6},cpppp[]'10 --> [alpha{n^3+6},cpppp]'10;
[cpppp,yes]'10 --> cpppp,yes[]'10;

/* R11 */

beta{r+1}[beta{r}]'11 --> beta{r}[beta{r+1}]'11 : 0 <= r <= n^3 + 6;
[beta{n^3+7},no]'11 --> beta{n^3+7},no[]'11;

/* Ra{1,j} : 1 <= j <= n */

[ap{r}]'a{1,j} --> [ap{r-1}]'a{1,j} [ap{r-1}]'a{1,j} : 2 <= r <= n^3, 1 <= j <= n;

[ap{1}]'a{1,j} --> [ap]'a{1,j} [ap]'a{1,j} : 1 <= j <= n;
[ap]'a{1,j} --> ap[]'a{1,j} : 1 <= j <= n;

/* Ra{2,j} : 1 <= j <= n */
```

```
[app{r}]'a{2,j} --> [app{r-1}]'a{2,j} [app{r-1}]'a{2,j} : 2 <= r <= n^3, 1 <= j <= n;

[app{1}]'a{2,j} --> [app]'a{2,j} [app]'a{2,j} : 1 <= j <= n;
[app]'a{2,j} --> app[]'a{2,j} : 1 <= j <= n;

/* Re{i,j,k} : 1 <= i,j,k <= n */

[epp{i,j,k,r}]'e{i,j,k} --> [epp{i,j,k,r-1}]'e{i,j,k} [epp{i,j,k,r-1}]'e{i,j,k} :
2 <= r <= n^3, 1 <= i <= n, 1 <= j <= n, 1 <= k <= n;

[epp{i,j,k,1}]'e{i,j,k} --> [epp{i,j,k}]'e{i,j,k} [epp{i,j,k}]'e{i,j,k} :
1 <= i <= n, 1 <= j <= n, 1 <= k <= n;

[epp{i,j,k}]'e{i,j,k} --> epp{i,j,k}[]'e{i,j,k} :
1 <= i <= n, 1 <= j <= n, 1 <= k <= n;
}

def module_input_instance()
{
@ms(2) += e{1,2,1},e{1,2,2},e{1,2,3},e{3,1,1},e{3,1,2},e{3,1,3},e{3,2,1},e{3,2,2},e{3,2,3};
}
```

8.3.2 SAT in CSC(3)

An arbitrary instance of the **SAT** problem is a boolean formula in conjunctive normal form $\varphi = C_1 \wedge \ldots \wedge C_m$ with n variables $\{x_1, \ldots, x_n\}$ and m clauses $\{C_1, \ldots, C_m\}$. We denote it by $\varphi \in \mathbf{SAT}(n, m)$. Let us recall that formula φ can be codified as $cod(\varphi) = \{x_{i,j} \mid x_i \in C_j\} \cup \{\overline{x}_{i,j} \mid \neg x_i \in C_j\}$.

Let us consider the instance of $\mathbf{SAT}(9, 10)$ encoded by

$$cod(\varphi) = \left\{ \begin{array}{l} x_{3,1}, x_{8,1}, \\ x_{1,2}, \overline{x}_{2,2}, x_{5,2}, \overline{x}_{6,2}, x_{9,2}, \\ x_{3,3}, x_{6,3}, x_{9,3}, \\ x_{3,4}, x_{5,4}, \overline{x}_{6,4}, \overline{x}_{8,4}, \\ x_{1,5}, x_{2,5}, \overline{x}_{5,5}, x_{7,5}, \overline{x}_{8,5}, \overline{x}_{9,5}, \\ \overline{x}_{1,6}, x_{2,6}, \overline{x}_{4,6}, x_{5,6}, \overline{x}_{6,6}, \overline{x}_{7,6}, x_{9,6}, \\ x_{1,7}, x_{2,7}, x_{4,7}, \overline{x}_{6,7}, x_{8,7}, \overline{x}_{9,7}, \\ \overline{x}_{1,8}, x_{2,8}, \overline{x}_{3,8}, \overline{x}_{4,8}, x_{7,8}, \overline{x}_{8,8}, \\ \overline{x}_{1,9}, x_{2,9}, x_{3,9}, x_{5,9}, \overline{x}_{6,9}, x_{8,9}, \overline{x}_{9,9}, \\ x_{2,10}, \overline{x}_{3,10}, x_{4,10}, \overline{x}_{6,10}, \overline{x}_{7,10}, \overline{x}_{9,10} \end{array} \right\}$$

The following model in **CSC(3)** written in P–Lingua solves such instance:

```
@model<infEnv_symport_antiport>

def main()
{
let n = 9;  /* variables */
let m = 10; /* clauses */

call module_alphabet(n,m);
call module_init_conf(n,m);
call module_rules(n,m);
call module_input();
}

def module_alphabet (n,m)
{
@msInfEnv += alpha{i,j,k}, alphap{i,j,k} : 0 <= k <= 1, 1 <= j <= 3*(n-1), 1 <= i <= n-1;

@msInfEnv += beta{j},betap{j},betapp{j},gamma{j},gammap{j},gammapp{j},gammappp{j} :
0 <= j <= 3*(n-1);

@msInfEnv += rho{i,j}, tau{i,j} : 1 <= j <= 3*n-1, 1 <= i <= n;
@msInfEnv += T{i,j},Tp{i,j},F{i,j},Fp{i,j} : i< j <= n, 1 <= i <= n-1;
@msInfEnv += T{i,i},Fp{i,i} : 1 <= i <= n;
@msInfEnv += T{i},F{i} : 1 <= i <= n;
@msInfEnv += A{i},Ap{i},B{i},Bp{i} : 2 <= i <= n + 1;
@msInfEnv += b{i},bp{i},c{i},cp{i} : 2 <= i <= n;
@msInfEnv += v{i} : 2 <= i <= n-1;
@msInfEnv += y{i},a{i},w{i} : 1 <= i <= n-1;
@msInfEnv += z{i} : 1 <= i <= n-2;
@msInfEnv += q{i,j},t{i,j},f{i,j},r{i,j},s{i,j} : i <= j <= n-1, 1 <= i <= n-1;
@msInfEnv += u{i,j} : i <= j <= n-2, 1 <= i <= n-2;
@msInfEnv += e{i,j},_e{i,j} : 1 <= j <= m, 1 <= i <= n;
@msInfEnv += d{i,j,k},_d{i,j,k} : 1 <= k <= n-1, 1 <= j <= m, 1 <= i <= n;
@msInfEnv += f{r} : 1 <= r <= 3*n+2*m;
@msInfEnv += fp{r} : 1 <= r <= 3*n+2*m+1;
@msInfEnv += delta{s,j} : 1 <= j <= 3*n, 0 <= s <= m;
@msInfEnv += E{s} : 0 <= s <= m;
@msInfEnv += S;
}

def module_init_conf(n,m)
{
@mu = [ [[]'2 []'3]'1 ]'environment;

@ms1 += A{i},B{i} : 1 <= i <= n + 1;
@ms1 += T{i,j},F{i,j} : 1 <= j <= n, 1 <= i <= n;

@ms2 += Ap{i},Bp{i} : 2 <= i <= n + 1;
@ms2 += Tp{i,j},Fp{i,j} : i < j <= n, 1 <= i <= n-1;
@ms2 += Fp{i,i} : 1 <= i <= n;

@ms(1) += alpha{i,0,k},alphap{i,0,k} : 0 <= k <= 1, 1 <= i <= n-1;
@ms(1) += rho{i,0},tau{i,0} : 1 <= i <= n;

@ms(1) += beta{0},betap{0},betapp{0},
gamma{0},gammap{0},gammapp{0},gammappp{0},sigma{0},sigmap{0};

@ms(1) += c{1},cp{1},b{1},bp{1},v{1},q{1,1}, f{0},yes;
@ms(1) += delta{j,0} : 0 <= j <= m;
@ms(1) += fp{r} : 1 <= r <= 3*n+2*m+1;

@ms(2) += A{1},B{1};

@ms(3) += fp{0},no;
```

```
}

def module_rules(n,m)
{
call module_rules_R1(n,m);
call module_rules_R2(n,m);
call module_rules_R3(n,m);
}

def module_rules_R1 (n,m)
{
alpha{i,j+1,k}[alpha{i,j,k}]'1 -->
alpha{i,j,k}[alpha{i,j+1,k}]'1 : 0<=j<=2, 0 <= k <= 1, 1 <= i <= n-1;

alphap{i,j+1,k}[alphap{i,j,k}]'1 -->
alphap{i,j,k}[alphap{i,j+1,k}]'1 : 0<=j<=2, 0 <= k <= 1, 1 <= i <= n-1;

alpha{i,3*r+1,0},F{i,r+1}[alpha{i,3*r,0}]'1 -->
alpha{i,3*r,0}[alpha{i,3*r+1,0},F{i,r+1}]'1 : 1 <= i <= r, 1 <= r <= n - 2;

alpha{i,3*r+1,1},T{i,r+1}[alpha{i,3*r,1}]'1 -->
alpha{i,3*r,1}[alpha{i,3*r+1,1},T{i,r+1}]'1 : 1 <= i <= r, 1 <= r <= n - 2;

alphap{i,3*r+1,0},Fp{i,r+1}[alphap{i,3*r,0}]'1 -->
alphap{i,3*r,0}[alphap{i,3*r+1,0},Fp{i,r+1}]'1 : 1 <= i <= r, 1 <= r <= n - 2;

alphap{i,3*r+1,1},Tp{i,r+1}[alphap{i,3*r,1}]'1 -->
alphap{i,3*r,1}[alphap{i,3*r+1,1},Tp{i,r+1}]'1 : 1 <= i <= r, 1 <= r <= n - 2;

alpha{i,3*r+1,k}[alpha{i,3*r,k}]'1 -->
alpha{i,3*r,k}[alpha{i,3*r+1,k}]'1 : 0 <= k <= 1, r + 1 <= i <= n - 1 , 1 <= r <= n - 2;

alphap{i,3*r+1,k}[alphap{i,3*r,k}]'1 -->
alphap{i,3*r,k}[alphap{i,3*r+1,k}]'1 : 0 <= k <= 1, r + 1 <= i <= n - 1 , 1 <= r <= n - 2;

alpha{i,3*r+2,k}[alpha{i,3*r+1,k}]'1 -->
alpha{i,3*r+1,k}[alpha{i,3*r+2,k}]'1 : 0 <= k <= 1, 1 <= i <= n - 1, 1 <= r <= n - 2;

alphap{i,3*r+2,k}[alphap{i,3*r+1,k}]'1 -->
alphap{i,3*r+1,k}[alphap{i,3*r+2,k}]'1 : 0 <= k <= 1, 1 <= i <= n - 1, 1 <= r <= n - 2;

alpha{i,3*r+3,k}*2[alpha{i,3*r+2,k}]'1 -->
alpha{i,3*r+2,k}[alpha{i,3*r+3,k}*2]'1 : 0 <= k <= 1, 1 <= i <= n - 1, 1 <= r <= n - 2;

alphap{i,3*r+3,k}*2[alphap{i,3*r+2,k}]'1 -->
alphap{i,3*r+2,k}[alphap{i,3*r+3,k}*2]'1 : 1 <= i <= n - 1, 1 <= r <= n - 2, 0 <= k <= 1;

F{i,n}[alpha{i,3*(n-1),0}]'1 --> alpha{i,3*(n-1),0}[F{i,n}]'1 : 1 <= i <= n - 1;
T{i,n}[alpha{i,3*(n-1),1}]'1 --> alpha{i,3*(n-1),1}[T{i,n}]'1 : 1 <= i <= n - 1;

Fp{i,n}[alphap{i,3*(n-1),0}]'1 --> alphap{i,3*(n-1),0}[Fp{i,n}]'1 : 1 <= i <= n - 1;
Tp{i,n}[alphap{i,3*(n-1),1}]'1 --> alphap{i,3*(n-1),1}[Tp{i,n}]'1 : 1 <= i <= n - 1;

/* First group of "beta rules" */

beta{3*r+1},B{r+2}[beta{3*r}]'1 --> beta{3*r}[beta{3*r+1},B{r+2}]'1 : 0 <= r <= n - 3;
betap{3*r+1},Bp{r+2}[betap{3*r}]'1 --> betap{3*r}[betap{3*r+1},Bp{r+2}]'1 : 0 <= r <= n - 3;
betapp{3*r+1},S[betapp{3*r}]'1 --> betapp{3*r}[betapp{3*r+1},S]'1 : 0 <= r <= n - 3;
```

```
beta{3*r+2}[beta{3*r+1}]'1 --> beta{3*r+1}[beta{3*r+2}]'1 : 0 <= r <= n - 3;
betap{3*r+2}[betap{3*r+1}]'1 --> betap{3*r+1}[betap{3*r+2}]'1 : 0 <= r <= n - 3;
betapp{3*r+2}[betapp{3*r+1}]'1 --> betapp{3*r+1}[betapp{3*r+2}]'1 : 0 <= r <= n - 3;

beta{3*r+3}*2[beta{3*r+2}]'1 --> beta{3*r+2}[beta{3*r+3}*2]'1 : 0 <= r <= n - 3;
betap{3*r+3}*2[betap{3*r+2}]'1 --> betap{3*r+2}[betap{3*r+3}*2]'1 : 0 <= r <= n - 3;
betapp{3*r+3}*2[betapp{3*r+2}]'1 --> betapp{3*r+2}[betapp{3*r+3}*2]'1 : 0 <= r <= n - 3;

/* Second group of "beta rules" */

beta{3*(n-2)+1},B{n}[beta{3*(n-2)}]'1 --> beta{3*(n-2)}[beta{3*(n-2)+1},B{n}]'1;
betap{3*(n-2)+1},Bp{n}[betap{3*(n-2)}]'1 --> betap{3*(n-2)}[betap{3*(n-2)+1},Bp{n}]'1;
betapp{3*(n-2)+1},S[betapp{3*(n-2)}]'1 --> betapp{3*(n-2)}[betapp{3*(n-2)+1},S]'1;
beta{3*(n-2)+2}[beta{3*(n-2)+1}]'1 --> beta{3*(n-2)+1}[beta{3*(n-2)+2}]'1;
betap{3*(n-2)+2}[betap{3*(n-2)+1}]'1 --> betap{3*(n-2)+1}[betap{3*(n-2)+2}]'1;
betapp{3*(n-2)+2}[betapp{3*(n-2)+1}]'1 --> betapp{3*(n-2)+1}[betapp{3*(n-2)+2}]'1;
beta{3*(n-2)+3}*2[beta{3*(n-2)+2}]'1 --> beta{3*(n-2)+2}[beta{3*(n-2)+3}*2]'1;
betap{3*(n-2)+3}*2[betap{3*(n-2)+2}]'1 --> betap{3*(n-2)+2}[betap{3*(n-2)+3}*2]'1;
betapp{3*(n-2)+3}*2[betapp{3*(n-2)+2}]'1 --> betapp{3*(n-2)+2}[betapp{3*(n-2)+3}*2]'1;

/* Third group of "beta rules" */

B{n+1}[beta{3*(n-1)}]'1 --> beta{3*(n-1)}[B{n+1}]'1;
Bp{n+1}[betap{3*(n-1)}]'1 --> betap{3*(n-1)}[Bp{n+1}]'1;
S[betapp{3*(n-1)}]'1 --> betapp{3*(n-1)}[S]'1;

/* First group of "gamma rules" */

gamma{3*r+1},T{r+1,r+1}[gamma{3*r}]'1 -->
gamma{3*r}[gamma{3*r+1},T{r+1,r+1}]'1 : 0 <= r <= n - 3;

gammap{3*r+1},Fp{r+1,r+1}[gammap{3*r}]'1 -->
gammap{3*r}[gammap{3*r+1},Fp{r+1,r+1}]'1 : 0 <= r <= n - 3;

gammapp{3*r+1},A{r+2}[gammapp{3*r}]'1 -->
gammapp{3*r}[gammapp{3*r+1},A{r+2}]'1 : 0 <= r <= n - 3;

gammappp{3*r+1},Ap{r+2}[gammappp{3*r}]'1 -->
gammappp{3*r}[gammappp{3*r+1},Ap{r+2}]'1 : 0 <= r <= n - 3;

gamma{3*r+2}[gamma{3*r+1}]'1 --> gamma{3*r+1}[gamma{3*r+2}]'1 : 0 <= r <= n - 3;
gammap{3*r+2}[gammap{3*r+1}]'1 --> gammap{3*r+1}[gammap{3*r+2}]'1 : 0 <= r <= n - 3;
gammapp{3*r+2}[gammapp{3*r+1}]'1 --> gammapp{3*r+1}[gammapp{3*r+2}]'1 : 0 <= r <= n - 3;
gammappp{3*r+2}[gammappp{3*r+1}]'1 --> gammappp{3*r+1}[gammappp{3*r+2}]'1 : 0 <= r <= n - 3;

gamma{3*r+3}*2[gamma{3*r+2}]'1 --> gamma{3*r+2}[gamma{3*r+3}*2]'1 : 0 <= r <= n - 3;
gammap{3*r+3}*2[gammap{3*r+2}]'1 --> gammap{3*r+2}[gammap{3*r+3}*2]'1 : 0 <= r <= n - 3;
gammapp{3*r+3}*2[gammapp{3*r+2}]'1 --> gammapp{3*r+2}[gammapp{3*r+3}*2]'1 : 0 <= r <= n - 3;

gammappp{3*r+3}*2[gammappp{3*r+2}]'1 -->
gammappp{3*r+2}[gammappp{3*r+3}*2]'1 : 0 <= r <= n - 3;

/* Second group of "gamma rules" */

gamma{3*(n-2)+1},T{n-1,n-1}[gamma{3*(n-2)}]'1 -->
gamma{3*(n-2)}[gamma{3*(n-2)+1},T{n-1,n-1}]'1;

gammap{3*(n-2)+1},Fp{n-1,n-1}[gammap{3*(n-2)}]'1 -->
gammap{3*(n-2)}[gammap{3*(n-2)+1},Fp{n-1,n-1}]'1;
```

```
gammapp{3*(n-2)+1},A{n}[gammapp{3*(n-2)}]'1 -->
gammapp{3*(n-2)}[gammapp{3*(n-2)+1},A{n}]'1;

gammappp{3*(n-2)+1},Ap{n}[gammappp{3*(n-2)}]'1 -->
gammappp{3*(n-2)}[gammappp{3*(n-2)+1},Ap{n}]'1;

gamma{3*(n-2)+2}[gamma{3*(n-2)+1}]'1 --> gamma{3*(n-2)+1}[gamma{3*(n-2)+2}]'1;
gammap{3*(n-2)+2}[gammap{3*(n-2)+1}]'1 --> gammap{3*(n-2)+1}[gammap{3*(n-2)+2}]'1;
gammapp{3*(n-2)+2}[gammapp{3*(n-2)+1}]'1 --> gammapp{3*(n-2)+1}[gammapp{3*(n-2)+2}]'1;
gammappp{3*(n-2)+2}[gammappp{3*(n-2)+1}]'1 --> gammappp{3*(n-2)+1}[gammappp{3*(n-2)+2}]'1;
gamma{3*(n-2)+3}*2[gamma{3*(n-2)+2}]'1 --> gamma{3*(n-2)+2}[gamma{3*(n-2)+3}*2]'1;
gammap{3*(n-2)+3}*2[gammap{3*(n-2)+2}]'1 --> gammap{3*(n-2)+2}[gammap{3*(n-2)+3}*2]'1;
gammapp{3*(n-2)+3}*2[gammapp{3*(n-2)+2}]'1 --> gammapp{3*(n-2)+2}[gammapp{3*(n-2)+3}*2]'1;

gammappp{3*(n-2)+3}*2[gammappp{3*(n-2)+2}]'1 -->
gammappp{3*(n-2)+2}[gammappp{3*(n-2)+3}*2]'1;

/* Third group of "gamma rules" */

T{n,n}[gamma{3*(n-1)}]'1 --> gamma{3*(n-1)}[T{n,n}]'1;
Fp{n,n}[gammap{3*(n-1)}]'1 --> gammap{3*(n-1)}[Fp{n,n}]'1;
A{n+1}[gammapp{3*(n-1)}]'1 --> gammapp{3*(n-1)}[A{n+1}]'1;
Ap{n+1}[gammappp{3*(n-1)}]'1 --> gammappp{3*(n-1)}[Ap{n+1}]'1;

/* First group of "rho-tau rules"*/

rho{i,j+1}[rho{i,j}]'1 --> rho{i,j}[rho{i,j+1}]'1 : 0<=j<=2, 1 <= i <= n;
tau{i,j+1}[tau{i,j}]'1 --> tau{i,j}[tau{i,j+1}]'1 : 0<=j<=2, 1 <= i <= n;

/* Second group of "rho-tau rules"*/

rho{i,3*r+1}[rho{i,3*r}]'1 --> rho{i,3*r}[rho{i,3*r+1}]'1 : 1 <= r <= n - 2, 1 <= i <= n;
tau{i,3*r+1}[tau{i,3*r}]'1 --> tau{i,3*r}[tau{i,3*r+1}]'1 : 1 <= r <= n - 2, 1 <= i <= n;

rho{i,3*r+2}*2[rho{i,3*r+1}]'1 -->
rho{i,3*r+1}[rho{i,3*r+2}*2]'1 : 1 <= r <= n - 2, 1 <= i <= n;

tau{i,3*r+2}*2[tau{i,3*r+1}]'1 -->
tau{i,3*r+1}[tau{i,3*r+2}*2]'1 : 1 <= r <= n - 2, 1 <= i <= n;

rho{i,3*r+3}[rho{i,3*r+2}]'1 -->
rho{i,3*r+2}[rho{i,3*r+3}]'1 : 1 <= r <= n - 2, 1 <= i <= n;

tau{i,3*r+3}[tau{i,3*r+2}]'1 -->
tau{i,3*r+2}[tau{i,3*r+3}]'1 : 1 <= r <= n - 2, 1 <= i <= n;

/* Third group of "rho-tau rules"*/

rho{i,3*(n-1)+1}[rho{i,3*(n-1)}]'1 -->
rho{i,3*(n-1)}[rho{i,3*(n-1)+1}]'1 : 1 <= i <= n;

tau{i,3*(n-1)+1}[tau{i,3*(n-1)}]'1 -->
tau{i,3*(n-1)}[tau{i,3*(n-1)+1}]'1 : 1 <= i <= n;

rho{i,3*(n-1)+2}*2[rho{i,3*(n-1)+1}]'1 -->
rho{i,3*(n-1)+1}[rho{i,3*(n-1)+2}*2]'1 : 1 <= i <= n;

tau{i,3*(n-1)+2}*2[tau{i,3*(n-1)+1}]'1 -->
```

```
tau{i,3*(n-1)+1}[tau{i,3*(n-1)+2}*2]'1 : 1 <= i <= n;

T{i}[rho{i,3*(n-1)+2}]'1 --> rho{i,3*(n-1)+2}[T{i}]'1 : 1 <= i <= n;
F{i}[tau{i,3*(n-1)+2}]'1 --> tau{i,3*(n-1)+2}[F{i}]'1 : 1 <= i <= n;

a{i}[A{i}]'1 --> A{i}[a{i}]'1 : 1 <= i <= n - 1;
a{i}[Ap{i}]'1 --> Ap{i}[a{i}]'1 : 1 <= i <= n - 1;
a{i}[B{i}]'1 --> B{i}[a{i}]'1 : 1 <= i <= n - 1;
a{i}[Bp{i}]'1 --> Bp{i}[a{i}]'1 : 1 <= i <= n - 1;

z{i},w{i}[y{i}]'1 --> y{i}[z{i},w{i}]'1 : 1 <= i <= n - 2;
w{n-1}[y{n-1}]'1 --> y{n-1}[w{n-1}]'1;

c{i+1},cp{i+1}[w{i}]'1 --> w{i}[c{i+1},cp{i+1}]'1 : 1 <= i <= n - 1;
v{i+1}[z{i}]'1 --> z{i}[v{i+1}]'1 : 1 <= i <= n - 2;

y{i}*2[v{i}]'1 --> v{i}[y{i}*2]'1 : 1 <= i <= n - 1;
b{i+1},bp{i+1}[a{i}]'1 --> a{i}[b{i+1},bp{i+1}]'1 : 1 <= i <= n - 1;

r{1,1}[q{1,1}]'1 --> q{1,1}[r{1,1}]'1;
r{i,j}*2[q{i,j}]'1 --> q{i,j}[r{i,j}*2]'1 : i <= j <= n - 1, 1 <= i <= n - 1;

s{i,j},u{i,j}[r{i,j}]'1 --> r{i,j}[s{i,j},u{i,j}]'1 : i <= j <= n - 2, 1 <= i <= n - 2;
s{i,n-1}[r{i,n-1}]'1 --> r{i,n-1}[s{i,n-1}]'1 : 1 <= i <= n - 1;

t{i,j},f{i,j}[s{i,j}]'1 --> s{i,j}[t{i,j},f{i,j}]'1 : i <= j <= n - 1, 1 <= i <= n - 1;

q{1,j+1},q{2,j+1}[u{1,j}]'1 --> u{1,j}[q{1,j+1},q{2,j+1}]'1 : 1 <= j <= n - 2;
q{i+1,j+1}[u{i,j}]'1 --> u{i,j}[q{i+1,j+1}]'1 : 2 <= i <= j, 2 <= j <= n - 2;

[T{i,j},t{i,j}]'1 --> T{i,j},t{i,j}[]'1 : 1 <= i <= j,1 <= j <= n;
[Tp{i,j},t{i,j}]'1 --> Tp{i,j},t{i,j}[]'1 : 1 <= i <= j,1 <= j <= n;
[F{i,j},f{i,j}]'1 --> F{i,j},f{i,j}[]'1 : 1 <= i <= j,1 <= j <= n;
[Fp{i,j},f{i,j}]'1 --> Fp{i,j},f{i,j}[]'1 : 1 <= i <= j,1 <= j <= n;

d{i,j,1}*2[x{i,j}]'1 --> x{i,j}[d{i,j,1}*2]'1 : 1 <= j <= m, 1 <= i <= n;
_d{i,j,1}*2[_x{i,j}]'1 --> _x{i,j}[_d{i,j,1}*2]'1 : 1 <= j <= m, 1 <= i <= n;

d{i,j,k+1}*2[d{i,j,k}]'1 -->
d{i,j,k}[d{i,j,k+1}*2]'1 : 1 <= k <= n-2, 1 <= j <= m, 1 <= i <= n;

_d{i,j,k+1}*2[_d{i,j,k}]'1 -->
_d{i,j,k}[_d{i,j,k+1}*2]'1 : 1 <= k <= n-2, 1 <= j <= m, 1 <= i <= n;

e{i,j}[d{i,j,n-1}]'1 --> d{i,j,n-1}[e{i,j}]'1 : 1 <= j <= m, 1 <= i <= n;
_e{i,j}[_d{i,j,n-1}]'1 --> _d{i,j,n-1}[_e{i,j}]'1 : 1 <= j <= m, 1 <= i <= n;

[E{0},f{3*n+2*m},yes]'1 --> E{0},f{3*n+2*m},yes[]'1;

[f{3*n+2*m},no]'1 --> f{3*n+2*m},no[]'1;

delta{j,3*r+1}[delta{j,3*r}]'1 -->
delta{j,3*r}[delta{j,3*r+1}]'1 : 0 <= r <= n - 1, 0 <= j <= m;

delta{j,3*r+2}*2[delta{j,3*r+1}]'1 -->
delta{j,3*r+1}[delta{j,3*r+2}*2]'1 : 0 <= r <= n - 1, 0 <= j <= m;

delta{j,3*r+3}[delta{j,3*r+2}]'1 -->
delta{j,3*r+2}[delta{j,3*r+3}]'1 : 0 <= r <= n - 2, 0 <= j <= m;
```

```
E{1}[delta{1,3*(n-1)+2}]'1 --> delta{1,3*(n-1)+2}[E{1}]'1;

delta{j,3*(n-1)+3}[delta{j,3*(n-1)+2}]'1 -->
delta{j,3*(n-1)+2}[delta{j,3*(n-1)+3}]'1 : 0 <= j <= m, j <> 1;

E{j}[delta{j,3*n}]'1 --> delta{j,3*n}[E{j}]'1 : 0 <= j <= m, j <> 1;

f{i+1}[f{i}]'1 --> f{i}[f{i+1}]'1 : 0 <= i <= 3*n + 2*m - 1;

/* garbage rules */

[t{i,k},T{i,k}]'1 --> t{i,k},T{i,k}[]'1 : 1 <= i < k, 2 <= k <= n;
[t{i,k},Tp{i,k}]'1 --> t{i,k},Tp{i,k}[]'1 : 1 <= i < k, 2 <= k <= n;
[f{i,k},F{i,k}]'1 --> f{i,k},F{i,k}[]'1 : 1 <= i < k, 2 <= k <= n;
[f{i,k},Fp{i,k}]'1 --> f{i,k},Fp{i,k}[]'1 : 1 <= i < k, 2 <= k <= n;

[t{i,i},T{i,i}]'1 --> t{i,i},T{i,i}[]'1 : 1 <= i <= n;
[f{i,i},Fp{i,i}]'1 --> f{i,i},Fp{i,i}[]'1 : 1 <= i <= n;

[b{k},B{k+1}]'1 --> b{k},B{k+1}[]'1 : n - 1 <= k <= n;
[bp{k},Bp{k+1}]'1 --> bp{k},Bp{k+1}[]'1 : n - 1 <= k <= n;
[cp{k},Ap{k+1}]'1 --> cp{k},Ap{k+1}[]'1 : n - 1 <= k <= n;
[c{k},A{k+1}]'1 --> c{k},A{k+1}[]'1 : n - 1 <= k <= n;
}

def module_rules_R2 (n,m)
{
[S]'2 --> []'2[]'2;

c{i},cp{i}[A{i}]'2 --> A{i}[c{i},cp{i}]'2 : 1<= i <= n;
c{i},cp{i}[Ap{i}]'2 --> Ap{i}[c{i},cp{i}]'2 : 1<= i <= n;

b{i},bp{i}[B{i}]'2 --> B{i}[b{i},bp{i}]'2 : 1<= i <= n;
b{i},bp{i}[Bp{i}]'2 --> Bp{i}[b{i},bp{i}]'2 : 1<= i <= n;

B{i+1},S[b{i}]'2 --> b{i}[B{i+1},S]'2 : 1<= i <= n;
Bp{i+1}[bp{i}]'2 --> bp{i}[Bp{i+1}]'2 : 1<= i <= n;

T{i,i},A{i+1}[c{i}]'2 --> c{i}[T{i,i},A{i+1}]'2 : 1<= i <= n;
Fp{i,i},Ap{i+1}[cp{i}]'2 --> cp{i}[Fp{i,i},Ap{i+1}]'2 : 1<= i <= n;

E{1}[B{n+1}]'2 --> B{n+1}[E{1}]'2;
E{1}[Bp{n+1}]'2 --> Bp{n+1}[E{1}]'2;

E{0}[A{n+1}]'2 --> A{n+1}[E{0}]'2;
E{0}[Ap{n+1}]'2 --> Ap{n+1}[E{0}]'2;

t{i,j}[T{i,j}]'2 --> T{i,j}[t{i,j}]'2 : 1 <= i <= j,1 <= j <= n;
t{i,j}[Tp{i,j}]'2 --> Tp{i,j}[t{i,j}]'2 : 1 <= i <= j,1 <= j <= n;

f{i,j}[F{i,j}]'2 --> F{i,j}[f{i,j}]'2 : 1 <= i <= j,1 <= j <= n;
f{i,j}[Fp{i,j}]'2 --> Fp{i,j}[f{i,j}]'2 : 1 <= i <= j,1 <= j <= n;

T{i,j+1},Tp{i,j+1}[t{i,j}]'2 --> t{i,j}[T{i,j+1},Tp{i,j+1}]'2 : 1 <= i <= j,1 <= j <= n-1;
F{i,j+1},Fp{i,j+1}[f{i,j}]'2 --> f{i,j}[F{i,j+1},Fp{i,j+1}]'2 : 1 <= i <= j,1 <= j <= n-1;

T{i}[T{i,n}]'2 --> T{i,n}[T{i}]'2 : 1 <= i <= n;
T{i}[Tp{i,n}]'2 --> Tp{i,n}[T{i}]'2 : 1 <= i <= n;
```

```
F{i}[F{i,n}]'2 --> F{i,n}[F{i}]'2 : 1 <= i <= n;
F{i}[Fp{i,n}]'2 --> Fp{i,n}[F{i}]'2 : 1 <= i <= n;

e{i,j}[E{j},T{i}]'2 --> E{j},T{i}[e{i,j}]'2 : 1 <= j <= m, 1 <= i <= n;
_e{i,j}[E{j},F{i}]'2 --> E{j},F{i}[_e{i,j}]'2 : 1 <= j <= m, 1 <= i <= n;

E{j+1},T{i}[e{i,j}]'2 --> e{i,j}[E{j+1},T{i}]'2 : 1 <= j <= m - 1, 1 <= i <= n;
E{j+1},F{i}[_e{i,j}]'2 --> _e{i,j}[E{j+1},F{i}]'2 : 1 <= j <= m - 1, 1 <= i <= n;

[e{i,m},E{0}]'2 --> e{i,m},E{0}[]'2 : 1 <= i <= n;
[_e{i,m},E{0}]'2 --> _e{i,m},E{0}[]'2 : 1 <= i <= n;
}

def module_rules_R3 (n,m)
{
/* R3 */
fp{r+1}[fp{r}]'3 --> fp{r}[fp{r+1}]'3 : 0 <= r <= 3*n+2*m;
[fp{3*n+2*m+1},no]'3 --> fp{3*n+2*m+1},no[]'3;
}

def module_input ()
{

/* We define here the input for the P system */

@ms(1) +=  x{3,1},x{8,1},
           x{1,2},_x{2,2},x{5,2},_x{6,2},x{9,2},
           x{3,3},x{6,3},x{9,3},
           x{3,4},x{5,4},_x{6,4},_x{8,4},
           x{1,5},x{2,5},_x{5,5},x{7,5},_x{8,5},_x{9,5},
          _x{1,6},x{2,6},_x{4,6},x{5,6},_x{6,6},_x{7,6},x{9,6},
           x{1,7},x{2,7},x{4,7},_x{6,7},x{8,7},_x{9,7},
          _x{1,8},x{2,8},_x{3,8},_x{4,8},x{7,8},_x{8,8},
          _x{1,9},x{2,9},x{3,9},x{5,9},_x{6,9},x{8,9},_x{9,9},
           x{2,10},_x{3,10},x{4,10},_x{6,10},_x{7,10},_x{9,10};
}
```

8.4 SAMDS P systems simulation algorithm

Simulators for computational models usually are very interesting assistants to help in design tasks and the formal verification of solutions to decision problems defined in such models. Simulators work with an inference engine implemented through an algorithm capturing the semantics of the corresponding model. When a family of recognizer P systems is used to solve a decision problem, each member of the family processing an instance is confluent, in the sense that all computations for a given input must generate the same output. Consequently, a simulation algorithm for recognizer P systems only has to reproduce one possible computation of the simulated P system.

The simulation algorithm described below generates one possible computation of a P system in the class **CDC** or **CSC**, that we collectively denominate SAMDS P systems. We denote $(u, in)_i$ ($(u, out)_i$ or $(u\,,\,out\,;\,v\,,\,in)_i$, respectively) instead of $(u\,,\,in) \in \mathcal{R}_i$ ($(u\,,\,out) \in \mathcal{R}_i$ or $(u\,,\,out\,;\,v\,,\,in) \in \mathcal{R}_i$, respectively) for simplicity.

```
I. Initialization

  1. Let $C_0$ be the initial configuration with $q$ membranes
     denoted by $m_1, \ldots, m_q$
  2. Let $m_0$ be a virtual membrane with label $0$ representing
     the environment, where all initial objects have infinite multiplicity
  3. Let $R_{sel} \leftarrow \{\}$ be a set of tuples containing the selected rules
     to be executed at each computation step, the identifiers
     of the membranes involved and the number of times that each rule
     will be executed (omitted if 1)
  4. Let $C_t \leftarrow C_0$ be the current configuration
  5. Let $SetSymAnt \leftarrow \{\}$ be the set of membranes that are executing
     a symport/antiport rule in the current configuration
  6. Let $SetDivSep \leftarrow \{\}$ be the set of membranes that are executing
     a division or separation rule in the current configuration

II. Selection of symport/antiport rules

  1. For each membrane $m_i \in C_t$ with label $i$ do

     (a) For each send-in symport rule $(u\,,\,in)_i$ do
           • Let $m_k$ be the parent membrane of $m_i$
           • Let $M$ be the greatest number such that the multiset of $m_k$
             contains $M$ copies of the multiset $u$
           • Remove $M$ copies of $u$ from the multiset of $m_k$
           • Add $\langle m_i, m_k, (u\,,\,in)_i, M \rangle$ to $R_{sel}$
           • Add $m_i$ to $SetSymAnt$
     (b) For each send-out symport rule $(u\,,\,out)_i$ do
           • Let $m_k$ be the parent membrane of $m_i$
           • Let $M$ be the greatest number such that the multiset of $m_i$
             contains $M$ copies of the multiset $u$
           • Remove $M$ copies of $u$ from the multiset of $m_i$
           • Add $\langle m_i, m_k, (u\,,\,out)_i, M \rangle$ to $R_{sel}$
           • Add $m_i$ to $SetSymAnt$
     (c) For each antiport rule $(u\,,\,out\,;\,v\,,\,in)_i$ do
           • Let $m_k$ be the parent membrane of $m_i$
           • Let $M$ be the greatest number such that the multiset of $m_i$
             contains $M$ copies of the multiset $u$
           • Let $N$ be the greatest number (considering that $N \leq M$) such
             that the multiset of $m_k$ contains $N$ copies of the multiset $v$
```

```
          • Remove $N$ copies of $u$ from the multiset of $m_i$
          • Remove $N$ copies of $v$ from the multiset of $m_k$
          • Add $\langle m_i, m_k, (u\,,\,out\,;\,v\,,\,in)_i, N\rangle$ to $R_{sel}$
          • Add $m_i$ to $SetSymAnt$

III. Selection of division and separation rules

  1. For each membrane $m_i$ at configuration $C_t$ with label $i$ not contained in
     $SetSymAnt$ and not contained in $SetDivSep$ do
     (a) If the simulated P system is in the class CDC, then
         i. For each division rule $[a]_i \to [b]_i[c]_i$ do
            • If $a$ is contained in the multiset of $m_i$, then
              – Remove one instance of $a$ from the multiset of $m_i$
              – Add $\langle m_i, [a]_i \to [b]_i[c]_i\rangle$ to $R_{sel}$
              – Add $m_i$ to $SetDivSep$
     (b) If the simulated P system is in the class CSC, then
         i. For each separation rule $[a]_i \to [\Gamma_1]_i[\Gamma_2]_i$ do
            • If $a$ is contained in the multiset of $m_i$, then
              – Remove one instance of $a$ from the multiset of $m_i$
              – Add $\langle m_i, [a]_i \to [\Gamma_1]_i[\Gamma_2]_i\rangle$ to $R_{sel}$
              – Add $m_i$ to $SetDivSep$

IV. Execution of rules

  1. For each tuple $\langle m_i, m_k, (u\,,\,in)_i, M\rangle$ from $R_{sel}$ do
     (a) Add $M$ copies of $u$ to the multiset of $m_i$
  2. For each tuple $\langle m_i, m_k, (u\,,\,out)_i, M\rangle$ from $R_{sel}$ do
     (a) Add $M$ copies of $u$ to the multiset of $m_k$
  3. For each tuple $\langle m_i, m_k, (u\,,\,out\,;\,v\,,\,in)_i, N\rangle$ from $R_{sel}$ do
     (a) Add $N$ copies of $v$ to the multiset of $m_i$
     (b) Add $N$ copies of $u$ to the multiset of $m_k$
  4. For each tuple $\langle m_i, [a]_i \to [b]_i[c]_i\rangle$ from $R_{sel}$ do
     (a) Create a new cell $m'_i$ with label $i$ and empty multiset
     (b) Copy in the multiset of $m'_i$ the objects that are contained
         in the multiset of $m_i$
     (c) Add one instance of $b$ to the multiset of $m_i$
     (d) Add one instance of $c$ to the multiset of $m'_i$
  5. For each tuple $\langle m_i, [a]_i \to [\Gamma_1]_i[\Gamma_2]_i\rangle$ from $R_{sel}$ do
     (a) Create a new cell $m'_i$ with label $i$ and empty multiset
     (b) Copy in the multiset of $m'_i$ the objects of $\Gamma_2$ that are contained
         in the multiset of $m_i$
```

```
    (c) Remove from the multiset of $m_i$ the objects of $\Gamma_2$

V. Ending

  1. If $R_{sel} \neq \emptyset$, then
      • Let $C_{t+1} \leftarrow C_t$
      • Let $R_{sel} \leftarrow \{\}$
      • Let $SetSymAnt \leftarrow \{\}$
      • Let $SetDivSep \leftarrow \{\}$
      • Goto II
  2. End
```

A total order in the finite set of rules is considered. At each computation step, for each membrane, the algorithm processes the symport/antiport rules first and then goes through the membrane division or membrane separation rules (depending on the kind of P system specified) for those membranes not executing any of the previously processed symport/antiport rules. This strategy is motivated by the intuition that division and separation rules add more descriptive complexity than symport/antiport rules, so we give them "lower priority".

8.5 Performance results

In this Section performance results of the developed simulator in reference to concrete instances of the solutions of **SAT** in **CDC(3)** and **CSC(3)**, respectively, are shown. To obtain such results, several P systems of the families solving different instances of the SAT problem in **CDC(3)** (see [90]) and **CSC(3)** (see Chapter 7) have been simulated. Simulation results are shown in Tables 8.1 and 8.2. We denote $\neg x$ by $\overline{x}$, $l_1 \vee l_2$ by $l_1 + l_2$ and $C_1 \wedge C_2$ by $C_1 \cdot C_2$, where x is a propositional variable, l_i are literals and C_j are clauses. Execution times have been calculated averaging the times obtained after performing three simulations for each instance. Simulation platform features are described in Section 6.7.

Clearly, the simulation time of the solution to the SAT problem by using a family of P systems in **CSC**(3) takes much more time than the simulation of the solution by P systems in **CDC**(3). This can be explained by the following fact. In the P systems of **CDC**(3), a replication of objects by means of division rules takes place. Nevertheless, in the P systems of **CSC**(3), there is no replication of objects, but a distribution of them, consequently, in order to generate an exponential amount of some objects, it is necessary to use the skin membrane, interacting with the environment by using antiport rules with length 3 (in a computation step an object is released into the environment and, simultaneously, two objects enter the system).

It is worth pointing out that the developed simulator is based on sequential technology, so the inherent parallelism of P systems cannot be exploited. This limitation cannot be overcome because a real implementation of P systems does not exist, but some performance improvements can be achieved by means of some parallel architectures and programming models, such as *GPGPU* (General-Purpose Computing on Graphics Processing Units). Specifically, the technology *NVIDIA GPU* with *CUDA* (Compute Unified Device Architecture) has been considered for the development of parallel simulators of P systems such as tissue P systems with cell division [95]. It is still of interest to develop parallel architectures for simulating P systems in the classes **CDC** and **CSC**.

Table 8.1: Satisfiability and simulation time for instances solved in **CDC(3)**

Formula	**n**	**m**	**SAT**	**Time (s)**
$(\bar{x}_1 + \bar{x}_2) \cdot x_1 \cdot x_2$	2	3	F	0,121
$(\bar{x}_1 + \bar{x}_2) \cdot x_2 \cdot (\bar{x}_1 + x_2)$	2	3	T	0,099
$(x_1 + x_2) \cdot (x_1 + x_2 + \bar{x}_3) \cdot \bar{x}_1 \cdot \bar{x}_2$	3	4	F	0,219
$(\bar{x}_1 + x_2) \cdot \bar{x}_1 \cdot x_3 \cdot (\bar{x}_1 + x_3)$	3	4	T	0,201
$(x_1 + x_4) \cdot (x_1 + \bar{x}_4) \cdot x_3 \cdot (x_2 + \bar{x}_3 + x_4) \cdot \bar{x}_1$	4	5	F	0,315
$(x_3 + \bar{x}_4) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + x_4) \cdot (x_1 + x_2) \cdot (\bar{x}_1 + x_2 + x_3 + x_4) \cdot (\bar{x}_1 + x_3)$	4	5	T	0,325
$(x_1 + \bar{x}_2 + x_3 + x_5) \cdot (\bar{x}_1 + x_4) \cdot (\bar{x}_2 + \bar{x}_4) \cdot x_4 \cdot x_2 \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + x_4)$	5	6	F	0,570
$(x_3 + x_4) \cdot (x_4 + \bar{x}_5) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4) \cdot (x_1 + \bar{x}_2 + x_4) \cdot (x_1 + \bar{x}_3 + x_4) \cdot (x_3 + x_5)$	5	6	T	0,564
$(x_3 + x_5 + x_6) \cdot (x_3 + \bar{x}_4 + x_5 + \bar{x}_6) \cdot \bar{x}_3 \cdot \bar{x}_6 \cdot (x_1 + \bar{x}_2 + \bar{x}_3 + x_5 + x_6) \cdot (x_1 + x_4 + x_5) \cdot (\bar{x}_5 + x_6)$	6	7	F	0,932
$(\bar{x}_1 + \bar{x}_2 + x_5) \cdot (x_2 + x_3) \cdot (x_3 + \bar{x}_5 + \bar{x}_6) \cdot (\bar{x}_1 + x_2 + \bar{x}_3 + x_4 + x_5 + x_6) \cdot (\bar{x}_2 + \bar{x}_3) \cdot (x_2 + x_3 + x_6) \cdot (x_1 + \bar{x}_2 + x_3 + x_4 + x_5 + x_6)$	6	7	T	1,017
$(\bar{x}_5 + \bar{x}_6 + \bar{x}_7) \cdot (x_3 + \bar{x}_4 + x_7) \cdot (\bar{x}_1 + x_3 + x_5 + x_6 + \bar{x}_7) \cdot (x_1 + x_3 + \bar{x}_5 + x_6 + x_7) \cdot (x_2 + x_6) \cdot (x_2 + \bar{x}_6) \cdot \bar{x}_2 \cdot (x_2 + x_3 + x_4 + \bar{x}_5 + x_7)$	7	8	F	1,974
$(\bar{x}_2 + x_5 + x_6 + x_7) \cdot (x_2 + \bar{x}_4 + \bar{x}_5 + \bar{x}_7) \cdot (x_1 + x_2 + \bar{x}_3 + \bar{x}_6 + x_7) \cdot (x_1 + x_2 + x_3 + \bar{x}_5 + x_6 + \bar{x}_7) \cdot (\bar{x}_3 + \bar{x}_5 + x_6 + \bar{x}_7) \cdot (x_1 + x_2 + \bar{x}_3 + \bar{x}_7) \cdot (\bar{x}_1 + x_2 + \bar{x}_4 + \bar{x}_6) \cdot (x_3 + x_5 + x_6 + \bar{x}_7)$	7	8	T	1,804
$(x_3 + x_4 + \bar{x}_6 + \bar{x}_8) \cdot (x_6 + \bar{x}_7) \cdot (\bar{x}_2 + x_3 + \bar{x}_4 + x_5 + x_8) \cdot x_7 \cdot (x_1 + \bar{x}_2 + x_5 + \bar{x}_7 + \bar{x}_8) \cdot (x_2 + x_7 + x_8) \cdot (\bar{x}_6 + \bar{x}_7) \cdot (x_1 + x_5 + \bar{x}_8) \cdot (x_1 + \bar{x}_4 + x_5 + \bar{x}_6 + x_7)$	8	9	F	3,530
$(x_1 + \bar{x}_5 + \bar{x}_6 + \bar{x}_7 + \bar{x}_8) \cdot (x_2 + x_3 + x_4 + \bar{x}_6 + \bar{x}_7 + x_8) \cdot (x_3 + x_4 + \bar{x}_5 + \bar{x}_6 + \bar{x}_7 + \bar{x}_8) \cdot (\bar{x}_1 + \bar{x}_3 + \bar{x}_4 + \bar{x}_5 + x_6 + \bar{x}_7 + x_8) \cdot (\bar{x}_3 + \bar{x}_7) \cdot (x_4 + x_5 + \bar{x}_7) \cdot (x_1 + x_3 + \bar{x}_4) \cdot (x_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4 + \bar{x}_5 + \bar{x}_6 + \bar{x}_7) \cdot (x_4 + \bar{x}_5 + \bar{x}_6 + x_7 + \bar{x}_8)$	8	9	T	3,632

$(\bar{x}_2+\bar{x}_3+x_5+x_7)\cdot(x_2+x_5+x_6+x_7+x_9)\cdot(\bar{x}_3+x_5+x_7+x_8)\cdot(x_1+\bar{x}_4+\bar{x}_5+x_6+x_8)\cdot(\bar{x}_2+x_3+x_5+x_7+x_8+\bar{x}_9)\cdot(\bar{x}_2+\bar{x}_4+x_7+x_9)\cdot(\bar{x}_2+x_4+x_6+x_9)\cdot x_1\cdot x_5\cdot(\bar{x}_1+\bar{x}_5)$	9	10	F	8,262
$(x_3+x_8)\cdot(x_1+\bar{x}_2+x_5+\bar{x}_6+x_9)\cdot(x_3+x_6+x_9)\cdot(x_3+x_5+\bar{x}_6+\bar{x}_8)\cdot(x_1+x_2+\bar{x}_5+x_7+\bar{x}_8+\bar{x}_9)\cdot(\bar{x}_1+x_2+\bar{x}_4+x_5+\bar{x}_6+\bar{x}_7+x_9)\cdot(x_1+x_2+x_4+\bar{x}_6+x_8+\bar{x}_9)\cdot(\bar{x}_1+x_2+\bar{x}_3+\bar{x}_4+x_7+\bar{x}_8)\cdot(\bar{x}_1+x_2+x_3+x_5+\bar{x}_6+x_8+\bar{x}_9)\cdot(x_2+\bar{x}_3+x_4+\bar{x}_6+\bar{x}_7+\bar{x}_9)$	9	10	T	7,993

Table 8.2: Satisfiability and simulation time for instances solved in **CSC(3)**

Formula	**n**	**m**	**SAT**	**Time (s)**
$(\bar{x}_1+\bar{x}_2)\cdot x_1\cdot x_2$	2	3	F	0,233
$(\bar{x}_1+\bar{x}_2)\cdot x_2\cdot(\bar{x}_1+x_2)$	2	3	T	0,224
$(x_1+x_2)\cdot(x_1+x_2+\bar{x}_3)\cdot\bar{x}_1\cdot\bar{x}_2$	3	4	F	0,491
$(\bar{x}_1+x_2)\cdot\bar{x}_1\cdot x_3\cdot(\bar{x}_1+x_3)$	3	4	T	0,487
$(x_1+x_4)\cdot(x_1+\bar{x}_4)\cdot x_3\cdot(x_2+\bar{x}_3+x_4)\cdot\bar{x}_1$	4	5	F	0,827
$(x_3+\bar{x}_4)\cdot(\bar{x}_1+x_2+\bar{x}_3+x_4)\cdot(x_1+x_2)\cdot(\bar{x}_1+x_2+x_3+x_4)\cdot(\bar{x}_1+x_3)$	4	5	T	0,981
$(x_1+\bar{x}_2+x_3+x_5)\cdot(\bar{x}_1+x_4)\cdot(\bar{x}_2+\bar{x}_4)\cdot x_4\cdot x_2\cdot(\bar{x}_1+x_2+\bar{x}_3+x_4)$	5	6	F	2,369
$(x_3+x_4)\cdot(x_4+\bar{x}_5)\cdot(\bar{x}_1+x_2+\bar{x}_3+\bar{x}_4)\cdot(x_1+\bar{x}_2+x_4)\cdot(x_1+\bar{x}_3+x_4)\cdot(x_3+x_5)$	5	6	T	2,312
$(x_3+x_5+x_6)\cdot(x_3+\bar{x}_4+x_5+\bar{x}_6)\cdot\bar{x}_3\cdot\bar{x}_6\cdot(x_1+\bar{x}_2+\bar{x}_3+x_5+x_6)\cdot(x_1+x_4+x_5)\cdot(\bar{x}_5+x_6)$	6	7	F	4,877
$(\bar{x}_1+\bar{x}_2+x_5)\cdot(x_2+x_3)\cdot(x_3+\bar{x}_5+\bar{x}_6)\cdot(\bar{x}_1+x_2+\bar{x}_3+x_4+x_5+x_6)\cdot(\bar{x}_2+\bar{x}_3)\cdot(x_2+x_3+x_6)\cdot(x_1+\bar{x}_2+x_3+x_4+x_5+x_6)$	6	7	T	4,195
$(\bar{x}_5+\bar{x}_6+\bar{x}_7)\cdot(x_3+\bar{x}_4+x_7)\cdot(\bar{x}_1+x_3+x_5+x_6+\bar{x}_7)\cdot(x_1+x_3+\bar{x}_5+x_6+x_7)\cdot(x_2+x_6)\cdot(x_2+\bar{x}_6)\cdot\bar{x}_2\cdot(x_2+x_3+x_4+\bar{x}_5+x_7)$	7	8	F	10,320
$(\bar{x}_2+x_5+x_6+x_7)\cdot(x_2+\bar{x}_4+\bar{x}_5+\bar{x}_7)\cdot(x_1+x_2+\bar{x}_3+\bar{x}_6+x_7)\cdot(x_1+x_2+x_3+\bar{x}_5+x_6+\bar{x}_7)\cdot(\bar{x}_3+\bar{x}_5+x_6+\bar{x}_7)\cdot(x_1+x_2+\bar{x}_3+\bar{x}_7)\cdot(\bar{x}_1+x_2+\bar{x}_4+\bar{x}_6)\cdot(x_3+x_5+x_6+\bar{x}_7)$	7	8	T	8,862
$(x_3+x_4+\bar{x}_6+\bar{x}_8)\cdot(x_6+\bar{x}_7)\cdot(\bar{x}_2+x_3+\bar{x}_4+x_5+x_8)\cdot x_7\cdot(x_1+\bar{x}_2+x_5+\bar{x}_7+\bar{x}_8)\cdot(x_2+x_7+x_8)\cdot(\bar{x}_6+\bar{x}_7)\cdot(x_1+x_5+\bar{x}_8)\cdot(x_1+\bar{x}_4+x_5+\bar{x}_6+x_7)$	8	9	F	16,364
$(x_1+\bar{x}_5+\bar{x}_6+\bar{x}_7+\bar{x}_8)\cdot(x_2+x_3+x_4+\bar{x}_6+\bar{x}_7+x_8)\cdot(x_3+x_4+\bar{x}_5+\bar{x}_6+\bar{x}_7+\bar{x}_8)\cdot(\bar{x}_1+\bar{x}_3+\bar{x}_4+\bar{x}_5+x_6+\bar{x}_7+x_8)\cdot(\bar{x}_3+\bar{x}_7)\cdot(x_4+x_5+\bar{x}_7)\cdot(x_1+x_3+\bar{x}_4)\cdot(x_1+\bar{x}_2+\bar{x}_3+\bar{x}_4+\bar{x}_5+\bar{x}_6+\bar{x}_7)\cdot(x_4+\bar{x}_5+\bar{x}_6+x_7+\bar{x}_8)$	8	9	T	18,856
$(\bar{x}_2+\bar{x}_3+x_5+x_7)\cdot(x_2+x_5+x_6+x_7+x_9)\cdot(\bar{x}_3+x_5+x_7+x_8)\cdot(x_1+\bar{x}_4+\bar{x}_5+x_6+x_8)\cdot(\bar{x}_2+x_3+x_5+x_7+x_8+\bar{x}_9)\cdot(\bar{x}_2+\bar{x}_4+x_7+x_9)\cdot(\bar{x}_2+x_4+x_6+x_9)\cdot x_1\cdot x_5\cdot(\bar{x}_1+\bar{x}_5)$	9	10	F	34,669
$(x_3+x_8)\cdot(x_1+\bar{x}_2+x_5+\bar{x}_6+x_9)\cdot(x_3+x_6+x_9)\cdot(x_3+x_5+\bar{x}_6+\bar{x}_8)\cdot(x_1+x_2+\bar{x}_5+x_7+\bar{x}_8+\bar{x}_9)\cdot(\bar{x}_1+x_2+\bar{x}_4+x_5+\bar{x}_6+\bar{x}_7+x_9)\cdot(x_1+x_2+x_4+\bar{x}_6+x_8+\bar{x}_9)\cdot(\bar{x}_1+x_2+\bar{x}_3+\bar{x}_4+x_7+\bar{x}_8)\cdot(\bar{x}_1+x_2+x_3+x_5+\bar{x}_6+x_8+\bar{x}_9)\cdot(x_2+\bar{x}_3+x_4+\bar{x}_6+\bar{x}_7+\bar{x}_9)$	9	10	T	36,450

9

Extending the SN P systems framework

9.1 Introduction

SN P systems have become a *hot topic* in Membrane Computing [128, 211], with a focus on the use this kind of neural-like systems to solve engineering problems. In this way, SN P systems variants have appeared in quick succession, along with the corresponding research on their computational properties and possible practical applications. This *research boom* contrasts with the scarcity of SN P systems simulation tools. Prior to the work object of this dissertation, to the best knowledge of the author only the following SN P systems simulators have been made publicly available[1]:

- *Ramírez-Martínez et al. Simulator (2007).* Introduced in [159], it is the first developed SN P systems simulator. It provides a GUI that allows defining the P system in a graphical way (definition in text files is also supported) as well as performing the simulation, which result is a collection of images showing the transition diagram for each simulation step.

[1] it is also worth mentioning the work by *Padmavati et al.* in [101] which describes an algorithm to translate SN P systems without delays into Petri Nets, therefore allowing their simulation with Petri Nets related software tools such as PNet Lab [190].

- *Cabarle et al. Simulator (2012).* Introduced in [15] and improved in [16], able to simulate SN P systems without delays on CUDA architectures. Its simulation algorithm uses a matrix representation of the model [210] that allows a straightforward implementation on the GPU.

Due to this unbalanced relation between research interest and availability of software applications, developing a simulator tool to support a wide range of SN P systems variants has become a key part of the work object of this dissertation. Development of this tool has involved an extensive research on SN P systems, focusing on correctly capturing their semantics and defining the corresponding simulation algorithms. Three have been the *main design goals* when constructing the application: (1) to support as many SN P systems variants as possible; (2) to define a *unified* simulation algorithm able to cover the features of all the considered variants, which greatly favours the simplicity and reusability of the developed code; and (3) to make the simulation tool able to run on virtually any platform.

It is worth pointing out that, although the use of GPUs have shown that SN P systems can be efficiently simulated on parallel devices, their related programming models impose some constraints that make difficult for GPUs based simulators to easily capture the whole syntactic ingredients participating in the definition of the SN P systems models, such as delays and regular expressions, as well as to keep in pace with the aforementioned research boom that SN P systems are experiencing. Moreover, not all the simulator potential users may have access to HPC environments. In consequence, according to the design goals (1) and (3) stated above, at the expense of sacrificing efficiency for expressivity, other alternatives involving sequential approaches must be considered. At this respect, P–Lingua is a natural choice as it offers the high flexibility of the Java programming language as well as a general acceptance within the Membrane Computing community.

On the other hand, as a result of the research conducted on SN P systems, two achievements involving theoretical aspects of SN P systems have been reached: (1) designing a simulation algorithm for Limited Asynchronous SN P systems (LASNPS, for short) by re-interpreting this variant specification; and (2) defining a *brand new variant* of SN P systems incorporating a novel kind of astrocytes, called *functional astrocytes*, giving place to the eponymous *SN P systems with Functional Astrocytes (SNPSFA, for short).*

Therefore, an extension of the SN P systems framework, within the Membrane Computing paradigm, has been produced as a result of all this work, involving both theoretical (variants, simulation algorithms) and practical (simulation tools) aspects. Such extension involves producing a general purpose

SN P systems simulator, which has been included into pLinguaCore library. SN P systems variants incorporating the following ingredients fall into this extension (see Chapter 4):

- ⋆ extended spiking rules
- ⋆ neuron division and budding rules
- ⋆ sequential modes
- ⋆ asynchronous modes
- ⋆ hybrid astrocytes
- ⋆ functional astrocytes
- ⋆ anti-spikes

In this Chapter the aforementioned extension of the SN P systems framework is discussed. Section 9.2 presents the simulation algorithm for LASNPS, while Section 9.3 introduces the SNPSFA variant, along with its corresponding simulation algorithm and some example models. Section 9.4 details P–Lingua syntax for specifying the considered SN P systems variants. Section 9.5 presents the unified simulation algorithm implemented into pLinguaCore library. Finally, Section 9.6 provides some P–Lingua example programs corresponding to the addressed SN P systems variants. Before going on with this Chapter, the reader is advised to review contents of Chapters 4 and 6 for the shake of a better understanding.

9.2 A simulation algorithm for Limited Asynchronous SN P systems

Recalling from Section 4.3.2.3, in LASNPS a global bound $b \geq 2$ (imposed on all rules) is specified in such a way that if one (and only one) 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 now the rule non-applicable, then the computation continues in the new circumstance (maybe other rules are enabled now). If more than one rule is applicable, the neuron non-deterministically chooses and fires one of them in the interval t to $t + b - 1$. In LASNPS, a configuration is described by the number of spikes present in each neuron, the number of time units for neurons to become open as well as the time that has elapsed for each rule since it became applicable.

Since the same global bound b applies to all the rules, it is possible to easily simulate the limited asynchronous firing mechanism by adding a *virtual* syntactic ingredient to each neuron σ_i called *bound counter*, and denoted by b_i. This counter takes value in $[-1, b-1]$ (in the initial configuration $b_i = 0$ for all neurons). The bound counter controls (a) if the neuron can hold applicable rules execution in relation to the global bound b; and (b) if the neuron must check for new applicable rules.

For each time instant t the firing mechanism can be reproduced by the following simulation algorithm:

1. If σ_i is executing a rule at step t, then <u>GOTO</u> $t + 1$.
2. If σ_i received spikes in the previous step, then: $b_i \leftarrow 0$.
3. If σ_i is not executing a rule, then σ_i decides if it has to check for new applicable rules:
    - If $b_i = -1$, then no new applicable rules exist.
    - If $b_i > -1$, then new applicable rules may exist.
4. If no new applicable rules exist, then <u>GOTO</u> $t + 1$.
5. If new applicable rules may exist, then σ_i checks for them.
6. If no new applicable rules exist, then:
    - $b_i \leftarrow -1$.
    - <u>GOTO</u> $t + 1$.
7. If applicable rules exist, then:
    - b_i is updated:
        - If $b_i = 0$, then $b_i \leftarrow b - 1$.
        - If $b_i > 0$, then $b_i \leftarrow b_i - 1$.
    - σ_i non-deterministically selects an applicable rule.
    - σ_i decides whether to fire or not:
        - If $b_i > 0$, then σ_i non-deterministically chooses whether to fire or not the selected rule.
        - If $b_i = 0$, then σ_i must fire the selected rule.
    - If neuron σ_i does not have to fire, then <u>GOTO</u> $t + 1$.
    - If neuron σ_i has to fire, then:
        - $b_i \leftarrow 0$.
        - σ_i fires the selected rule.

It is worth pointing out that under the assumption of that a bound counter is associated to each neuron, a configuration may be described by the number of spikes present in each neuron, the number of time units for each neuron to become open as well as the number of time units that neuron can postpone the application of one of its applicable rules, that is, the current value of the neuron bound counter.

9.3 SN P systems with Functional Astrocytes

In this Section, a new variant of SN P systems incorporating *functional* astrocytes is introduced. This variant is called SN P systems with functional astrocytes (SNPSFA, for short). Astrocytes are glial cells connected to one or more synapses that can sense the whole spike traffic passing along their neighbouring synapses and, eventually, modify it. Biological background related to astrocytes can be consulted in Section 4.3.5.

9.3.1 Antecedents

The first SN P systems variant containing astrocytes was introduced in [13]. The model presented there, pretty complex, was then simplified in [152], in which only inhibitory astrocytes were considered. This simplification was revised again in [123], where *hybrid* astrocytes were introduced. Behaviour of an astrocyte of this kind, inhibitory or excitatory, relied on the amount of spikes passing on its neighbouring synapses, in relation to a given threshold associated to it. This kind of systems were also discussed in Section 4.3.5.

SN P systems with functional astrocytes take inspiration from the original model defined in [13]. In the SNPSFA variant, new ingredients are introduced in order to turn astrocytes into *function computation devices*. They operate in the following general way. A set of pairs (threshold, function) is associated with each astrocyte. Existing spike traffic measured on distinguished neighbouring *control synapses* attached to the astrocyte is matched against the thresholds until one of them is selected. Subsequently, the associated function to the matched threshold is selected. At this point, that function is computed taking as arguments the amounts of spikes measured on distinguished neighbouring *operand synapses* attached to the astrocyte. Finally, the result of the function computation is sent through a distinguished *output operand synapse*. With this mechanism any computable partial function between natural numbers can be computed in a single computation step. Moreover, this kind of astrocytes eases the design of machines that calculate functions, as astrocytes can be viewed as "macros".

9.3.2 Formal definition

Definition 9.1. *A SN P system with functional astrocytes of degree* (m, l), *with* $m \geq 1$, $l \geq 1$, *is a tuple of the form:*

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_m, ast_1, ast_2, \ldots, ast_l, syn, in, out),$$

where:

1. $O, \sigma_1, \sigma_2, \ldots, \sigma_m, syn, in, out$ *are defined as in classic SN P systems.*

2. $ast_1, ast_2, \ldots, ast_l$ *are astrocytes, with* ast_j $(1 \leq j \leq l)$ *of the form*

$$ast_j = (syn^o_j, syn^c_j, \omega_j, T_j, F_j, p_j(0), \gamma_j),$$

where:

- $syn^o_j = \{s^o_{j,1}, \ldots, s^o_{j,r_j}\} \subseteq syn, r_j \geq 1$, *is the astrocyte finite set of* operand synapses, *ordered by a lexicographical order imposed on* syn^o_j;
- $syn^c_j = \{s^c_{j,1}, \ldots, s^c_{j,q_j}\} \subseteq syn, q_j \geq 0$, *is the astrocyte finite set of* control synapses*;*
- $\omega_j \in \{true, false\}$ *is the astrocyte control-as-operand flag;*
- $T_j = \{T_{j,1}, \ldots, T_{j,k_j}\}, k_j \geq 1$, *is the astrocyte finite set of thresholds, such that,* $T_{j,\alpha} \in \mathbb{N}$, $(1 \leq \alpha \leq k_j)$ *and* $T_{j,1} < \cdots < T_{j,k_j}$*;*
- $F_j = \{f_{j,1}, \ldots, f_{j,k_j}\}$ *is the astrocyte finite* multiset *(some elements in* F_j *can be the same) of natural functions such that for each* α $(1 \leq \alpha \leq k_j)$*:*
  - $f_{j,\alpha}$ *is a computable function between natural numbers;*
  - *if* $\omega_j = true$ *then* $f_{j,\alpha}$ *is a unary function;*
  - *if* $\omega_j = false$ *and* $r_j = 1$ *then* $f_{j,\alpha}$ *is a unary constant function;*
  - *if* $\omega_j = false$ *and* $r_j > 1$ *then* $f_{j,\alpha}$ *has arity* $r_j - 1$*;*
- $p_j(0) \in \mathbb{N}$ *is the astrocyte initial potential;*
- $\gamma_j \in \{true, false\}$ *is the astrocyte potential update flag;*

Functional astrocytes are graphically represented as diamond-shaped figures, with ingoing synapses coming from the control synapses and outgoing synapses connected to the operand synapses. An example is shown in Fig. 9.1.

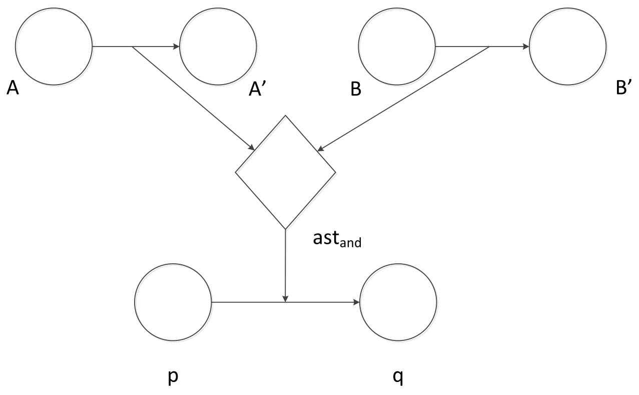


Figure 9.1: A functional astrocyte.

9.3.3 Semantics and associated simulation algorithm

Semantics of SNPSFA follows from the SN P systems classic model (see Section 4.2), but incorporating functional astrocytes behaviour. A functional astrocyte can sense the spike traffic passing along its controlled synapses, both control and operand ones, and eventually modify it. When neurons spike, spikes reach the target neurons going along the synapses unless they are intercepted by astrocytes.

Astrocytes behaviour can be modelled by the following simulation algorithm:

1. Let ast_j be an astrocyte, such that, at instant t, the following holds:
   - A total amount of k spikes are passing along its control synapses;
   - $x_1, x_2, \dots, x_{r_j}$ spikes are passing along its operand synapses;
   - Potential of ast_j is p;
2. Value $s = k + p$ is computed.
3. Value h satisfying that $s \in [T_{j,h}, T_{j,h+1})$ is computed out of s, with the constraints that if $s < T_{j,1}$ then $h = 1$, and if $s > T_{j,k_j}$ then $h = k_j$.
4. Value s' is computed as follows: if $\omega_j = true$ then $s' = f_{j,h}(s)$ directly; otherwise, two cases are considered:

- If $|syn_j^o| = 1$ then $s' = f_{j,h}(0)$.
- If $|syn_j^o| > 1$ then $s' = f_{j,h}(x_1, x_2, \ldots, x_{r_j-1})$.

5. For control synapses, ast_j shows an excitatory influence, all spikes are allowed to survive and reach their destination neurons.

6. For operand synapses, ast_j shows both an inhibitory and excitatory influence, in the following way:

   - Synapses $s_{j,1}^o, \ldots, s_{j,r_j-1}$ are applied an inhibitory influence, all the spikes passing along them are removed from the system.
   - Synapse s_{j,r_j} is applied an excitatory influence, s' spikes are added to the spikes passing along the synapse, and the resulting amount of spikes is allowed to reach their destination.

7. If $\gamma_j = true$ then astrocyte potential in $t+1$ will be incremented in s units. Otherwise, the astrocyte potential does not change.

On the other hand, the concepts of configuration, transition step and computation can be defined in a similar way to classic SN P systems.

9.3.4 Some example models

In what follows, some example models showing SNPSFA capabilities are shown.

Excitatory and Inhibitory Astrocytes

First couple of examples shows how to implement excitatory and inhibitory astrocytes respectively, with a given threshold k. Implementation involves defining two functions: $f(x)$, which is the identically zero function of arity one, and $g(x)$.

Excitatory astrocyte, ast_{exc}, is depicted in the Fig. 9.2 with its formal specification being:

$$ast_{exc} = (\{(p', q)\}, \{(p, q')\}, true, \{0, k\}, \{f(x), g(x)\}, 0, false)$$

and its working equation, assuming that α spikes pass through synapse (p, q') at a given instant t, being:

$$ast_{exc}(\alpha, t) = \begin{cases} f(\alpha) = 0 & \text{if} \quad 0 \leq \alpha < k \\ g(\alpha) & \text{if} \quad \alpha \geq k \end{cases}$$

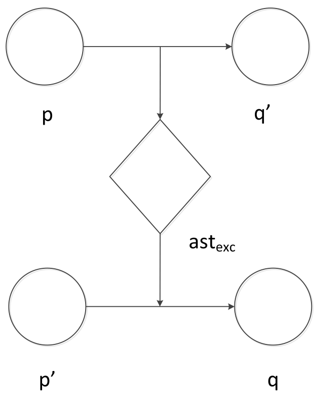


Figure 9.2: Excitatory astrocyte.

Inhibitory astrocyte, ast_{inh}, is structurally identical to ast_{exc}, with its formal specification being:

$$ast_{inh} = (\{(p', q)\}, \{(p, q')\}, true, \{0, k+1\}, \{g(x), f(x)\}, 0, false)$$

and its working equation, assuming that α spikes pass through synapse (p, q') at a given instant t, being:

$$ast_{inh}(\alpha, t) = \begin{cases} g(\alpha) & \text{if} \quad 0 \leq \alpha \leq k \\ f(\alpha) = 0 & \text{if} \quad \alpha \geq k+1 \end{cases}$$

Logic Gates

Second couple of examples shows how to implement logical gates, concretely AND-gates and NAND-gates respectively. Implementation involves defining two functions, $f(x)$ and $g(x)$, both of them unary constant functions, which associates the 0 and 1 natural values respectively for every $x \in \mathbb{N}$.

AND-gate astrocyte, ast_{and}, is depicted in the Fig. 9.3 with its formal specification being:

$$ast_{and} = (\{(p, q)\}, \{(A, A'), (B, B')\}, false, \{1, 2\}, \{f(x), g(x)\}, 0, false)$$

and its working equation, assuming that $\alpha, 0 \leq \alpha \leq 2$ spikes in total pass through synapses (A, A') and (B, B') at a given instant t, being:

$$ast_{and}(\alpha, t) = \begin{cases} f(0) = 0 & \text{if} \quad 0 \leq \alpha \leq 1 \\ g(0) = 1 & \text{if} \quad \alpha = 2 \end{cases}$$

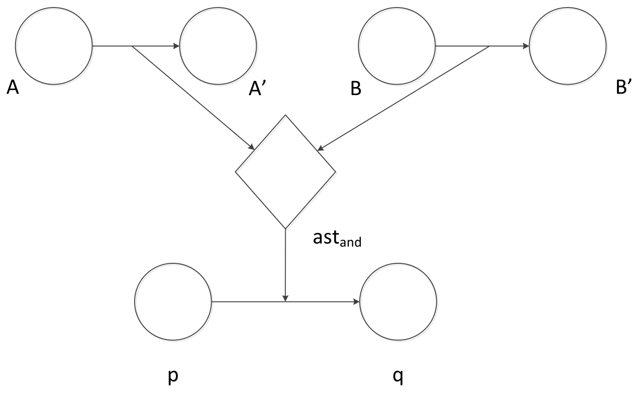


Figure 9.3: AND-gate astrocyte.

NAND-gate astrocyte, ast_{nand}, is structurally identical to ast_{and}, with its formal specification being:

$$ast_{nand} = (\{(p, q)\}, \{(A, A'), (B, B')\}, false, \{1, 2\}, \{g(x), f(x)\}, 0, false)$$

and its working equation, assuming that $\alpha, 0 \leq \alpha \leq 2$ spikes in total pass through synapses (A, A') and (B, B') at a given instant t, being:

$$ast_{nand}(\alpha, t) = \begin{cases} g(0) = 1 & \text{if} \quad 0 \leq \alpha \leq 1 \\ f(0) = 0 & \text{if} \quad \alpha = 2 \end{cases}$$

Discrete Amplifier

Last example shows how to implement a discrete amplifier which, as soon as the spike amount passing through control synapse (B, B') goes beyond a given threshold k, computes the amplification function $f_{*,n}(x) = n*x$ from the input given at E, otherwise no amplification is performed. Rules $a^l \rightarrow a^l$ belonging to neuron p are interpreted in the same way as in [13]. Implementation involves defining two functions: $g(x) = f_{*,n}(x)$ and $f(x)$, which associates x for every $x \in \mathbb{N}$.

Discrete amplifier astrocyte, ast_{amp}, is depicted in the Fig. 9.4 with its formal specification being:

$$ast_{amp} = (\{(p,p'),(q',q)\},\{(B,B')\}, false, \{0,k\},\{f(x),g(x)\},0,false)$$

and its working equation, assuming that at a given instant t α spikes pass through synapse (B,B') and β spikes pass through synapse (p,p'), being:

$$ast_{amp}(\alpha,\beta,t) = \begin{cases} f(\beta)=\beta & \text{if} \quad 0 \leq \alpha < k \\ g(\beta)=n*\beta & \text{if} \quad \alpha \geq k \end{cases}$$

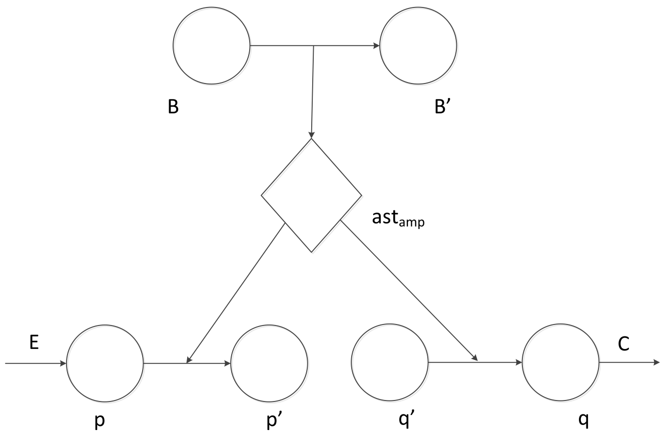


Figure 9.4: Discrete amplifier astrocyte.

9.4 P–Lingua syntax for SN P systems

Next, we detail the P–Lingua syntax for specifying SN P systems in P–Lingua. The starting point for this syntax is the one introduced in [131] and [45, 100, 132]. From there, P–Lingua language syntax is extended to incorporate new features related to SN P systems. Such features include, among others, different simulation modes, an initial configuration specification, consisting of set of neurons and a collection of synapses, and the possibility of defining a simulation input spike train and input and output neurons.

9.4.1 Reserved words

The set of reserved words has been updated by adding the following text strings:

```
|, @masynch, @mseq, @mvalid, @marcs, @mdict, @min, @minst,
@mout, @moutres_binary, @moutres_natural, @moutres_summatories,
@mastf, @masth, @mastfunc, zero, identity, pol,
@mboundall, @mlocset
```

A detailed explanation about this set of reserved words is provided in what follows.

9.4.2 Model specification

Any P–Lingua file defining a SN P system must begin with the following sentence:

```
@model<spiking_psystems>
```

9.4.3 Valid computation specification

The next sentence can be used to define a valid computation:

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

where, for each integer $i \in [1, \ldots, N]$:

- m_i is a neuron label in the SN P system.
- n_i is an integer expression which specifies the number of spikes contained in m_i at the end of the computation.

If the sentence is not used then every halting computation is considered valid.

9.4.4 Asynchronous modes

P–Lingua supports several asynchronous modes for SN P systems. To specify the asynchronous mode, the following sentence must be written, preferably after the model specification.

```
@masynch = v1;
```

where $v1 \in \{0, 1, 2, 3\}$. If this sentence is not present, then *v1* defaults to 0. These values denote the following modes (see Chapter 4 and [63, 124, 165] for details):

0: *Synchronous (standard)* mode.
1: *Asynchronous* mode.
2: *Limited asynchronous* mode.
3: *Asynchronous with local synchronization* mode.

If `@masynch` is set to 2, then the following sentence must be used to set the global upper bound:

```
@mboundall = b;
```

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

If `@masynch` is set to 3, then the following sentence must be used to set the local synchronizing sets:

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

where for an arbitrary element ls_h we have that $ls_h = \{\sigma_{h,1}, \ldots, \sigma_{h,u_h}\}$, which is a non-empty set of neuron labels.

9.4.5 Sequential modes

P–Lingua supports several sequential modes for SN P systems. To specify the sequential mode, the following sentence must be written, preferably after the model specification.

```
@mseq = v2;
```

where $v2 \in \{0, \ldots, 5\}$. If this sentence is not present, then *v2* defaults to 0. These values denote the following modes (see Chapter 4 and [63] for details):

0: *parallel (standard)* mode.
1: *pure-seq* mode.
2: *max-pseudo-seq* mode.
3: *max-seq* mode.
4: *min-pseudo-seq* mode.
5: *min-seq* mode.

9.4.6 Initial membrane structure

A SN P system specification in P–Lingua must define an initial membrane structure, which is composed of a set of neurons connected by synapses. These synapses are specified as a set of arcs. If a SN P system with division and budding rules is being specified, then defining a synapse dictionary is mandatory. Besides, an input neuron, a set of output neurons and an input spike train can also be specified.

Let us consider an initial membrane structure of a SN P system with N neurons and M synapses. In what follows, defining that initial membrane structure is explained:

Initial neurons

Initial neurons must be specified with the following sentence:

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

where, for each integer $i \in [1, \ldots, N]$, m_i is the label of neuron i. The label *environment* cannot be used.

Initial synapses

Initial synapses must be specified with the following sentence:

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

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

Synapse dictionary

The synapse dictionary can be specified with the following sentence:

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

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

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

Input neuron

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

```
@min = in;
```

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

Input spike train

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

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

where, for each integer $i \in [1, \ldots, S]$, $st_i = (i, a_i)$, i and a_i are two integer expressions, with $i \geq 1$ and $a_i \geq 0$. The pair $st_i = (i, a_i)$ is interpreted in the following way: at step i a number of a_i spikes are introduced in the input neuron. When the pair $st_j = (j, a_j)$ is undefined for some step j the simulator assumes that zero spikes are introduced in the system at that step. Only spikes, but not anti-spikes, can be introduced as part of the input spike train.

Output neurons

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

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

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

9.4.7 An initial membrane structure definition example

The following piece of code shows the first lines of a P-Lingua SN P system definition file. These lines show examples of the previously introduced syntax.

```
@model<spiking_psystems>

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

@mseq = 2;

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

9.4.8 Alphabet symbols

In SN P systems, as originally defined, the alphabet only contains one symbol, which is called *spike* and denoted by a. In SN P systems with anti-spikes, the symbol $\overline{a}$ is used to denote the anti-spike. The corresponding alphabet symbols are written in P–Lingua as `a` and `_a` respectively.

9.4.9 Initial multisets

Initial multisets of objects for neurons can be defined in the same way as initial multisets of objects for cell-like systems, with the restriction that only objects `a` and `_a` can be used. Examples:

```
@ms(0) = a*15;
@ms(3) = _a*9;
```

9.4.10 Regular expressions

In SN P systems, regular expressions are associated to the activation of rules. Consequently, it is necessary to provide a mechanism to define and evaluate regular expressions in P–Lingua. Since P–Lingua is written in Java, the standard Java support for regular expression provided by the Java package *java.util.regex* is used. Details about this package can be found at [187].

In P–Lingua, the following subset of symbols can be used to construct regular expressions, according to the syntax specified in [187]:

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

Regular expressions are not checked by P–Lingua parser. Instead, they are piped directly into the simulator, which performs the parsing. A regular expression E is written double-quoted in P–Lingua in this way: "E". Some examples are provided below.

9.4.11 Rules

SN P systems without anti-spikes

Four types of rules can be defined:

(1) *Firing rules*, that can be specified in the following ways:

- `[a*c]'h --> [a*p]'h "e" :: d;`
- `[a*c    -->  a*p]'h "e" :: d;`

(2) *Forgetting rules*, that can be specified in the following ways:

- `[a*c]'h --> [#]'h "e" :: d;`
- `[a*c    -->  #]'h "e" :: d;`

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

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

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

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

where h, i, j and k are neuron labels, c, p and d are integer expressions which satisfy $c{\geq}1$, $c{\geq}p$ and $d{\geq}0$, and e is a regular expression over $\{a\}$.

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

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

For instance, the following rules are valid firing/forgetting rules in P-Lingua:

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

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

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

Let us recall that in P-Lingua, the symbol # is optional (it can be omitted). For instance, the following rules:

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

can be written equivalently as:

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

SN P systems with anti-spikes

Two kinds of rules involving anti-spikes can be defined:

(1) *Firing rules*, that can be specified in the following ways:

- `[b*c]'h --> [b'*p]'h "e" :: d;`
- `[b*c    -->  b'*p]'h "e" :: d;`

(2) *Forgetting rules*, that can be specified in the following ways:

- `[b*c]'h --> [#]'h "e" :: d;`
- `[b*c    -->  #]'h "e" :: d;`

where h is a neuron label, c, p and d are integer expressions which satisfy $c \geq 1$, $c \geq p$ and $d \geq 0$, $b \in \{a, \overline{a}\}$, and e is a regular expression either over a or over $\overline{a}$ (but not over a and $\overline{a}$ simultaneously).

For instance, the following rules are valid firing/forgetting rules in P-Lingua:

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

9.4.12 Output results

Let Π be a SN P system with output neurons $o = \{o_1, \ldots, o_L\}$. Without loss of generality, we can assume a total ordering $<$ among elements of o given by the lexicographical ordering of the neuron labels. Let $\gamma = C_0 \Rightarrow C_1 \Rightarrow \ldots$ be a computation of Π (C_0 is the initial configuration, and $C_{i-1} \Rightarrow C_i$ is the i-th step of γ). Then, for each step t ($t \geq 1$):

- We denote $bs_i(t)$ as the binary spike train generated by output neuron o_i until step t, which is a binary sequence where the steps from γ when neuron o_i fires are marked with 1 and the steps when neuron o_i does not fire are marked with 0. This sequence can be formally defined as follows:

$bs_i(t) = \langle bs_i^{(1)}, \dots, bs_i^{(t)} \rangle, 1 \leq i \leq L$, such that

$$bs_i^{(j)} = \begin{cases} 1 & \text{if } o_i \text{ fired at step } j \\ 0 & \text{otherwise} \end{cases}, 1 \leq j \leq t.$$

- We denote $bs(t) = \langle bs_1(t), \dots, bs_L(t) \rangle$ as the binary spike train generated by system Π until step t.

  Let us notice that there is an implicit total order $<$ on elements $bs_i(t) \in bs(t)$ given by the total order among elements of o.

- We denote $ns_i(t)$ as the natural spike train generated by output neuron o_i until step t, which is an ordered sequence of steps from γ where only those steps when neuron o_i fires appear. This sequence can be formally defined in the following way:

  - $K_i(t) = \{j \mid bs_i^{(j)} = 1 \wedge 1 \leq i \leq L, 1 \leq j \leq t\}$, with $|K_i(t)| = k_{i,t}$.
  - We impose a total order $<$ among elements of $K_i(t)$ following the natural order of its elements, and write the sequence as follows: $ns_i(t) = \langle ns_{i,1}, \dots, ns_{i,k_{i,t}} \rangle, 1 \leq i \leq L, 1 \leq k_{i,t} \leq t$.

- We denote $ns(t) = \langle ns_1(t), \dots, ns_L(t) \rangle$ as the natural spike train generated by system Π until step t.

  Let us notice that there is an implicit total order $<$ on elements $ns_i(t) \in ns(t)$ given by the total order among elements of o.

The previous definitions generalize the concept of output spike trains for SN P systems with multiple output neurons for each computation step t, following from the discussion on SN P systems output (see Section 4.2, and [63, 128] for more details). From here, it is also possible to consider generalized versions of other outputs such as (alternate) differences between spiking instants for each output neuron, requiring for a given output neuron to fire at least k times (weak case) or exactly k times (strong case), etc.

According to all of this, the simulator provides the following output information for each step t:

- $bs(t)$, binary spike train generated by Π until step t.
- $ns(t)$, natural spike train generated by Π until step t.

- C_t, configuration generated by Π after step t, which also includes the number of spikes placed in the environment.

For halting computations, additional output information can be provided by the simulator writing the following optional sentences in the P–Lingua specification file:

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

Let us suppose that computation halts in t' steps. Then we have the following:

- When sentence `@moutres_binary` is specified, the complete binary spike train generated by Π is shown, that is: $bs(t') = \langle bs_1(t'), \dots, bs_L(t') \rangle$.

- When sentence `@moutres_natural(k,strong,alternate)` is specified, a sequence of time instants between spikes generated by output neurons of Π is shown, with parameters k, *strong* and *alternate* determining how to generate the output. Parameter k is an integer expression with $k \geq 2$, which indicates the number of spiking times for each output neuron. Parameter *strong* takes a boolean value (*true* or *false*), which indicates if the output is computed in a *strong* case (the output neuron fires exactly k times) or not. Parameter *alternate* also takes boolean values and determines if all the times differences (*false*) should be considered, or only alternate ones (*true*).

- When sentence `@moutres_summatories` is specified, the output of Π is calculated as a set consisting of, for each output neuron, the total number of spikes/anti-spikes sent by the neuron to the environment.

9.4.13 Astrocytes

In what follows, we present P–Lingua syntax for defining astrocytes. Two kinds of astrocytes are considered: functional astrocytes and hybrid astrocytes.

Syntax for functional astrocytes

Neuron division rules cannot be used together with functional astrocytes. This comes from a model design issue that arises when splitting into two new synapses an existing operand synapse as a result of a neuron division. As functions have a fixed number of operands, it would be necessary to define which synapse would be associated to the function operand. P–Lingua syntax for defining SN P systems with functional astrocytes follows:

- *Astrocytes.*

  The following sentence can be used to define a functional astrocyte ast_j^f:

  ```
  @mastf =
  (
  label-j,
  operand-synapses-j,control-synapses-j,control-operand-flag-j,
  set-thresholds-j,set-functions-j,
  potential-j,update-potential-j
  );
  ```

  where:

  - *label-j* is the label of the astrocyte;
  - *operand-synapses-j* is the set of operand synapses associated to the astrocyte, being that set $\{s^o_{j,1}, \dots, s^o_{j,r_j}\}$, where $s^o_{j,v} = (\sigma^{o,1}_{j,v}, \sigma^{o,2}_{j,v})$, a pair of neuron labels defining the synapse;
  - *control-synapses-j* is the set of control synapses associated to the astrocyte, being that set $\{s^c_{j,1}, \dots, s^c_{j,q_j}\}$, with $s^c_{j,u} = (\sigma^{c,1}_{j,u}, \sigma^{c,2}_{j,u})$, a pair of neuron labels defining the synapse;
  - *control-operand-flag-j* is the astrocyte control-as-operand flag, with *control-operand-flag-j* $\in \{true, false\}$;
  - *set-thresholds-j* is the astrocyte natural set of thresholds, being that set $\{T_{j,1}, \dots, T_{j,k_j}\}$, with $T_{j,1} < \cdots < T_{j,k_j}$;
  - *set-functions-j* is the astrocyte set of natural computable functions, being that set $\{f_{j,1}, \dots, f_{j,k_j}\}$, with all functions having the same arity;
  - *potential-j* is the astrocyte initial potential, with *potential-j* $\in \mathbb{N}$;

  - *update-potential-j* is the astrocyte potential update flag, verifying that *update-potential-j* $\in \{true, false\}$;

- *Functions.*

  The following sentence can be used to define a function of name *f-name*:

```
@mastfunc =
(
f-name(x1,...,xN),
f-name(x1,...,xN) = "expr(x1,...,xN)"
);
```

  where:

  - *f-name* is the function name, a P–Lingua identifier;
  - $x1, \ldots, xN$ is the list of arguments; notation for naming arguments must follow the convention of starting with x and immediately being followed by a integer literal, starting with 1 and being incremented by one each time;
  - $exp(x1, ..., xN)$ is an expression defining the function, which must yield a natural value;

  For defining and dynamically calculate functions value, an external library called *exp4j* [30] is used. Version of the library used is 0.2.9, which syntax offers a variety of operators and built-in functions. This syntax can be consulted at `http://projects.congrace.de/exp4j/`. Let us notice that, as we are restricted when defining functions, SN P systems with functional astrocytes are only partially simulated. The following functions are pre-defined and can be used directly, without having to be explicitly defined in the P–Lingua source file:

  - $zero(x1)$ is the identically zero function of arity one;
  - $identity(x1)$ is the identity function of arity one;
  - $pol()$ is a *function template* allowing the definition of a polynomial astrocyte function $pol(x_0, x_1, \ldots, x_n, x)$ of any arity $n + 2$, defined as follows:

$$pol(x_0, x_1, \ldots, x_n, x) = x_0 + \sum_{i=1}^{n} x_i * x^i$$

$x_0, \ldots, x_n, x$ arguments over $\mathbb{N}$ that take value from the number of spikes/anti-spikes passing through the operand synapses associated to a given astrocyte ast_j at a instant t in the following way:

$$\begin{cases} x_0 \leftarrow s^o_{j,1}(t) \\ x_1 \leftarrow s^o_{j,2}(t) \\ \ldots \\ x_n \leftarrow s^o_{j,r_j-2}(t) \\ x \leftarrow s^o_{j,r_j-1}(t) \end{cases}$$

– *sub*() is a *function template* allowing the definition of a natural substraction function $sub(x_1, \ldots, x_n)$ of any arity $n \geq 1$, defined as follows:

$$sub(x_1, \ldots, x_n) = \begin{cases} x_1 - x_2 - \cdots - x_n & \text{if } x_1 - x_2 - \cdots - x_n \geq 0; \\ 0 & \text{otherwise;} \end{cases}$$

with $x_i \in \mathbb{N}, 1 \leq i \leq n$;

$x_1, \ldots, x_n$ arguments take value from the spikes passing through the operand synapses of a given astrocyte ast_j at an instant t in the following way:

$$\begin{cases} x_1 \leftarrow s^o_{j,1}(t) \\ \ldots \\ x_n \leftarrow s^o_{j,r_j-1}(t) \end{cases}$$

Syntax for hybrid astrocytes

The following P–Lingua sentence can be used to define hybrid astrocyte ast^h_j:

```
@masth = (label-j,synapses-j,potential-j);
```

where:

- *label-j* is the label of the astrocyte;

- *synapses-j* is the set of neighbouring synapses associated to the astrocyte, being that set $\{s_{j,1}, \ldots, s_{j,r_j}\}$, with $s_{j,y} = (\sigma^1_{j,y}, \sigma^2_{j,y})$, a pair of neuron labels defining the synapse;
- *potential-j* is the astrocyte potential with *potential-j* $\in \mathbb{N}$;

Using functional and hybrid astrocytes together

P–Lingua allows using both kind of astrocytes, functional and hybrid, together, but the following restrictions apples: it is not allowed to connect a synapse associated to an excitatory/inhibitatory astrocyte to an operand synapse associated to a functional astrocyte.

9.5 A unified algorithm for simulating SN P systems in P–Lingua

Let us recall that when a family of recognizer P systems is used to solve a decision problem, each member of the family processing an instance is confluent, in the sense that all computations for a given input must generate the same output. Consequently, a simulation algorithm for recognizer P systems only has to reproduce one possible computation of the simulated P system.

The unified simulation algorithm described below generates one possible computation for a recognizer SN P system, incorporating all the features of the previously discussed SN P systems variants, with an initial configuration C_0 containing n neurons $m_1, \ldots, m_n$ plus a virtual neuron m_0 representing the environment.

The simulation algorithm is structured in the following stages:

I. Initialization

Data structures needed to perform the simulation are initialized.

II. Selection of rules

The set of rules to be executed in the current step is calculated.

III. Sequential filtering

A filtering on the neurons to execute rules is applied according to the sequential mode in which the simulator operates.

IV. Asynchronous filtering

A filtering on the neurons to execute rules is applied according to the asynchronous mode in which the simulator operates.

V. Remove left hand rule objects

For those rules that start their execution in the current steps, the associate neurons remove the left hand rule objects.

VI. Classify selected rules

The rules to be executed are split into different sets, according to their kind.

VII. 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.

VIII. Execute spiking rules

Execution of spiking rules is performed.

IX. Ending

The current configuration is updated with the newly calculated one and the halting condition is checked (that is, if no more rules are applicable).

The simulation algorithm follows.

```
I. Initialization
    1. Let $C_0$ be the initial configuration with $n$ neurons
       denoted by $m_1, \ldots, m_n$ plus a virtual neuron $m_0$
       representing the environment
    2. Let $C_t \leftarrow C_0$ be the current configuration
    3. Let $asynch$ be the asynchronous mode in which the simulation
       is performed; the following values are possible:
         0: synchronous (standard) mode.
         1: asynchronous mode.
         2: limited asynchronous mode.
         3: asynchronous with local synchronization mode.
    4. Let $seq$ be the sequential mode in which the simulation
       is performed; the following values are possible:
```

```
    0: parallel (standard) mode.
    1: pure-seq mode.
    2: max-pseudo-seq mode.
    3: max-seq mode.
    4: min-pseudo-seq mode.
    5: min-seq mode.
5. For each neuron $m_i \in C_t$ the following information is stored:
    • Let $id(m_i)$ be the neuron identifier of $m_i$
      (each time a neuron is created, an identifier is assigned,
      starting from $0$, which is associated to the environment)
    • Let $l(m_i)$ be the neuron label of $m_i$
    • Let $r(m_i) \leftarrow \langle\rangle$ be the last selected rule to be executed by $m_i$
      ($\langle\rangle$, the void rule, denotes no selected rule)
    • Let $o(m_i) \leftarrow a$ the kind of object (spike/anti-spike)
      contained in $m_i$
    • Let $n(m_i)$ the number of spikes/anti-spikes contained in $m_i$
    • Let $d(m_i) \leftarrow 0$ be an integer that controls if $m_i$ is open or closed;
      it functions as follows:
        – If $d(m_i) = 0$ then $m_i$ is open, otherwise it is closed
        – If $r(m_i)$ is a firing/forgetting rule then $d(m_i)$ stores
          the number of steps left for $m_i$ to become open
        – If $r(m_i)$ is a budding/division rule then $d(m_i)$ is equal to $-1$
    • Let $b(m_i) \leftarrow 0$ be an integer with a double function:
        – Controlling if $m_i$ has changed its state from
          the last computation step;
          if $b(m_i) = -1$ then $m_i$ has not changed its state and
          does not need to check for new applicable rules
          ($m_i$ state is changed when its multiset is changed or
          it finalizes a rule execution)
        – Storing the number of steps left for $m_i$ to postpone
          the execution of $r(m_i)$ (in this case $b(m_i) \geq 0$)
6. For each arc $e \in C_t$ from neuron $m$ to $m'$ with $e \equiv (m, m')$
   the following information is stored:
    • Let $n_i(e)$ be the number of spikes/anti-spikes
      contained in the input buffer of $e$
    • Let $n_o(e)$ be the number of spikes/anti-spikes
      contained in the output buffer of $e$
    • Let $o(e) \leftarrow a$ the kind of object (spike/anti-spike)
      contained in $e$
    • Let $inh(e) \leftarrow false$ a boolean value indicating
      if traffic in $e$ is inhibited
    • Let $ast(e)$ be a boolean value indicating
      if traffic in $e$ is controlled by some astrocyte
```

7. For each functional astrocyte $fast_j \in C_t$ the following information is stored:
   - Let $l(fast_j)$ be the label of $fast_j$
   - Let $op(fast_j) = \{s^o_{j,1}, \dots, s^o_{j,r_j}\}, r_j \geq 1$, be the *operand synapses* set
   - Let $ct(fast_j) = \{s^c_{j,1}, \dots, s^c_{j,q_j}\}, q_j \geq 0$ be the *control synapses* set
   - Let $\omega(fast_j) \in \{true, false\}$ be the control-as-operand flag
   - Let $T(fast_j) = \{T_{j,1}, \dots, T_{j,k_j}\}, k_j \geq 1$ be thresholds set
   - Let $F(fast_j) = \{f_{j,1}, \dots, f_{j,k_j}\}, k_j \geq 1$ be the natural functions finite *multiset*
   - Let $p(fast_j) \in \mathbb{N}$ be the potential
   - Let $\gamma(fast_j) \in \{true, false\}$ be the potential update flag
   - For each set $S \in \{op(fast_j), ct(fast_j), T(fast_j), F(fast_j)\}$ with $S \equiv \{s_1, \dots, s_z\}$ S_x yields s_x $1 \leq x \leq z$
8. For each hybrid astrocyte $hast_j \in C_t$ the following information is stored:
   - Let $l(hast_j)$ be the label of $hast_j$
   - Let $syn(fast_j)$ be the *synapses* set influenced by $hast_j$
   - Let $th(fast_j) \in \mathbb{N}$ be the threshold
9. Let b be the global bound that marks the number of steps for neurons to postpone the execution of its selected rule; (if $asynch \neq 3$ then $b = 0$, which means no bound)
10. Let $Loc \leftarrow \{loc_1, loc_2, \dots, loc_l\} \subseteq \mathcal{P}(\{m_1, m_2, \dots, m_n\})$ be the family of sets of locally synchronous neurons with $\mathcal{P}(\{m_1, m_2, \dots, m_n\})$ the power set of $\{m_1, m_2, \dots, m_n\}$
11. Let $R_{sel} \leftarrow \{\}$ be the set that stores the selected rules to be executed at the current execution step; each element of R_{sel} has the form $\langle m_i, r(m_i)\rangle$
12. Let $toFireNotFiltered \leftarrow \{\}$ be a set that stores the neurons having a selected rule for the current computation step; each of its elements has the form $\langle id(m_i), m_i\rangle$
13. Let $toFireFiltered \leftarrow \{\}$ be a set that stores the neurons having a selected rule for the current computation step and that will be executed after applying a filtering according to the sequential/asynchronous mode in which the simulator is operating; each of its elements has the form $\langle id(m_i), m_i\rangle$
14. Let $justOpenNow \leftarrow \{\}$ be a set that stores the neurons becoming open at the current computation step after being closed executing a firing/forgetting rule; each of its elements has the form $\langle id(m_i), m_i\rangle$
15. Let $Division \leftarrow \{\}$ be the set of neurons having a division rule selected to start its execution at the current computation step

```
 16. Let $Budding \leftarrow \{\}$ be the set of neurons having a budding rule
     selected to start its execution at the current computation step
 17. Let $Spiking \leftarrow \{\}$ be the set of neurons having a spiking rule
     selected to start its execution at the current computation step
II. Selection of rules
  1. For each neuron $m_i \in C_t$ do
     (a) If $b(m_i) = 0 \wedge d(m_i) > 0$ Then
           • $d(m_i) \leftarrow d(m_i) - 1$
           • If $d(m_i) = 0$ Then Add $id(m_i)$ to $justOpenNow$
           • Add $\langle m_i, r(m_i)\rangle$ to $R_{sel}$
           • Goto II.1 /* process next neuron */
     (b) Clear neuron $m_i$ state:
           • $r(m_i) \leftarrow \langle\rangle$
           • $d(m_i) \leftarrow 0$
     (c) If $b(m_i) = -1$ Then Goto II.1 /* process next neuron */
     (d) Let $S \leftarrow \{\}$
     (e) For each spiking rule $r' \equiv E/s^c \rightarrow s'^p; d \in \mathcal{R}_{l(m_i)}$ do
           • If all the following statements are true:
               – $o(m_i)^{n(m_i)} \in L(E)$
               – $o(m_i) = s$
               – $n(m_i) \geq c$
             Then Add $r'$ to $S$
     (f) For each division rule $r' \equiv [E]_{l(m_i)} \rightarrow [\ ]_j || [\ ]_k \in \mathcal{R}_{l(m_i)}$ do
           • If all the following statements are true:
               – $o(m_i)^{n(m_i)} \in L(E)$
               – $\nexists m' \in C_t : l(m') \in \{j, k\} \wedge$
                   $((l(m'), l(m_i)) \in syn \vee (l(m_i), l(m')) \in syn)$
             Then Add $r'$ to $S$
     (g) For each budding rule $r' \equiv [E]_{l(m_i)} \rightarrow [\ ]_{l(m_i)}/[\ ]_j \in \mathcal{R}_{l(m_i)}$ do
           • If all the following statements are true:
               – $o(m_i)^{n(m_i)} \in L(E)$
               – $\nexists m' \in C_t : l(m') = j \wedge (l(m_i), l(m')) \in syn$
             Then Add $r'$ to $S$
     (h) If $S = \{\}$ Then
           • $b(m_i) \leftarrow -1$
           • Goto II.1 /* process next neuron */
         Else
           • Let $r'$ be a non-deterministically selected rule from $S$
           • $r(m_i) \leftarrow r'$
           • If $r(m_i) \equiv E/s^c \rightarrow s'^p; d$ Then $d(m_i) \leftarrow d$ Else $d(m_i) \leftarrow -1$
           • If $b \neq 0$ Then
```

```
            – If $b(m_i) = 0$ Then $b(m_i) = b - 1$ Else $b(m_i) = b(m_i) - 1$
        • Add $\langle id(m_i), m_i \rangle$ to $toFireNotFiltered$
        • Add $\langle id(m_i), m_i \rangle$ to $toFireFiltered$
        • Add $\langle m_i, r(m_i) \rangle$ to $R_{sel}$
III. Sequential fitering
  1. If $toFireFiltered = \{\}$ Then Return
  2. If $seq = 0$ Then Return
  3. Let $filtered \leftarrow \{\}$
  4. Let $min \leftarrow min\{n(m_i) : \langle id(m_i), m_i \rangle \in toFireFiltered\}$
  5. Let $max \leftarrow max\{n(m_i) : \langle id(m_i), m_i \rangle \in toFireFiltered\}$
  6. Call $decideSequential(min, max, filtered)$
  7. $toFireFiltered \leftarrow filtered$
Procedure $decideSequential(min, max, filtered)$
  1. Let $min, max, filtered$ be input/output arguments declared
     consistently as specified above
  2. If $seq = 1$ Then
     (a) Let $\langle id(m_i), m_i \rangle$ be a non-deterministically
         selected tuple from $toFireFiltered$
     (b) Add $\langle id(m_i), m_i \rangle$ to filtered
  3. If $seq = 2$ Then
     (a) For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
           • If $n(m_i) = max$ Then Add $\langle id(m_i), m_i \rangle$ to $filtered$
  4. If $seq = 3$ Then
     (a) Let $filteredAux \leftarrow \{\}$
     (b) For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
           • If $n(m_i) = max$ Then Add $\langle id(m_i), m_i \rangle$ to $filteredAux$
     (c) Let $\langle id(m_i), m_i \rangle$ be a non-deterministically
         selected tuple from $filteredAux$
     (d) Add $\langle id(m_i), m_i \rangle$ to $filtered$
  5. If $seq = 4$ Then
     (a) For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
           • If $n(m_i) = min$ Then Add $\langle id(m_i), m_i \rangle$ to $filtered$
  6. If $seq = 5$ Then
     (a) Let $filteredAux \leftarrow \{\}$
     (b) For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
           • If $n(m_i) = min$ Then Add $\langle id(m_i), m_i \rangle$ to $filteredAux$
        (c) Let $\langle id(m_i), m_i\rangle$ be a *non-deterministically*
            selected tuple from $filteredAux$
        (d) Add $\langle id(m_i), m_i\rangle$ to $filtered$
IV. Asynchronous fitering
   1. If $toFireFiltered = \{\}$ Then Return
   2. If $asynch = 0$ Then Return
   3. Let $filtered \leftarrow \{\}$
   4. For each tuple $\langle id(m_i), m_i\rangle \in toFireFiltered$ do
        (a) If $DecideAsynch(m_i) = true$ Then Add $\langle id(m_i), m_i\rangle$ to $filtered$
   5. If $asynch = 3$ Then
        (a) Let $filteredAux \leftarrow \{\}$
        (b) Let $locProcessed \leftarrow \{\}$
        (c) For each tuple $\langle id(m_i), m_i\rangle \in filtered$ do
              • Call $addLocMembranes(id(m_i), locProcessed, filteredAux)$
        (d) $filtered = filteredAux$
   6. $toFireFiltered = filtered$
Procedure $decideAsynch(m_i)$
   1. Let $m_i$ be an input/output argument declared consistently as
      specified above
   2. Let $result \leftarrow false$
   3. Let $p$ be a boolean value *non-deterministically*
      selected from $\{true, false\}$
   4. If *any* of the following statements is *true*:
        • $asynch \in \{1, 3\} \wedge p = true$
        • $asynch = 2 \wedge b(m_i) = 0$
        • $asynch = 2 \wedge b(m_i) > 0 \wedge p = true$
      Then
        • $result \leftarrow true$
        • $b(m_i) \leftarrow 0$
   5. Return $result$
Procedure $addLocMembranes(id(m_i), locProcessed, filteredAux)$
   1. Let $id(m_i), locProcessed, filteredAux$ be input/output arguments
      declared consistently as specified above
   2. If $id(m_i) \in locProcessed$ Then Return
      Else
        (a) Add $id(m_i)$ to $locProcessed$
        (b) If $\exists \langle id(m_i), m_i\rangle \in toFireNotFiltered$ Then
```

```
            • $b(m_i) \leftarrow 0$
            • Add $\langle id_i, m_i \rangle$ to $filteredAux$
            • Let $S \leftarrow \{id(m_k) : m_k \in \bigcup_{1 \le j \le l} \{loc_j : loc_j \in Loc \wedge m_i \in loc_j\} \setminus \{m_i\}\}$
            • For each identifier $id(m_k) \in S$ do
                – Call $addLocMembranes(id(m_k), locProcessed, filteredAux)$

V. Remove left hand rule objects
  1. For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
      (a) If $r(m_i) \equiv E/s^c \rightarrow s'^p; d$ (spiking rule) Then
            • Remove $c$ occurrences of $s$ from $m_i$
      (b) If $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_j || [\ ]_k$ (division rule) Then
            • Remove all of the objects from $m_i$
      (c) If $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_{l(m_i)} / [\ ]_j$ (budding rule) Then
            • Remove all of the objects from $m_i$

VI. Classify selected rules
  1. For each tuple $\langle id(m_i), m_i \rangle \in toFireFiltered$ do
      (a) If $r(m_i) \equiv E/s^c \rightarrow s'^p; d$ (spiking rule) Then
            • Add $m_i$ to $Spiking$
      (b) If $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_j || [\ ]_k$ (division rule) Then
            • Add $m_i$ to $Division$
      (c) If $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_{l(m_i)} / [\ ]_j$ (budding rule) Then
            • Add $m_i$ to $Budding$
  2. For each tuple $\langle id(m_i), m_i \rangle \in justOpenNow$ do
      (a) Add $m_i$ to $Spiking$

VII. Execute division and budding rules
  1. Let $S \leftarrow \{\}$
  2. For each neuron $m_i \in Division$ with $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_j || [\ ]_k$ do
      (a) $l(m_i) \leftarrow j$
      (b) Let $m'$ be a newly created neuron such that:
            • $id(m') \leftarrow \{$a new identifier$\}$
            • $l(m') \leftarrow k$
            • $r(m') \leftarrow r(m_i)$
            • $o(m') \leftarrow a$
            • $d(m') \leftarrow -1$
            • $b(m') \leftarrow 0$
      (c) Add $m'$ to $C_t$
      (d) For each arc $e \in C_t$ with $e \equiv (m'', m_i)$ do
            • Let $e' \equiv (m'', m')$ be a newly created arc such that:
                – $n_i(e') \leftarrow 0$
                – $n_o(e') \leftarrow 0$
```

```
          – $o(e') \leftarrow a$
          – $inh(e') \leftarrow false$
          – $ast(e') \leftarrow false$
        • Add $e'$ to $C_t$
    (e) For each arc $e \in C_t$ with $e \equiv (m_i, m'')$ do
        • Let $e' \equiv (m', m'')$ be a newly created arc such that:
          – $n_i(e') \leftarrow 0$
          – $n_o(e') \leftarrow 0$
          – $o(e') \leftarrow a$
          – $inh(e') \leftarrow false$
          – $ast(e') \leftarrow false$
        • Add $e'$ to $C_t$
    (f) Add $m_i$ to $S$
    (g) Add $m'$ to $S$
    (h) Let $sLoc \leftarrow \{loc_j : loc_j \in Loc \wedge m_i \in loc_j\}$
    (i) For each $loc_j \in sLoc$ do
        • Add $m'$ to $loc_j$
3. For each neuron $m_i \in Budding$ with $r(m_i) \equiv [E]_{l(m_i)} \rightarrow [\ ]_{l(m_i)}/[\ ]_j$ do
    (a) Let $m'$ be a newly created neuron such that:
        • $id(m') \leftarrow \{$a new identifier$\}$
        • $l(m') \leftarrow j$
        • $r(m') \leftarrow r(m_i)$
        • $o(m') \leftarrow a$
        • $d(m') \leftarrow -1$
        • $b(m') \leftarrow 0$
    (b) Add $m'$ to $C_t$
    (c) For each arc $e \in C_t$ with $e \equiv (m_i, m'')$ do
        • Remove $e$ from $C_t$
        • Let $e' \equiv (m', m'')$ be a newly created arc such that:
          – $n_i(e') \leftarrow 0$
          – $n_o(e') \leftarrow 0$
          – $o(e') \leftarrow a$
          – $inh(e') \leftarrow false$
          – $ast(e') \leftarrow false$
        • Add $e'$ to $C_t$
    (d) Let $e'' \equiv (m, m')$ be a newly created arc such that:
        • $n_i(e'') \leftarrow 0$
        • $n_o(e'') \leftarrow 0$
        • $o(e'') \leftarrow a$
        • $inh(e'') \leftarrow false$
        • $ast(e'') \leftarrow false$
    (e) Add $e''$ to $C_t$
    (f) Add $m'$ to $S$
```

```
      (g) Let $sLoc \leftarrow \{loc_j : loc_j \in Loc \wedge m_i \in loc_j\}$
      (h) For each $loc_j \in sLoc$ do
            • Add $m'$ to $loc_j$
   4. For each neuron $m \in S$ do
      (a) Let $Ingoing \leftarrow \{m' : m' \in C_t \wedge (l(m'), l(m)) \in syn\}$
      (b) For each neuron $m' \in Ingoing$ do
            • Let $e \equiv (m', m)$ be a newly created arc such that:
                – $n_i(e) \leftarrow 0$
                – $n_o(e) \leftarrow 0$
                – $o(e) \leftarrow a$
                – $inh(e) \leftarrow false$
                – $ast(e) \leftarrow false$
            • Add $e$ to $C_t$
      (c) Let $Outgoing \leftarrow \{m' : m' \in C_t \wedge (l(m), l(m')) \in syn\}$
      (d) For each neuron $m' \in Ingoing$ do
            • Let $e \equiv (m, m')$ be a newly created arc such that:
                – $n_i(e) \leftarrow 0$
                – $n_o(e) \leftarrow 0$
                – $o(e) \leftarrow a$
                – $inh(e) \leftarrow false$
                – $ast(e) \leftarrow false$
            • Add $e$ to $C_t$
VIII. Execute spiking rules
   1. Let $S \leftarrow \{\}$
   2. For each neuron $m_i \in Spiking$ with $r(m_i) \equiv E/s^c \rightarrow s'^p; d \in \mathcal{R}_{l(m_i)}$ do
      (a) For each arc $e \in C_t$ with $e \equiv (m_i, m')$ do
            • $o(e) \leftarrow s'$
            • $n_i(e) \leftarrow p$
            • Add $e$ to $S$
   3. Call $applyFunctionalAstrocytes(S)$
   4. Call $applyHybridAstrocytes(S)$
   5. For each arc $e \in S$ with $e \equiv (m_1, m_2)$ do
      (a) Let $object \leftarrow o(e)$
      (b) Let $spikes \leftarrow \begin{cases} 0 & \text{if} \quad inh(e) = true \vee d(m_2) \neq 0 \\ n_i(e) & \text{if} \quad ast(e) = false \\ n_o(e) & \text{if} \quad \text{none of the above} \end{cases}$
      (c) $n_i(e) \leftarrow 0$
      (d) $n_o(e) \leftarrow 0$
      (e) $o(e) \leftarrow a$
      (f) $inh(e) \leftarrow false$
      (g) Add $spikes$ occurrences of $object$ to $m_2$
```

```
Procedure $applyFunctionalAstrocytes(S)$
  1. Let $S$ be an input/output argument declared consistently
     as specified above
  2. For each functional astrocyte $fast_j \in C_t$ do
       • Let $O \leftarrow \{n_i(e) : e \in ct(fast_j)\}$
       • Let $spikes \leftarrow \sum_{x \in O} O$
       • Let $selector \leftarrow spikes + p(fast_j)$
       • Let $k \leftarrow |T(fast_j)|$
       • Let $r \leftarrow |op(fast_j)|$
       • Let $h \leftarrow \begin{cases} 1 & \text{if} \quad selector < T(fast_j)_1 \\ k & \text{if} \quad selector > T(fast_j)_k \\ e & \text{if} \quad e = \max\{x \mid 1 \le x \le k \wedge T(fast_j)_x \le selector\} \end{cases}$
       • Let $output \leftarrow 0$
       • Let $f^* \leftarrow F(fast_j)_h$
       • If $\omega(fast_j) = true$ Then $output \leftarrow f^*(selector)$
       • If $\omega(fast_j) = false \wedge r = 1$ Then $output \leftarrow f^*(0)$
       • If $\omega(fast_j) = false \wedge r > 1$ Then
         $output \leftarrow f^*(n_i(op(fast_j)_1), \ldots, n_i(op(fast_j)_{r-1}))$
       • For $v = 1$ to $r - 1$ do
           – $n_o(op(fast_j)_v) \leftarrow 0$
           – $o(op(fast_j)_v) \leftarrow a$
           – $inh(op(fast_j)_v) \leftarrow true$
       • $n_o(op(fast_j)_r) \leftarrow n_o(op(fast_j)_r) + output$
       • $o(op(fast_j)_r) \leftarrow a$
       • If $\gamma(fast_j) = true$ Then $p(fast_j) \leftarrow spikes$
       • $S \leftarrow S \cup op(fast_j)$

Procedure $applyHybridAstrocytes(S)$
  1. Let $S$ be an input/output argument declared consistently
     as specified above
  2. For each hybrid astrocyte $hast_j \in C_t$ do
       • Let $O \leftarrow \{n_i(e) : e \in syn(hast_j)\}$
       • Let $sum \leftarrow \sum_{x \in O} O$
       • Let $p$ be a boolean value non-deterministically
         selected from $\{true, false\}$
       • Let $inh \leftarrow \begin{cases} false & \text{if} \quad sum < th(hast_j) \\ true & \text{if} \quad sum > th(hast_j) \\ p & \text{if} \quad sum = th(hast_j) \end{cases}$
       • For each arc $e \in syn(hast_j)$ do
           – Let $inhibitArc \leftarrow inh \vee inh(e)$
```

```
        – If $inhibitArc = true$ Then
            * $inh(e) \leftarrow true$
            * $n_o(e) \leftarrow 0$
          Else
            * $n_o(e) \leftarrow n_i(e)$
        – $o(e) \leftarrow a$
      • $S \leftarrow S \cup syn(hast_j)$
IX. Ending
  1. If $R_{sel} \neq \{\}$ Then
      • Let $C_{t+1} \leftarrow C_t$
      • $toFireNotFiltered \leftarrow \{\}$
      • $toFireFiltered \leftarrow \{\}$
      • $justOpenNow \leftarrow \{\}$
      • $Division \leftarrow \{\}$
      • $Budding \leftarrow \{\}$
      • $Spiking \leftarrow \{\}$
      • $R_{sel} \leftarrow \{\}$
      • Goto II
  2. End
```

9.6 P–Lingua example models

In this Section, some P–Lingua programs representing SN P system models incorporating features of the variants discussed in this Chapter are shown.

9.6.1 Extended spiking, budding and division rules

The following P–Lingua program models a solution to **SAT(n,m)** by means of a family of SN P systems, as discussed in [119].

```
/* instantiation of the family is made by means of MeCoSim */

@model<spiking_psystems>

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

```
}

def spiking_init_conf(n)
{

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

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

/* Encoding initial synapse graph (also updating synapse dictionary) */
@marcs = (d{i},d{i+1}):0<=i<=n-1;
@marcs += (d{n},d{1});
@marcs += (in,Cx{i}):1<=i<=n;
@marcs += (d{i},Cx{i}):1<=i<=n;
@marcs += (Cx{i},Cx{i,0}):1<=i<=n;
@marcs += (Cx{i},Cx{i,1}):1<=i<=n;
@marcs += ({i+1},{i}):0<=i<=2;
@marcs += (1,2);
@marcs += (0,out);

/* Encoding additional synapse dictionary updating */
@mdict = (Cx{i,1},t{i}):1<=i<=n;
@mdict+= (Cx{i,0},f{i}):1<=i<=n;

/* Encoding input neuron
@min = in;

/* Encoding input formula spike train */
@minst+= ((2*n+j)+(n*(i-1)),val{i,j}):1<=i<=m, 1<=j<=n;

/* Encoding output neuron */
@mout = out;

/* Encoding module spiking_rules() */

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

/* Encoding module neuron_division_rules() */

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

/* Encoding module neuron_budding_rules() */

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

9.6.2 Asynchronous modes

Let us consider the following SN P system working on *limited asynchronous mode*, as depicted in Figure 9.5 and discussed in [88]:

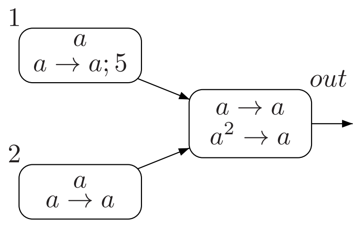


Figure 9.5: An example of a LASNPS Π_1

The following P–Lingua program models system Π_1:

```
@model<spiking_psystems>

def main()
{
       let b = 4;

       call spiking_init_conf(b);
       call spiking_rules(b);
}

def spiking_init_conf(b)
{
       @mu = 1,2,out;

       @ms(1) = a;
       @ms(2) = a;

       @marcs += (1,out);
       @marcs += (2,out);

       @mout = out;

       @masynch = 3;
       @mboundall = b;
}

def spiking_rules(b)
{
       [a --> a]'1 :: 5;
       [a --> a]'2 ;

       [a --> a]'out ;
       [a*2 --> a]'out ;

}
```

Let us consider the following SN P system working on *asynchronous mode with local synchronization*, as depicted in Figure 9.6 and discussed in [88]:
The following P–Lingua program models system Π_2:

```
@model<spiking_psystems>

def main()
{
       let c = 4;

       call spiking_init_conf(c);
       call spiking_rules();
}

def spiking_init_conf(c)
{
       @mu = 1,2,3,4,out;

       @ms(1) += a;
```

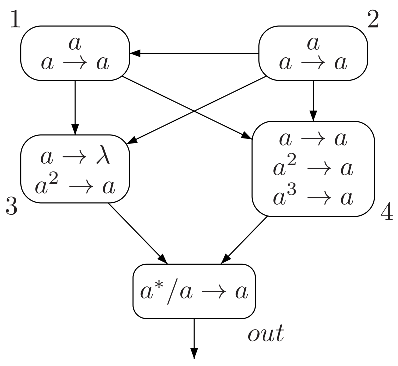


Figure 9.6: An example of a LASNPS Π_2

```
        @ms(2) += a;

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

        @mout = out;

        @masynch = 4;

        @mlocset = {}             :  c <= i <= c, i <> 2, i <> 3, i <> 4;
        @mlocset = {{1,2}}        :  c <= i <= c, i <> 1, i <> 3, i <> 4;
        @mlocset = {{1,3}}        :  c <= i <= c, i <> 1, i <> 2, i <> 4;
        @mlocset = {{1,2},{3,4}}  :  c <= i <= c, i <> 1, i <> 2, i <> 3;
}

def spiking_rules()
{
        [a --> a]'1 ;

        [a --> a]'2 ;

        [a --> #]'3 ;
        [a*2 --> a]'3 ;

        [a --> a]'4 ;
        [a*2 --> a]'4 ;
        [a*3 --> a]'4 ;

        [a --> a]'out "a*";
}
```

9.6.3 Functional Astrocytes

The following P–Lingua program models an AND-gate by means of the SNPSFA depicted in Figure 9.3:

```
@model<spiking_psystems>

def main()
{
call spiking_init_conf();
call spiking_rules();
}

def spiking_init_conf()
{
@mu = A,A1,B,B1,p,q;

@ms(A) = a*6;  /* 6 times */
@ms(B) = a*6;  /* 6 times */

@marcs += (A,A1);
@marcs += (B,B1);
@marcs += (p,q);

@mastfunc = (f(x1), "f(x1) = 0");
/* constant function 0 of arity one */

@mastfunc = (g(x1), "g(x1) = 1");
/* constant function 1 of arity one */

@mastb =
(ast1,{(p,q)},{(A,A1),(B,B1)},false,{1,2},{f(x1),g(x1)},0,false);

@mout = q;
}

def spiking_rules()
{
[a --> a]'A "a+" ;
[a --> #]'A "a+" ;

[a --> a]'B "a+" ;
[a --> #]'B "a+" ;

[a --> a]'q;
}
```

9.6.4 Hybrid Astrocytes

The following P–Lingua program models a SNPSHA corresponding to the ADD module of a register machine, as described in [123]:

```
@model<spiking_psystems>

def main()
{
call spiking_init_conf(n);
call spiking_rules();
}

def spiking_init_conf()
{

/* this machine has only one register r, and a pair of instructions, l0, inital add
    instruction and lh, halt instruction */

@mu = l0,l01,l02,l03;       /* instruction l0 */
@mu += lh,h1,h2,h3,h4,out;  /* instruction lh */
@mu += r1,r2;                       /* register r */

@ms(r) = a*n; /* n is the initial value of r */

@ms(l0) = a; /* l0 is the initial instruction */

/* implementing l0 = ADD(r,lh,lh) */

@marcs =  (l0,r1);
@marcs += (l0,r2);

@marcs += (l0,lh);

@marcs += (l0,l03);
@marcs += (l0,l01);
@marcs += (l0,l02);

@marcs += (l01,l03);
@marcs += (l01,l02);

@marcs += (l01,lh);

@marcs += (r1,r2);
@marcs += (r2,r1);

/* implementing astrocyte for l0 */

@mastw = (astl0,{(l0,lh),(l0,l03),(l01,l03),(l0,l01),(l01,l02),(l01,lh),(l02,lh)},3);

/* implementing lh */

@marcs += (lh,lh1);
@marcs += (lh,lh2);

@marcs += (lh1,out);
@marcs += (lh1,lh2);
@marcs += (lh2,out);
@marcs += (lh2,lh3);
@marcs += (lh2,lh4);

@marcs += (lh3,lh2);
@marcs += (lh3,lh4);

@marcs += (lh4,lh2);

/* implementing astrocytes for lh */

@mastw = (astlh1,{(r1,r2),(r2,r1),(lh2,out),(lh2,lh3)},3);
@mastw = (astlh2,{(lh3,lh2),(lh3,lh4),(lh4,lh2),(lh2,lh4)},3);

/* setting up output neuron */

@mout = out;

}

def spiking_rules()
{

/* every neuron has one single firing rule of the form: [a --> a]'h :: "a*"; */

[a --> a]'l0 :: "a*";
[a --> a]'l01 :: "a*";
[a --> a]'l02 :: "a*";
[a --> a]'l03 :: "a*";

[a --> a]'lh :: "a*";
[a --> a]'lh1 :: "a*";
[a --> a]'lh2 :: "a*";
[a --> a]'lh3 :: "a*";
[a --> a]'lh4 :: "a*";
[a --> a]'out :: "a*";

[a --> a]'r1 :: "a*";
[a --> a]'r2 :: "a*";

}
```

9.6.5 Anti-spikes

The following P–Lingua program models a SNPSAS corresponding to the ADD module of a register machine, as described in [122]:

```
@model<spiking_psystems>

def main()
{
call spiking_init_conf(0);
call spiking_rules();

}

def spiking_init_conf(n)
{

/* this machine has only one register r, and a pair of instructions, l0, inital add
     instruction and lh, halt instruction */

@mu = l0,l01,l02,l03,l04,l05;       /* instruction l0 */
@mu += lh,h1,h2,h3,h4,h5,h6,1,out;  /* instruction lh */
@mu += r;                                           /* register r */

@ms(r) = a*(n+2); /* n is the initial value of r */

@ms(l0) = a; /* l0 is the initial instruction */

@ms(1) = a*2; /* l0 is the initial instruction */

/* implementing l0 = ADD(r,lh,lh) */

@marcs = (l0,r);

@marcs += (l0,l01);
@marcs += (l0,l02);
@marcs += (l0,l03);

@marcs += (l01,l04);
@marcs += (l01,l05);

@marcs += (l02,l05);

@marcs += (l03,l04);
@marcs += (l03,l05);

@marcs += (l04,lh);

@marcs += (l05,lh);

/* implementing lh */

@marcs += (lh,h1);
@marcs += (lh,h2);

@marcs += (h1,h4);

@marcs += (h2,h1);
@marcs += (h2,h3);
@marcs += (h2,h5);

@marcs += (h3,h6);

@marcs += (h4,h1);
@marcs += (h4,h5);

@marcs += (h5,1);

@marcs += (h6,out);

@marcs += (1,h1);
@marcs += (1,h4);
@marcs += (1,h5);
@marcs += (1,out);

/* setting up output neuron */

@mout = out;
}

def spiking_rules()
{
/* implementing l0 = ADD(r,lh,lh) */

[a --> a]'l0;

[a --> a]'l01;

[a --> a]'l02;

[a --> a]'l03;
[a --> _a]'l03;

[a*2 --> a]'l04;

[a --> a]'l05;
[a*3 --> #]'l05;

/* implementing lh */

[a --> a]'lh;
[a --> a]'h1;
[a --> a]'h2;
[a --> a]'h3;
[a --> a]'h4;
[a --> _a]'h5;
[a --> a]'h6;
[a --> a]'1;
[a --> a]'out;

}
```

10
Efficient simulation of Fuzzy Reasoning SN P systems

10.1 Introduction

In Chapter 5 Fuzzy Reasoning SN P systems (FRSN P systems, for short) were discussed. These systems incorporate fuzzy logic elements, as they are intended to model the *fuzzy diagnosis knowledge and reasoning* associated to tackling real-life problems involving uncertain knowledge. In this sense, FRSN P systems have shown promising applications in the engineering field, addressing problems like fault diagnosis in electric power systems 5.

To date, five types of FRSN P systems have been proposed: fuzzy reasoning spiking neural P systems with real numbers (rFRSN P systems, [130, 181]), fuzzy reasoning spiking neural P systems with linguistic terms (lFRSN P systems, [175]), adaptive fuzzy reasoning spiking neural P systems with real numbers (AFRSN P systems, [176]), weighted fuzzy reasoning spiking neural P systems (WFRSN P systems, [177]) and fuzzy reasoning spiking neural P systems with trapezoidal fuzzy numbers (tFRSN P systems, [179]).

Each of these variants have their own specificities and are suitable to address different sets of problems. Nevertheless, they share an important feature: their models semantics can be simulated by means of matrix-based algorithms. As such, they are susceptible of being simulated on High Performance Computing platforms (HPC, for short, see Sections 6.5 and 6.6), specially intended to simultaneously work with hundreds to millions of data stored in matrices. In this way, simulation on these architectures can take advantage of the corresponding execution speedup, thus fully exploiting the models maximally parallel capacities and, consequently, enabling attacking real-life size instance problems.

To the best knowledge of the author, no simulation tools for FRSN P systems, either of the *sequential* or *parallel* kind, have been made publicly available for the scientific community. Due to the potential interesting applications related to these systems, specially in terms of tackling relevant problems, a key goal of the work object of this dissertation has been providing a *first public parallel simulation tool* for FRSN P systems, with a first version of such tool successfully simulating rFRSN P systems instances on CUDA-enabled devices [192].

The aforementioned tool has a rather special feature, since it consist in a *hybrid sequential/parallel simulator* included into pLinguaCore library which, while externally behaving as any other existing (sequential) simulator in the library, has the ability of making calls to native CUDA kernels executed on an underlying GPU architecture.

This Chapter deals the efficient simulation of rFRSN P systems by means of the developed hybrid simulator. Section 10.2 details the design guidelines of the simulator. Section 10.3 covers the syntax for specifying rFRSN P systems in the P–Lingua programming language. Section 10.4 details the simulation algorithm for rFRSN P systems implemented in the simulator. Finally, Section 10.5 discusses the simulator validation process as well as a brief performance analysis.

10.2 Simulator design guidelines

As previously stated, providing a simulation tool for FRSN P systems has been addressed as part of the work object of this dissertation. In order to accomplish this task, the following challenges were identified:

(1) Selecting FRSN P systems variant(s) to consider.
(2) Available HPC platforms where the simulator could be developed for.
(3) Designing a proper software architecture for the simulator, providing (a) integration mechanisms with existing Membrane Computing software applications and (b) a friendly user interface.

With respect to (1), an analysis of the different variants was conducted, resulting in selecting rFRSN P systems as a suitable candidate for a first version of the simulation tool, since (a) it is the simplest model and (b) the corresponding matrix-based simulation algorithm is adequately well-specified [130]. An incremental development cycle was devised, considering the incorporation of additional FRSN P systems variants in future versions of the simulator.

With respect to (2), GPU based simulation on CUDA [192] platform was selected, since this technology provides a powerful HPC architecture built into relatively cheap commercial graphic cards. Moreover, 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 [74], 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 17x of speedups for some applications [192]. Finally, it is worth mentioning the extensive amount of available documentation on CUDA programming model, and specifically on optimization techniques (see [74, 202] for example).

With respect to (3), an architecture involving a *hybrid* JAVA-CUDA design was devised, as depicted in Figure 10.1. In this way, parsing of rFRSN P systems is conducted by means of pLinguaCore library, thus enabling specification of FRSN P systems in the well-known P–Lingua programming language. On the other hand, simulation of rFRSN P systems is conducted in a JAVA-CUDA hybrid way, as follows. A Java stub simulator is included into pLinguaCore library, which in turn performs the matrix-based operations on the GPU by making calls to kernel methods designed to be able to run on the majority of CUDA-compatible devices. In order for the Java stub simulator to be able to access the underlying CUDA hardware, an open source library called `JCUDA` [196] is used. In this sense, JCUDA library provides binding methods that allows calling CUDA kernels directly from JAVA code. JCUDA is available for Windows, Linux, MacOS and other operating systems. Version 7.0 of the library has been used to develop the rFRSN P systems simulator.

It is easy to see that this architecture fulfills the requirements regarding connectivity with other software applications and the friendly user interface, since the simulator is integrated within pLinguaCore library, which in turns can be connected for instance with tools like MeCoSim [197].

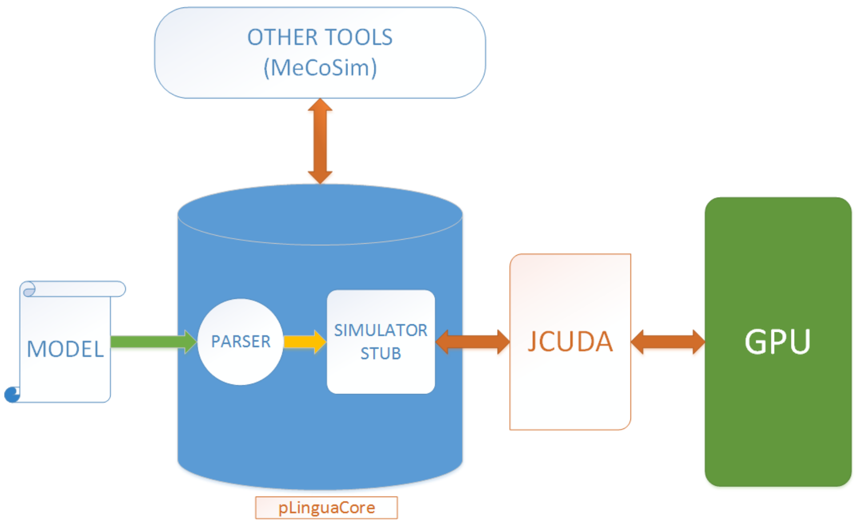


Figure 10.1: Architecture of the hybrid simulator

10.3 P–Lingua syntax for rFRSNP systems

Next, we detail the P–Lingua syntax for specifying rFRSN P systems in P–Lingua. The starting point for this syntax is the one defined for SN P systems (see Section 9.4), which is partially reused. Let us stress the fact that, with respect to P–Lingua, rFRSN P system variant is considered as separate model from SN P systems.

10.3.1 Reserved words

The set of reserved words has been updated by adding the following text strings:

```
@fvariant, @frule, @fpin, @fpout, @fand, @for, @parallel
```

A detailed explanation about this set of reserved words is provided in what follows.

10.3.2 Model specification

Although at present FRSN P systems support into P–Lingua only involves rFRSN P systems, it is convenient to provide a general extensible method for defining any other kind of fuzzy model. This is accomplished as follows.

Firstly, any P–Lingua file defining a FRSN P system must begin with the following sentence:

```
@model<fuzzy_psystems>
```

Secondly, in order to specify the kind of FRSN P system being defined, the following sentence must written:

```
@fvariant = v;
```

where `v` is a non-negative integer specifying the variant. In the case of rFRSN P systems, `v` must set to `1`, hence `@fvariant = 1;`.

The `@fvariant` sentence must be the first instruction included into the `main` module.

If the model is to be simulated in a *hybrid sequential/parallel* on a `CUDA` underlying architecture, the following instruction must be written under the `@fvariant` sentence:

```
@parallel
```

If this sentence is not included, a sequential simulation is performed.

10.3.3 Membrane structure

A rFRSN P system specification in P–Lingua must define a membrane structure, which is composed of a set of proposition neurons interconnected with rule neurons. Also, input and output proposition neurons has to be specified.

Proposition neurons

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

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

where p_i is the label of the i-th proposition neuron.

Input proposition neurons

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

```
@min = i1,...,iq,...,is;
```

where $i_q = (p_q, n_q)$ is an expression specifying the initial fuzzy value $n_q \in [0,1]$ for the input proposition neuron of label p_q, which must be declared in the `@mu` instruction.

Output proposition neurons

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

```
@mout = o1,...,ow,...,od;
```

where o_w is the label of the output proposition neuron of label p_w, which must be declared in the `@mu` instruction.

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 = τ_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 = τ_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 = τ_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 = τ_i)

  are written as:

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

10.3.4 A simple example

Next we illustrate the syntax presented above with the specification the rFRSN P system discussed in Section 5.5.

```
@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.

10.4 An algorithm for simulating rFRSN P systems in P–Lingua

In what follows, we present a simulation algorithm for rFRSN P systems. In general, simulation algorithms capture semantics of the simulated models, reproducing one or many of the associated computations. In the case of rFRSN P systems, since they 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 [130]. This new version re-defines the matrix-based functions and operations as well as provides an alternative way to compute fuzzy truth values for rule neurons that fits semantics introduced for rFRSN P systems as defined in Chapter 5. Let us recall that, since it is a matrix-based algorithm, it is specially suitable to run on parallel platforms such a CUDA-enabled devices.

10.4.1 Required notation

Next we introduce some required notation, as well as functions and operations, which derives from [130].

Let $\Pi = (A, \sigma_1, \dots, \sigma_{n+k}, syn, I, O)$ 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 $\sigma = \{\sigma_1, \dots, \sigma_{n+k}\}$, composed of n proposition neurons and k rule neurons;
2. The set of n proposition neurons $\sigma_p = \{\sigma_{p1}, \dots, \sigma_{pn}\}$;
3. The set of k rule neurons $\sigma_r = \{\sigma_{r1}, \dots, \sigma_{rk}\}$, each of them being either an AND-type or OR-type rule neuron;
4. The set $I = \{\sigma_{i_1}, \dots, \sigma_{i_s}\}$ of input proposition neurons, corresponding to fuzzy proposition neurons which fuzzy truth values are known;
5. The set $O = \{\sigma_{o_1}, \dots, \sigma_{o_d}\}$ 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. $\Lambda = diag(\tau_{r1}, \dots, \tau_{rk})$ is a diagonal real matrix, where τ_{rj} represents the confidence factor of the j-th production rule, which is associated with rule neuron σ_{rj};
4. $H_1 = diag(h_1, \dots, h_k)$ 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. $H_2 = diag(h_1, \dots, h_k)$ 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. $\alpha_p = (\alpha_{p1}, \dots, \alpha_{pn})^T$ is a truth value vector, where $\alpha_{pi} \in [0,1]$ represents the truth value of i-th proposition neuron σ_{pi};
7. $\alpha_r = (\alpha_{r1}, \dots, \alpha_{rk})^T$ is a truth value vector, where $\alpha_{rj} \in [0,1]$ represents the truth value of j-th rule neuron σ_{rj};
8. $a_p = (a_{p1}, \dots, a_{pn})^T$ is an integer vector, where a_{pi} represents the number of spikes received by the i-th proposition neuron σ_{pi};
9. $a_r = (a_{r1}, \dots, a_{rk})^T$ is an integer vector, where a_{rj} represents the number of spikes received by the j-th rule neuron σ_{rj};
10. $\lambda_p = (\lambda_{p1}, \dots, \lambda_{pn})^T$ is an integer vector, where λ_{pi} represents the number of spikes required to fire the i-th proposition neuron σ_{pi};
11. $\lambda_r = (\lambda_{r1}, \dots, \lambda_{rk})^T$ is an integer vector, where λ_{rj} represents the number of spikes required to fire the j-th rule neuron σ_{rj};

12. $\beta_p = (\beta_{p1}, \dots, \beta_{pn})^T$ is a truth value vector, where $\beta_{pi} \in [0,1]$ represents the truth value exported by the i-th proposition neuron σ_{pi} after firing;

13. $\beta_r = (\beta_{r1}, \dots, \beta_{rk})^T$ is a truth value vector, where $\beta_{rj} \in [0,1]$ represents the truth value exported by the j-th rule neuron σ_{rj} after firing;

14. $b_p = (b_{p1}, \dots, b_{pn})^T$ is an integer vector, where $b_{pi} \in \{0,1\}$ represents the number of spikes exported by the i-th proposition neuron σ_{pi} after firing;

15. $b_r = (b_{r1}, \dots, b_{rk})^T$ is an integer vector, where $b_{rj} \in \{0,1\}$ represents the number of spikes exported by the j-th rule neuron σ_{rj} after firing;

16. $o_p = (o_{p1}, \dots, o_{pn})^T$ is a binary vector, where $o_{pi} \in \{0,1\}$, defined as follows:
$$o_{pi} = \begin{cases} 1 & \text{if } \mathit{outdegree}(\sigma_{pi}) > 0; \\ 0 & \text{otherwise;} \end{cases}$$

17. $o_r = (o_{r1}, \dots, o_{rk})^T$ is a binary vector, where $o_{rj} \in \{0,1\}$, defined as follows:
$$o_{rj} = \begin{cases} 1 & \text{if } \mathit{outdegree}(\sigma_{rj}) > 0; \\ 0 & \text{otherwise;} \end{cases}$$

Let us consider the following matrix functions:

1. diag: $D = \mathit{diag}(b)$, where $D = (d_{i,j})$ is a $f \times f$ diagonal real matrix and $b = (b_1, \dots, b_f)$ a real vector, such that
$$d_{i,j} = \begin{cases} b_i & \text{if} \quad i = j \\ 0 & \text{if} \quad i \neq j \end{cases}, 1 \leq i,j \leq f;$$

2. fire: $\beta = \mathit{fire}(\alpha, a, \lambda, o)$, where $\beta = (\beta_1, \dots, \beta_f)^T, \alpha = (\alpha_1, \dots, \alpha_f)^T$, $a = (a_1, \dots, a_f)^T, \lambda = (\lambda_1, \dots, \lambda_f)^T, o = (o_1, \dots, o_f)^T$, such that
$$\beta_i = \begin{cases} 0 & \text{if} \quad a_i < \lambda_i \\ \alpha_i & \text{if} \quad a_i = \lambda_i \wedge o_i = 0 \\ 0 & \text{if} \quad a_i = \lambda_i \wedge o_i = 1 \end{cases}, 1 \leq i \leq f;$$

3. update: $\beta = \mathit{update}(\alpha, a, \lambda, o)$, where $\beta = (\beta_1, \dots, \beta_f)^T, \alpha = (\alpha_1, \dots, \alpha_f)^T$, $a = (a_1, \dots, a_f)^T, \lambda = (\lambda_1, \dots, \lambda_f)^T, o = (o_1, \dots, o_f)^T$, such that
$$\beta_i = \begin{cases} 0 & \text{if} \quad a_i = 0 \\ \alpha_i & \text{if} \quad 0 < a_i < \lambda_i \\ 0 & \text{if} \quad a_i = \lambda_i \wedge o_i = 0 \\ \alpha_i & \text{if} \quad a_i = \lambda_i \wedge o_i = 1 \end{cases}, 1 \leq i \leq f;$$

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

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

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

10.4.2 Simulation algorithm

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

- INPUT:
  - $U, V, \Lambda, H_1, H_2, \lambda_p, \lambda_r$;

- $\alpha_p^0 = (\alpha_{p1}^0, \ldots, \alpha_{pn}^0)$, with $\alpha_{pi}^0 = \begin{cases} \tau_{pi} & \text{if } \sigma_{pi} \in I,\ \tau_{pi} \text{ is the CF of } \sigma_{pi}; \\ 0 & \text{otherwise;} \end{cases}$
- $a_p^0 = (a_{p1}^0, \ldots, a_{pn}^0)$, with $a_{pi}^0 = \begin{cases} 1 & \text{if } \sigma_{pi} \in I; \\ 0 & \text{otherwise;} \end{cases}$

- OUTPUT:
  - $\alpha_{out} = (\alpha_{i_1}, \ldots, \alpha_{i_s})^T$, 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: $a_{pi} = \begin{cases} 1 & \text{if } o_{pi} = 1 \\ 0 & \text{otherwise} \end{cases}, 1 \leq i \leq n$;

then HALT, otherwise go to Step 3.

10.5 Validation and performance analysis

In what follows, validation and performance analysis of the developed simulator are discussed. As usual when developing CUDA based applications, validation of the hybrid simulator involves checking simulation results of concrete rFRSN P systems instances against pure sequential simulators capturing these variants semantics. Consequently, besides of the hybrid simulator, that we denominate `rFRSNPS-hybrid`, two sequential simulators for rFRSN P systems has been developed, a JAVA based one included into pLinguaCore, named `rFRSNPS-JAVA`, and a C++ based one, named `rFRSNPS-C++`, deployed as a stand-alone application. Additionally, due to the special hybrid nature of the `rFRSNPS-hybrid` architecture, a fourth pure CUDA simulator, named `rFRSNPS-CUDA` has been developed, also as a stand-alone application, for further validation purposes. All the aforementioned simulators implement the simulation algorithm described in Section 10.4.

10.5.1 Simulator validation

The validation process has involved the following:

(1) Consider a set of rFRSN P systems a test cases.
(2) Validate `rFRSNPS-C++`.
(3) Validate `rFRSNPS-JAVA` against `rFRSNPS-C++`.
(4) Validate `rFRSNPS-CUDA` against `rFRSNPS-C++`.
(5) Validate `rFRSNPS-hybrid` against `rFRSNPS-CUDA`.
(6) Validate `rFRSNPS-hybrid` against `rFRSNPS-JAVA`.

With respect of the test cases referred in (1), literature examples such as the example model considered in Section 10.3 have been used.

Due to the use of JCUDA library, setting a suitable execution environment for `rFRSNPS-hybrid` differs from the usual way of simulating other P systems variants by means of pLinguaCore library.

10.5.2 Simulation environment for `rFRSNPS-JAVA`

Invoking `rFRSNPS-JAVA` simulator requires for the system to host a Java runtime environment properly installed and configured. The Java runtime can be found at [195]. 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 rFRSN P systems 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.

10.5.3 Simulation environment for `rFRSNPS-hybrid`

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 [198].
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 this work, version 0.7 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 [196]. Also, the following directory structure must be created:

```
plingua/
 |__plinguacore.jar
 |__input.pli
 |__kernelReal.cu
 |__jcudaUtils-0.0.4.jar
 |__jcuda-0.7/
     |__*** jcuda-0.7 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 rFRSN P systems 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-0.7` folder hosts the contents of the zip file corresponding to the 0.7 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-0.7.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 `":"`.

10.5.4 Performance analysis summary

When developing the parallel hybrid simulator, one of the design constraints was to make it able to run on the majority of CUDA-compatible devices, while handling arbitrary matrix size instances. Such constraints have involved making some conservative choices in the implementation, which may affect the overall expected speedup in certain situations.

In this way, a standard *block size* equal to 256 (16*16) has been used, which is a good average value for most of cases, but may be unsuitable in some cases [74]. On the other hand, with respect to optimization techniques, the *tiling/memory coalescing* technique has been applied, which requires a relatively low amount of *shared memory* for blocks (see [74] for more details). It is worth pointing out that no matrix handling optimized libraries such as `cuBLAS` has been used, since the simulator is intended to run on *bare* execution environments.

Nevertheless, it is rather difficult at present to evaluate the real performance of the developed simulator. To date, model instances corresponding to real-life problems have not been available, preventing obtaining a reliable speedup value. Consequently, further collaboration efforts have to been taken with experts in the field of fault diagnosis in order to access the required data.

Once this data is obtained and the corresponding speedup calculated, further optimization of the simulator can be also considered. For instance, 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 [74]. Automation tools to reduce such efforts such as `CUDA`-lite [172] and others become indispensable.

Part III

Conclusions and future work

11 Conclusions and future work

This chapter puts an end to the this dissertation by summarizing the overall conclusions taken from the main contributions provided as a result of this work, as well as outlining analyzing future work and open research lines.

11.1 Antecedents

Membrane Computing, introduced by Gh. Păun at the end of 1998, is a relatively young branch of Natural Computing providing non-deterministic distributed parallel computing models whose computational devices are called *membrane systems or P systems*. These systems are inspired by some basic biological features, specifically by the structure and functioning of the living cells, as well as from the way the cells are organized in tissues, organs, and organisms.
There are basically three ways to categorize membrane systems: cell-like P systems ([148]), tissue–like P systems ([92, 93]) and neural-like P systems ([67]), also called Spiking Neural P systems (SN P systems, for short). Cell-like P systems arrange a series of membranes in a hierarchical way, inspired by the inner structure of the biological cells. Tissue-like P systems arrange elemental membranes in nodes of a directed graph, inspired from the cell inter-communication in tissues. Similarly, Spiking Neural P systems also arrange elemental membranes in nodes of a directed graph, while taking inspiration from the way in

which neurons exchange information by the transmission of electrical impulses (spikes) along axons.
In general, P systems operate by applying rewriting rules defined over multisets of objects associated to the different membranes, in a synchronized non-deterministic maximally parallel way. P systems show a double level of parallelism: a first level comprises parallel application of rules within individual membranes, while a second level comprises all the membranes working simultaneously, that is, in parallel. These features make P systems powerful computing devices. In particular, the double level of parallelism allows a space-time tradeoff enabling the generation of an exponential workspace in polynomial time. As such P systems are suitable to tackle relevant real-life problems, usually involving **NP**-complete problems. Moreover, P systems are excellent tools to investigate on the computational complexity boundaries, in particular tackling the P versus NP problem. In this way, by studying how the ingredients relative to their syntax and semantics affect to their ability to efficiently solve **NP**-complete problems, sharper frontiers between efficiency and non-efficiency can be discovered.
Despite of their attractive properties, working with P systems immediately drives to an important inconvenient: due to constraints of current technology, P systems are yet to be fully implemented *in vivo*, *in vitro*, or even *in silico*, because of their massively parallel, distributed, and non-deterministic nature. Thus, practical computations of P systems are driven by silicon-based simulators, and hence their potential results are compromised by the physical limitations of silicon architectures. They are often inefficient or not suitable when dealing with some P system features, such as the exponential workspace creation, non-determinism and massive parallelism.
Consequently, developing efficient simulation tools for P systems becomes an indispensable task in order to both assist in the computational complexity study involving such systems, as well as in the development and verification of solutions to relevant real-life problems.

11.2 Contributions summary

The main contributions derived from the work object of this dissertation are the following:

- Finding sharper computational complexity boundaries by modelling solutions to **NP**-complete problems in terms of cell-like P systems in **CDC** and **CSC**.

- Developing a simulation tools for membrane systems in **CDC** and **CSC** within the P–Lingua framework. These tools have played a major role as assistants in the specification and formal verification of the aforementioned solutions.

- Defining new SN P systems variants and the corresponding simulation tools, within the P–Lingua framework. Also, simulation support for a wide range of existing SN P systems variants has been included into that framework.

- Defining efficient simulation tools for Fuzzy Reasoning SN P systems working on High Performance Computation platforms, namely CUDA-enabled devices.

11.3 Conclusions

The following main conclusions derive from the conducted research to produce the aforementioned contributions:

- Developing efficient solutions to **NP**-complete problems by means of P systems involves handling computational models designs of such complexity that a manual process of design and verification proves an unaffordable task. Consequently, suitable simulation tools are indispensable for debugging and experimental validation purposes.

- A significant lack of simulation tools for SN P systems has been a major drawback in the research on these variants in recent years, when they have become a hot topic. As such, developing simulators addressing as many SN P systems variants as possible greatly favours the development of this field in the near future.

- Fuzzy Reasoning P systems hold a relevant position within the membrane systems variants ecosystem, since they have shown promising applications related to tackling real-life fault diagnosis problems in industrial systems. Furthermore, these variants are suitable to be naturally simulated on High Performance Computation platforms, due to their associated matrix-based simulation algorithms. Consequently, developing efficient simulation tools working on parallel devices such as graphic cards enable addressing medium size instances of real-life problems at a relatively cheap cost.

11.4 Future work and open research lines

The following future work and open research lines can be identified in relation to the tasks accomplished within the object of this dissertation:

- Furher work has to be conducted in the quest for finding complexity boundaries, possibly by means of other computational devices, since this may lead to significant achievements with an impact in many aspects of our society.
- Design and validation process of complex computational models is still a very unpleasant and time-consuming task. Consequently, a rather interesting research line would involve developing specific Integrated Development Environments to address that problem. A suitable degree of sophistication is required, involving features such as a rich Graphic User Interface, modular design support and connections to popular simulation frameworks (e.g. P–Lingua).
- The unavailability of public-access data regarding real-life fault diagnosis problems in industrial systems constitutes a major drawback when conducting modelling research on such type of problems. Moreover, that issue has prevented a suitable measurement of the speedup achieved for the developed rFRSN P system parallel simulator related to the work object of this dissertation. Consequently, a further effort has to be conducted to set joint collaboration ventures with experts from the aforementioned domain.

Bibliography

[1] B. Alberts, D. Bray, K. Hopkin, A. Johnson, J. Lewis, and M. Raff. *Essential Cell Biology. An introduction to the molecular biology of the cell. Third Edition.* Garlan Science, Taylor and Francis Group, New York and London, 2010.

[2] A. Alhazov and T.-O. Ishdorj. Membrane operations in p systems with active membranes. In A. R.-N. nez, A. Romero-Jiménez, and F. Sancho-Caparrini, editors, *Proceedings of the Second Brainstorming Week on Membrane Computing, Sevilla, 2-7 February 2004*, pages 37–44. Research Group on Natural Computing, University of Seville, 2004.

[3] A. Alhazov, C. Martín-Vide, and L. Pan. Solving a pspace-complete problem by recognizing P systems with restricted active membranes. *Fundam. Inform.*, 58(2):67–77, 2003.

[4] A. Alhazov, C. Martín-Vide, and L. Pan. Solving graph problems by P systems with restricted elementary active membranes. In N. Jonoska, G. Paun, and G. Rozenberg, editors, *Aspects of Molecular Computing, Essays Dedicated to Tom Head on the Occasion of His 70th Birthday*, volume 2950 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2004.

[5] A. Alhazov, L. Pan, and G. Păun. Trading polarizations for labels in P systems with active membranes. *Acta Inf.*, 41(2-3):111–144, 2004.

[6] A. Alhazov and M. J. Pérez-Jiménez. Uniform solution of **qsat**. In J. O. Durand-Lose and M. Margenstern, editors, *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007, Proceedings*, volume 4664 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2007.

[7] A. Araque, V. Parpura, R.-P. Sanzgiri, and P.-G. Haydon. Tripartite synapses: glia, the unacknowledged partner. *Trends in Neuroscience*, 22(5):208–215, 1999.

[8] I. Ardelean and M. Cavaliere. Modelling biological processes by using a probabilistic p system software. *Natural Computing*, 2(2):173–197, 2003.

[9] F. Arroyo, C. Luengo, A. V. Baranda, and L. F. Mingo. A software simulation of transition P systems in haskell. In *Membrane Computing, International Workshop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002, Revised Papers*, pages 19–32, 2002.

[10] V. Authors. 68030 features Harvard architecture. *Microprocessors and Microsystems*, 10(10):567, 1986.

[11] D. Balbontín-Noval, M. Pérez-Jiménez, and F. Sancho-Caparrini. A MzScheme implementation of transition P systems. In G. Păun, G. Rozenberg, A. Salomaa, and C. Zandron, editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 58–73. Springer Berlin Heidelberg, 2003.

[12] L. Bianco, F. Fontana, G. Franco, and V. Manca. P Systems for Biological Dynamics. In G. Ciobanu, G. Păun, and M. J. Pérez-Jiménez, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 83–128. Springer Berlin Heidelberg, 2006.

[13] A. Binder, R. Freund, M. Oswald, and L. Vock. Extended spiking neural P Systems with excitatory and inhibitory astrocytes. In *Proceedings of the 8th Conference on 8th WSEAS International Conference on Evolutionary Computing - Volume 8*, EC'07, pages 320–325, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[14] J. Blakes, J. Twycross, S. Konur, F. J. Romero-Campero, N. Krasnogor, and M. Gheorghe. Infobiotics Workbench: A P Systems Based Tool for Systems and Synthetic Biology. In *Applications of Membrane Computing in Systems and Synthetic Biology*, volume 7 of *Emergence, Complexity and Computation*, pages 1–41. Springer International Publishing, 2014.

[15] F. G. Cabarle, H. Adorna, M. A. Martínez-del Amor, and M. Pérez-Jiménez. Spiking neural P system simulations on a high performance

GPU platform. In Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, editors, *Algorithms and Architectures for Parallel Processing*, volume 7017 of *Lecture Notes in Computer Science*, pages 99–108. Springer Berlin Heidelberg, 2011.

[16] F. G. Cabarle, H. N. Adorna, M. A. Martínez-del-Amor, and M. J. Pérez-Jiménez. Improving GPU simulations of spiking neural P systems. *Romanian Journal of Information Science and Technology*, 15:5–20, 2012.

[17] M. Cardona, M. A. Colomer, A. Margalida, A. Palau, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and D. Sanuy. A computational modeling for real ecosystems based on p systems. *Natural Computing*, 2010.

[18] A. Castellini and V. Manca. Metaplab: A computational framework for metabolic p systems. In *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin Heidelberg, 2009.

[19] M. Cavaliere and I. Ardelean. Modeling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria Using a P System Simulator. In G. Ciobanu, G. Păun, and M. J. Pérez-Jiménez, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 129–158. Springer Berlin Heidelberg, 2006.

[20] M. Cavaliere, O. Egecioglu, O. H. Ibarra, M. Ionescu, G. Păun, and S. Woodworth. Asynchronous spiking neural p systems: Decidability and undecidability. In M. Garzon and H. Yan, editors, *DNA Computing*, volume 4848 of *Lecture Notes in Computer Science*, pages 246–255. Springer Berlin Heidelberg, 2008.

[21] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs. *The Journal of Logic and Algebraic Programming*, 79(6):317 – 325, 2010. Membrane computing and programming.

[22] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del Amor, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. Simulation of P systems with active membranes on CUDA. *Briefings in Bioinformatics*, 11(3):313–322, 2010.

[23] J. M. Cecilia, J. M. García, G. D. Guerrero, M. A. Martínez-del Amor, M. J. Pérez-Jiménez, and M. Ujaldón. The GPU on the simulation of cellular computing models. *Soft Computing*, 16(2):231–246, 2012.

[24] H. Chen, M. Ionescu, T.-O. Ishdorj, A. Păun, G. Păun, and M. J. Pérez-Jiménez. Spiking neural p systems with extended rules: universality and languages. *Natural Computing*, 7:147–166, 06/2008 2008.

[25] S. Chen. A fuzzy reasoning approach for rule-based systems based on fuzzy logics. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 26(5):769–778, 1996.

[26] S. Cheruku, A. Păun, F. J. Romero-Campero, M. J. Pérez-Jiménez, and O. H. Ibarra. Simulating FAS-induced apoptosis by using P systems. *Progress in Natural Science*, 17(4):424–431, 2007.

[27] G. Ciobanu and D. Paraschiv. P system software simulator. *Fundamenta Informaticae*, 49(1):61–66, 2002.

[28] G. Ciobanu and G. Wenyuan. P systems running on a cluster of computers. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin Heidelberg, 2004.

[29] M. A. Colomer, A. Margalida, D. Sanuy, and M. J. Pérez-Jiménez. A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. *Ecological Modelling*, 222:33–47, 01/2011 2011.

[30] Congrace Developer Team. The exp4j website.
`http://projects.congrace.de/exp4j/`.

[31] A. Cordón-Franco, M. Gutiérrez-Naranjo, M. Pérez-Jiménez, and F. Sancho-Caparrini. A Prolog simulator for deterministic P systems with active membranes. *New Generation Computing*, 22(4):349–364, 2004.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[33] F. Courteille, A. Crouzil, J.-D. Durou, and P. Gurdjos. 3d-spline reconstruction using shape from shading: Spline from shading. *Image and Vision Computing*, 26(4):466 – 479, 2008.

[34] M. A. Covington. Antialiasing on the IBM PS2 VGA by treating color bits as subpixels. *Journal of Microcomputer Applications*, 12(3):253 – 257, 1989.

[35] M. A. M. del Amor, J. Pérez-Carrasco, and M. J. Pérez-Jiménez. Simulating a family of tissue P systems solving SAT on the GPU. In *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*, pages 201–220. Fenix Editora, 2013.

[36] D. Díaz-Pernil, C. Graciani-Díaz, M. A. Gutiérrez-Naranjo, I. Pérez-Hurtado, and M. J. Pérez-Jiménez. *The Oxford Handbook of Membrane Computing*, chapter Software for P systems, pages 437–454. Volume 1 of Păun et al. [157], 2010.

[37] D. Díaz-Pernil, M. Gutiérrez-Naranjo, M. Pérez-Jiménez, and A. Riscos-Núñez. A logarithmic bound for solving subset sum with P systems. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 257–270. Springer Berlin Heidelberg, 2007.

[38] D. Dıaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. P-lingua: A programming language for membrane computing. *Sixth Brainstorming Week on Membrane Computing*, pages 135–155, 2008.

[39] D. Díaz-Pernil, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-lingua Programming Environment for Membrane Computing. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009.

[40] D. Díaz-Pernil, M. J. Pérez-Jiménez, and Á. R. Jiménez. Efficient simulation of tissue-like P systems by transition cell-like P systems. *Natural Computing*, 8(4):797–806, 2009.

[41] L. Diez Dolinski, R. Núñez Hervás, M. Cruz Echeandía, and A. Ortega. Distributed simulation of P systems by means of Map-Reduce: First steps with hadoop and P-Lingua. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*, volume 6691 of *Lecture Notes in Computer Science*, pages 457–464. Springer Berlin Heidelberg, 2011.

[42] A. El-Kateeb. Hardware switch for DMA transfer to augment CPU efficiency. *Microprocessors and Microsystems*, 7(3):117 – 120, 1983.

[43] M. García-Quismondo. *Modelling and simulation of real-life phenomena in Membrane Computing.* PhD thesis, University of Seville, 11 2013.

[44] M. García-Quismondo, R. Gutiérrez-Escudero, M. A. Martínez-del-Amor, E. Orejuela-Pinedo, and I. Pérez-Hurtado. P-Lingua 2.0: a software framework for cell-like P systems. *International Journal of Computers, Communications and Control*, 4(3):234–243, 2009.

[45] M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0. In G. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 264–288. Springer Berlin Heidelberg, 2010.

[46] M. García-Quismondo, D. Lobo-Fernández, and M. Levin. Modeling regenerative processes with membrane computing. *Information Sciences*, page In submission, 2016.

[47] M. García-Quismondo, L. F. Macias-Ramos, and M. J. Pérez-Jiménez. *Implementing Enzymatic Numerical P Systems for AI Applications by Means of Graphic Processing Units*, volume 4 of *Topics in Intelligent Engineering and Informatics*, pages 137–159. Springer Berlin Heidelberg, 2013.

[48] M. Garcıa-Quismondo, A. B. Pavel, and M. J. Pérez-Jiménez. Simulating Large-Scale ENPS Models by Means of GPU. In *Proc. Tenth Brainstorming Week on Membrane Computing, Sevilla, Spain*, volume 1, pages 137–152, 2012.

[49] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[50] M. Gheorghe, F. Ipate, and C. Dragomir. A kernel P system. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume I, pages 153–170. Fenix editora, 2012.

[51] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierla, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez. Kernel P Systems - Version I. *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*, pages 97–124, 08/2013 2013.

[52] M. Gheorghe, F. Ipate, R. Lefticaru, M. J. Pérez-Jiménez, A. Turcanu, L. Valencia Cabrera, M. García-Quismondo, and L. Mierla. 3-col problem modelling using simple kernel P systems. *International Journal of Computer Mathematics*, 90(4):816–830, 2013.

[53] O. Goldreich. *Computational Complexity: A Conceptual Perspective.* Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[54] S. Gottwald. Shaping the logic of fuzzy set theory. *Witnessed Years: Essays in Honour of Petr Hájek*, pages 193–208, 2009.

[55] W. Guo, F. Wen, G. Ledwich, Z. Liao, X. He, and J. Liang. An analytic model for fault diagnosis in power systems considering malfunctions of protective relays and circuit breakers. *Power Delivery, IEEE Transactions on*, 25(3):1393–1401, 2010.

[56] A. Gutiérrez, L. Fernández, F. Arroyo, and V. Martínez. Design of a hardware architecture based on microcontrollers for the implementation of membrane systems. In *Symbolic and Numeric Algorithms for Scientific Computing, 2006. SYNASC '06. Eighth International Symposium on*, pages 350–353, 2006.

[57] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A fast P system for finding a balanced 2-partition. *Soft Comput.*, 9(9):673–678, 2005.

[58] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, A. Riscos-Núñez, and F. J. Romero-Campero. On the power of dissolution in P systems with active membranes. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers*, volume 3850 of *Lecture Notes in Computer Science*, pages 224–240. Springer, 2005.

[59] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, A. Riscos-Núñez, F. J. Romero-Campero, and Á. Romero-Jiménez. Characterizing tractability

by cell-like membrane systems. *Formal models, languages and applications*, 66:137–154, 2006.

[60] R. Gutiérrez–Escudero, M. Pérez–Jiménez, and M. Rius–Font. Characterizing tractability by tissue-like p systems. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 289–300. Springer Berlin Heidelberg, 2010.

[61] P. G. Haydon. Glia: listening and talking to the synapse. *Nature Reviews Neuroscience*, 2(3):185–193, 2001.

[62] T. Head. Splicing systems, aqueous computing, and beyond. In I. Antoniou, C. Calude, and M. J. Dinneen, editors, *Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, Brussel, Belgium, 13-16 December 2000*, pages 68–84. Springer, 2000.

[63] O. Ibarra, A. Leporati, A. Păun, and S. Woodworth. *The Oxford Handbook of Membrane Computing*, chapter Spiking Neural P Systems. Volume 1 of Păun et al. [157], 2010.

[64] O. H. Ibarra, A. Păun, G. Păun, A. Rodríguez-Patón, P. Sosík, and S. Woodworth. Normal forms for spiking neural p systems. *Theoretical Computer Science*, 372(2–3):196 – 217, 2007. Membrane Computing.

[65] O. H. Ibarra and S. Woodworth. Spiking neural p systems: Some characterizations. In E. Csuhaj-Varjú and Z. Ésik, editors, *Fundamentals of Computation Theory*, volume 4639 of *Lecture Notes in Computer Science*, pages 23–37. Springer Berlin Heidelberg, 2007.

[66] O. H. Ibarra, S. Woodworth, F. Yu, and A. Păun. On spiking neural p systems and partially blind counter machines. In C. S. Calude, M. J. Dinneen, G. Păun, G. Rozenberg, and S. Stepney, editors, *Unconventional Computation*, volume 4135 of *Lecture Notes in Computer Science*, pages 113–129. Springer Berlin Heidelberg, 2006.

[67] M. Ionescu, G. Păun, and T. Yokomori. Spiking Neural P systems. *Fundam. Inf.*, 71(2,3):279–308, Feb. 2006.

[68] F. Ipate, R. Lefticaru, L. Mierla, L. , Valencia-Cabrera, H. Han, G. Zhang, C. Dragomir, M. J. Pérez-Jiménez, and M. Gheorghe. Kernel P systems: Applications and implementations. In Z. Yin, L. Pan,

and X. Fang, editors, *Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013*, volume 212 of *Advances in Intelligent Systems and Computing*, pages 1081–1089. Springer Berlin Heidelberg, 2013.

[69] M. Ito, C. Martín-Vide, and G. Păun. A Characterization of Parikh Sets of ET0L Languages in Terms of P systems. In M. Ito, G. Păun, and S. Yu, editors, *Words, Semigroups, and Transductions*, pages 239–253. World Scientific, 2001.

[70] P.-J. M. J., Álvaro Romero, and F. S. Caparrini. A polynomial complexity class in p systems using membrane division. *J. Autom. Lang. Comb.*, 11(4):423–434, Jan 2006.

[71] M. J. P. Jiménez, A. R. Jiménez, and F. S. Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, 2(3):265–285, 2003.

[72] R. A. Juayong, F. Cabarle, H. Adorna, and M. A. M. del Amor. On the simulations of evolution-communication P systems with energy without antiport rules for GPUs. In *Tenth Brainstorming Week on Membrane Computing*, volume I, pages 267–290, Seville, Spain, 2012. Fenix Editora.

[73] M. S. C. Keeler, B. Goodale, and J. M. B. C. Reed. Modelling the impacts of two exotic invasive species on a native butterfly: top-down vs. bottom-up effects. *Journal of Animal Ecology*, 75(3):777–788, 2006.

[74] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[75] D. Klaua. Über einen ansatz zur mehrwertigen mengenlehre. *Monatsberichte der Deutschen Akademie der Wissenschaften Berlin*, 7:859–867, 1965.

[76] D. Klaua. Über einen zweiten ansatz zur mehrwertigen mengenlehre. *Monatsberichte der Deutschen Akademie der Wissenschaften Berlin*, 8:161–177, 1966.

[77] S. Krishna and R. Ram. A variant of p systems with active membranes: Solving np–complete problems. *Romanian Journal of Information Science and Technology*, 2(4):357–367, 1999.

[78] R. E. Ladner. On the structure of polynomial time reducibility. *J. ACM*, 22(1):155–171, Jan. 1975.

[79] A. Leporati, G. Mauri, C. Zandron, G. Păun, and M. J. Pérez-Jiménez. Uniform solutions to SAT and subset sum by spiking neural P systems. *Natural Computing*, 8(4):681–702, 2009.

[80] A. Leporati, C. Zandron, C. Ferretti, and G. Mauri. Solving numerical np-complete problems with spiking neural p systems. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 336–352. Springer Berlin Heidelberg, 2007.

[81] A. Leporati, C. Zandron, C. Ferretti, and G. Mauri. On the computational power of spiking neural P systems. *IJUC*, 5(5):459–473, 2009.

[82] C. Ltd. *Handel-C Language Reference Manual.* Celoxica Ltd, 2005.

[83] L. F. Macías-Ramos, M. A. M. del Amor, M. J. Pérez-Jiménez, A. Riscos-Núñez, and L. Valencia-Cabrera. The role of the direction in tissue p systems with cell separation. *Journal of Automata, Languages and Combinatorics*, 19 (2014) 1–4:185–199, 2014.

[84] L. F. Macías-Ramos, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia-Cabrera, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua based simulator for Spiking Neural P systems. *Lecture Notes in Computer Science*, 7184:257–281, 2012.

[85] L. F. Macías-Ramos and M. J. Pérez-Jiménez. On recent developments in P-lingua based simulators for Spiking Neural P systems. *Asian Conference on Membrane Computing*, pages 14–29, 10/2012 2012.

[86] L. F. Macías-Ramos and M. J. Pérez-Jiménez. Spiking Neural P systems with functional astrocytes. In *Membrane Computing*, pages 228–242. Springer, 2013.

[87] L. F. Macías-Ramos, M. J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, and L. Valencia-Cabrera. The efficiency of tissue P systems with cell separation relies on the environment. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, and G. Vaszil, editors, *Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*, volume 7762 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2012.

[88] L. F. Macías-Ramos, M. J. Pérez-Jiménez, T. Song, and L. Pan. Extending Simulation of Asynchronous Spiking Neural P Systems in P-Lingua. *Fundam. Inf.*, 136(3):253–267, July 2015.

[89] L. F. Macías-Ramos, B. Song, L. Valencia-Cabrera, L. Pan, and M. J. Pérez-Jiménez. Membrane fission: A computational complexity perspective. *Complexity*, 4 2015.

[90] L. F. Macías-Ramos, L. Valencia-Cabrera, B. Song, L. Pan, and M. J. Pérez-Jiménez. A p–lingua based simulator for p systems with symport/antiport rules. *Fundamenta Informaticae*, 139:211–277, 04/2015 2015.

[91] M. Malița. Membrane Computing in Prolog. In C. S. Calude, M. J. Dinneen, and G. Păun, editors, *Pre-proceedings of the Workshop on Multiset Processing*, volume 140 of *CDMTCS Researh Reports*, pages 159–175, 7 2000.

[92] C. Martín-Vide, J. Pazos, G. Păun, and A. Rodríguez-Patón. A new class of symbolic abstract neural nets: Tissue p systems. In O. Ibarra and L. Zhang, editors, *Computing and Combinatorics*, volume 2387 of *Lecture Notes in Computer Science*, pages 290–299. Springer Berlin Heidelberg, 2002.

[93] C. Martín-Vide, G. Păun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theor. Comput. Sci.*, 296(2):295–326, Mar. 2003.

[94] M. Á. Martínez-del Amor. *Accelerating Membrane Systems Simulators using High Performance Computing with GPU.* PhD thesis, University of Seville, 5 2013.

[95] M. A. Martínez-del Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núòez, and M. J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. *Fundam. Inf.*, 136(3):269–284, July 2015.

[96] M. A. Martínez-del Amor, J. Pérez-Carrasco, and M. J. Pérez-Jiménez. Characterizing the parallel simulation of P systems on the GPU. *International Journal of Unconventional Computing*, 9(5-6):405–424, 2013.

[97] M. Á. Martínez-del Amor, I. Pérez-Hurtado, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, Á. Romero-Jiménez, C. Graciani,

A. Riscos-Núñez, M. A. Colomer, and M. J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P Systems with Proportional Object Distribution. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, and G. Vaszil, editors, *Membrane Computing*, volume 7762 of *Lecture Notes in Computer Science*, pages 257–276. Springer Berlin Heidelberg, 2013.

[98] M. A. Martínez-del Amor, I. Pérez-Hurtado, M. García-Quismondo, L. F. Macias-Ramos, L. Valencia-Cabrera, A. Romero-Jiménez, C. Graciani, A. Riscos-Núñez, M. A. Colomer, and M. J. Pérez-Jiménez. DCBA: Simulating Population Dynamics P systems with proportional object distribution. In E. Csuhaj-Varje, M. Gheorghe, G. Rozenberg, A. Salomaa, and G. Vaszil, editors, *Membrane Computing*, volume 7762 of *Lecture Notes in Computer Science*, pages 257–276. Springer Berlin Heidelberg, 2013.

[99] M. A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A. C. Elster, and M. J. Pérez-Jiménez. Population Dynamics P Systems on CUDA. In D. Gilbert and M. Heiner, editors, *Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 247–266. Springer Berlin Heidelberg, 2012.

[100] M. A. Martınez-del Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, and A. Riscos-Núñez. A P-Lingua based simulator for tissue P systems. *The Journal of Logic and Algebraic Programming*, 79(6):374 – 382, 2010. Membrane computing and programming.

[101] V. P. Metta, K. Krithivasan, and D. Garg. Simulation of Spiking Neural P Systems Using Pnet Lab. In M. Gheorghe, G. Paun, and S. Verlan, editors, *Proceedings of the 12th International Conference on Membrane Computing*, pages 381–394, 2011.

[102] T. Min, H. Peng, and S. Peng. Application of Adaptive Fuzzy Spiking Neural P Systems in Fault Diagnosis of Power Systems. *Chinese J. Electron.*, 23(1):87–92, 2014.

[103] R. Molla-Vayá and R. Vive-Hernando. Fixed-point digital differential analyser with antialiasing (FDDAA). *Computers and Graphics*, 26(2):329 – 339, 2002.

[104] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley, 1st edition, 2011.

[105] N. Murphy and D. Woods. Active membrane systems without charges and using only symmetric elementary division characterise p. In G. Eleftherakis, P. Kefalas, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 367–384. Springer Berlin Heidelberg, 2007.

[106] M. Mutyam and K. Krithivasan. P systems with membrane creation: Universality and efficiency. In *Proceedings of the Third International Conference on Machines, Computations, and Universality*, MCU '01, pages 276–287, London, UK, UK, 2001. Springer-Verlag.

[107] V. Nguyen, D. Kearney, and G. Gioiosa. Balancing performance, flexibility, and scalability in a parallel computing platform for membrane computing applications. In *Proceedings of the 8th Workshop on Membrane computing*, WMC'07, pages 385–413, Berlin, Heidelberg, 2007. Springer-Verlag.

[108] V. Nguyen, D. Kearney, and G. Gioiosa. An implementation of Membrane Computing using reconfigurable hardware. *Computing and informatics*, 27(3):551–569, 2008.

[109] V. Nguyen, D. Kearney, and G. Gioiosa. An algorithm for non-deterministic object distribution in P systems and its implementation in hardware. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 325–354. Springer Berlin Heidelberg, 2009.

[110] V. Nguyen, D. Kearney, and G. Gioiosa. An extensible, maintainable and elegant approach to hardware source code generation in reconfig-p. *Journal of Logic and Algebraic Programming*, 79(6):383–396, 2010.

[111] V. Nguyen, D. Kearney, and G. Gioiosa. A region-oriented hardware implementation for Membrane Computing applications. In G. Păun, M. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 385–409. Springer Berlin Heidelberg, 2010.

[112] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[113] T. Y. Nishida. Membrane algorithms. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3850 of

Lecture Notes in Computer Science, pages 55–66. Springer Berlin Heidelberg, 2006.

[114] A. Obtulowicz. Deterministic p systems for solving SAT problem. *Romanian Journal of Information Science and Technology*, 4(1-2):551–558, 2001.

[115] C. B. Octavian Arsene and N. Popescu. SNUPS - a simulator for numerical membrane computing. *International Journal of Innovative Computing, Information and Control*, 7(6):3509–3522, 2011.

[116] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[117] P. S. Pacheco. *Parallel programming with MPI.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[118] L. Pan and T.-O. Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, 10(5):630–649, may 2004.

[119] L. Pan, G. Păun, and M. J. Pérez-Jiménez. Spiking Neural P systems with neuron division and budding. *Science China Information Sciences*, 54(8):1596–1607, 2011.

[120] L. Pan, M. Pérez-Jiménez, and M. Rius-Font. New frontiers of the efficiency in tissue p systems. In *Pre-proceedings of Asian Conference on Membrane Computing (ACMC 2012), Huazhong University of Science and Technology, Wuhan, China, October 15-18, 2012*, pages 61–73, 2012.

[121] L. Pan and M. J. Pérez-Jiménez. Computational complexity of tissue-like P systems. *Journal of Complexity*, 26(3):296 – 315, 2010.

[122] L. Pan and G. Păun. Spiking Neural P systems with anti-spikes. *International Journal of Computers, Communications and Control*, IV:273–282, 09/2009 2009.

[123] L. Pan, J. Wang, and H. J. Hoogeboom. Asynchronous Extended Spiking Neural P systems with Astrocytes. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Int. Conf. on Membrane Com-*

puting, volume 7184 of *Lecture Notes in Computer Science*, pages 243–256. Springer, 2011.

[124] L. Pan, J. Wang, and H. J. Hoogeboom. Limited asynchronous spiking neural P systems. *Fundam. Inform.*, 110(1-4):271–293, 2011.

[125] A. Paun. On P systems with active membranes. In *Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, Brussel, Belgium, 13-16 December 2000*, pages 187–201, 2000.

[126] A. Paun. On P systems with active membranes. In *Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, Brussel, Belgium, 13-16 December 2000*, pages 187–201, 2000.

[127] G. Paun. Further twenty six open problems in membrane computing. *Third Brainstorming Week on Membrane Computing*, pages 249–262, 1/31/05/-2/4/05 2005.

[128] G. Păun, M. J. Pérez-Jiménez, and G. Rozenberg. Spike trains in spiking neural p systems. *International Journal of Foundations of Computer Science*, 17(04):975–1002, 2006.

[129] A. Pavel, O. Arsene, and C. Buiu. Enzymatic numerical P systems - a new class of membrane computing systems. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, pages 1331–1336, 2010.

[130] H. Peng, J. Wang, M. J. Pérez-Jiménez, H. Wang, J. Shao, and T. Wang. Fuzzy Reasoning Spiking Neural P System for fault diagnosis. *Information Sciences*, 235(0):106 – 116, 2013.

[131] I. Pérez-Hurtado. *Desarrollo y aplicaciones de un entorno de programación para Computación Celular: P-Lingua.* PhD thesis, University of Seville, 2010.

[132] I. Pérez-Hurtado, L. Valencia-Cabrera, J. M. Chacón, A. Riscos-Núñez, and M. J. Pérez-Jiménez. A P-Lingua based Simulator for Tissue P Systems with Cell Separation. *Romanian Journal of Information Science and Technology*, 17:89–102, 2014.

[133] I. Pérez-Hurtado, L. Valencia-Cabrera, M. J. Pérez-Jiménez, M. A. Colomer, and A. Riscos-Núñez. Mecosim: A general purpose software tool for simulating biological phenomena by means of p systems. In *Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on*, pages 637–643. IEEE, 2010.

[134] M. Pérez-Jiménez and F. Romero-Campero. A CLIPS simulator for recognizer P systems with active membranes. In *2nd Brainstorming Week on Membrane Computing*, pages 387–413, Seville, Spain, February 2004. Fénix Editora.

[135] M. Pérez-Jiménez and P. Sosík. Improving the efficiency of tissue P systems with cell separation. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume 2, pages 105–140, Seville, Spain, February 2012. Fénix Editora.

[136] M. J. Pérez-Jiménez and A. Riscos-Núñez. A linear-time solution to the knapsack problem using p systems with active membranes. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004.

[137] M. J. Pérez-Jiménez and A. Riscos-Núñez. Solving the subset-sum problem by P systems with active membranes. *New Generation Comput.*, 23(4):339–356, 2005.

[138] M. J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, and F. J. Romero-Campero. A polynomial alternative to unbounded environment for tissue P systems with cell division. *Int. J. Comput. Math.*, 90(4):760–775, 2013.

[139] M. J. Pérez-Jiménez and F. J. Romero-Campero. Attacking the common algorithmic problem by recognizer P systems. In M. Margenstern, editor, *Machines, Computations, and Universality, 4th International Conference, MCU 2004, Saint Petersburg, Russia, September 21-24, 2004, Revised Selected Papers*, volume 3354 of *Lecture Notes in Computer Science*, pages 304–315. Springer, 2004.

[140] M. J. Pérez-Jiménez and F. J. Romero-Campero. An efficient family of P systems for packing items into bins. *J. UCS*, 10(5):650–670, 2004.

[141] M. J. Pérez-Jiménez and F. J. Romero-Campero. A study of the robustness of the EGFR signalling cascade using continuous membrane

systems. In *Mechanisms, Symbols, and Models Underlying Cognition: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005, Las Palmas, Canary Islands, Spain, June 15-18, 2005, Proceedings, Part I*, pages 268–278, 2005.

[142] B. Petreska and C. Teuscher. A reconfigurable hardware membrane system. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 269–285. Springer Berlin Heidelberg, 2004.

[143] A. Porreca, N. Murphy, and M. Pérez-Jiménez. An efficient solution of Ham Cycle problem in tissue P systems with cell division and communication rules with length at most 2. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, pages 141–166, Seville, Spain, February 2012. Fénix Editora.

[144] A. E. Porreca. *Computational Complexity Classes for Membrane Systems*. PhD thesis, Universita' di Milano-Bicocca, 2008.

[145] A. Păun and G. Păun. The power of communication: P systems with symport/antiport. *New Gen. Comput.*, 20(3):295–305, July 2002.

[146] A. Păun, G. Păun, and G. Rozenberg. Computing by communication in networks of membranes. *International Journal of Foundations of Computer Science*, 13:779–798, 2002.

[147] A. Păun and M. Sidoroff. Sequentiality induced by spike number in snp systems: Small universal machines. In M. Gheorghe, G. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 333–345. Springer Berlin Heidelberg, 2012.

[148] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.

[149] G. Păun. P systems with active membranes: Attacking np complete problems. *JOURNAL OF AUTOMATA, LANGUAGES AND COMBINATORICS*, 6:75–90, 1999.

[150] G. Păun. Computing with membranes: Attacking np-complete problems. In I. Antoniou, C. Calude, and M. Dinneen, editors, *Unconventional Models of Computation, UMC'2K*, Discrete Mathematics and Theoretical Computer Science, pages 94–115. Springer London, 2001.

[151] G. Păun. *Membrane Computing: An Introduction.* Natural Computing Series. Springer–Verlag, Berlin, 2002.

[152] G. Păun. Spiking neural P Systems with astrocyte-like control. *J. UCS*, 13(11):1707–1721, 2007.

[153] G. Păun, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Tissue P systems with cell division. *International Journal of Computers Communications and Control*, 3(3):295–303, 2008.

[154] G. Păun, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Tissue P systems with Cell Division. In G. Păun, A. Riscos-Núñez, Á. Romero-Jiménez, and F. Sancho-Caparrini, editors, *Second Brainstorming Week on Membrane Computing*, 2004.

[155] G. Păun and R. Păun. Membrane Computing and economics: Numerical P systems. *Fundam. Inf.*, 73(1,2):213–227, 2006.

[156] G. Păun and G. Rozenberg. A guide to membrane computing. *Theoretical Computer Science*, 287(1):73–100, 2002. Natural Computing.

[157] G. Păun, G. Rozenberg, and A. Salomaa, editors. *The Oxford Handbook of Membrane Computing.* Oxford University Press, Oxford (U.K.), 2010.

[158] K. Qureshi and P. Manuel. Adaptive pre-task assignment scheduling strategy for heterogeneous distributed raytracing system. *Computers and Electrical Engineering*, 33(1):70 – 78, 2007.

[159] D. Ramírez-Martínez and M. A. Gutiérrez-Naranjo. A software Tool for Dealing with Spiking Neural P Systems. *Fifth Brainstorming Week on Membrane Computing*, pages 299–313, 1/29/07-2/2/07 2007.

[160] F. J. Romero-Campero and M. J. Pérez-Jiménez. A model of the quorum sensing system in vibrio fischeri using p systems. *Artif. Life*, 14(1):95–109, 1 2008.

[161] F. J. Romero-Campero and M. J. Pérez-Jiménez. Modelling gene expression control using P systems: The Lac Operon, a case study. *Biosystems*, 91(3):438 – 457, 2008.

[162] E. Sanchez. *Bio-Inspired Computing Machines: Toward Novel Computational Machines*, chapter An Introduction to Digital Systems, pages 13–48. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[163] S. Sedwards and T. Mazza. Cyto-Sim: a formal language model and stochastic simulator of membrane-enclosed biochemical processes. *Bioinformatics Applications Note*, 23(20):2800–2802, 2007.

[164] G. P. Silva and J. S. Aude. Evaluation of a sparc architecture with Harvard bus and branch target cache. *Microprocessing and Microprogramming*, 34(1-5):157 – 160, 1992.

[165] T. Song, L. Pan, and G. Păun. Asynchronous spiking neural P systems with local synchronization. *Information Sciences*, 219:197–207, 01/2013 2013.

[166] P. Sosík and A. Rodríguez-Patón. Membrane computing and complexity theory: A characterization of {PSPACE}. *Journal of Computer and System Sciences*, 73(1):137 – 152, 2007.

[167] P. Sosík and A. Rodríguez-Patón. Membrane computing and complexity theory: A characterization of PSPACE. *J. Comput. Syst. Sci.*, 73(1):137–152, 2007.

[168] Y. Suzuki, Y. Fujiwara, J. Takabayashi, and H. Tanaka. Artificial Life Applications of a Class of P Systems: Abstract Rewriting Systems on Multisets. In C. S. Calude, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 299–346. Springer Berlin Heidelberg, 2001.

[169] Y. Suzuki and H. Tanaka. On a LISP implementation of a class of P systems. *Romanian Journal of Information Science and Technology*, 3(2):173–186, 2000.

[170] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. Checl: Transparent checkpointing and process migration of opencl applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 864–876, 2011.

[171] S. M. Trimberger. *Field-Programmable Gate Array Technology.* Springer-Verlag New York, Inc., Boston, MA, USA, 1994.

[172] S.-Z. Ueng, M. Lathara, S. Baghsorkhi, and W.-m. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In J. Amaral, editor, *Languages and Compilers for Parallel Computing*, volume 5335 of *Lecture*

Notes in Computer Science, chapter CUDA-Lite: Reducing GPU Programming Complexity, pages 1–15. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[173] L. Valencia-Cabrera. *An environment for virtual experimentation with computational models based on P systems.* PhD thesis, University of Seville, 2 2015.

[174] J. Villasenor and B. Hutchings. The flexibility of configurable computing. *Signal Processing Magazine, IEEE*, 15(5):67–84, 1998.

[175] J. Wang and H. Peng. An extended spiking neural p systems for fuzzy knowledge representation. *Int. J. Innov. Comput. Inf. Control*, 7(7A):3709–3724, 2011.

[176] J. Wang and H. Peng. Adaptive fuzzy spiking neural P systems for fuzzy inference and learning. *Int. J. Comput. Math.*, 90(4):857–868, 2013.

[177] J. Wang, P. Shi, H. Peng, M. J. Pérez-Jiménez, and T. Wang. Weighted fuzzy spiking neural P systems. *IEEE T. Fuzzy Systems*, 21(2):209–220, 2013.

[178] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739 – 750, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[179] T. Wang, J. Wang, H. Peng, and Y. Deng. Knowledge representation using fuzzy spiking neural P system. In *Fifth International Conference on Bio-Inspired Computing: Theories and Applications, BIC-TA 2010, University of Hunan, Liverpool Hope University, Liverpool, United Kingdom / Changsha, China, September 8-10 and September 23-26, 2010*, pages 586–590. IEEE, 2010.

[180] T. Wang, J. Wang, H. Peng, and H. Wang. Knowledge representation and reasoning based on frsn p system. In *Intelligent Control and Automation (WCICA), 2011 9th World Congress on*, pages 849–854, June 2011.

[181] T. Wang, G. Zhang, and M. J. Pérez-Jiménez. Fault Diagnosis Models for Electric Locomotive Systems Based on Fuzzy Reasoning Spiking

Neural P Systems. In *Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers*, pages 385–395, 2014.

[182] T. Wang, G. Zhang, and M. J. Pérez-Jiménez. Fuzzy Membrane Computing: Theory and Applications. *International Journal of Computers, Communications and Control, Special Issue on Fuzzy Sets and Applications (Celebration of the 50th Anniversary of Fuzzy Sets)*, 10(6):903–934, 2015.

[183] T. Wang, G. Zhang, M. J. Pérez-Jiménez, and J. Cheng. Weighted fuzzy reasoning spiking neural p systems: Application to fault diagnosis in traction power supply systems of high-speed railways. *Journal of Computational and Theoretical Nanoscience*, 12(7):1103–1114, 2015.

[184] T. Wang, G. Zhang, H. Rong, and M. J. Pérez-Jiménez. Application of fuzzy reasoning spiking neural p systems to fault diagnosis. *International Journal of Computers Communications & Control*, 9(6):786–799, 2014.

[185] T. Wang, G. Zhang, J. Zhao, Z. He, J. Wang, and M. Perez-Jimenez. Fault diagnosis of electric power systems based on fuzzy reasoning spiking neural p systems. *Power Systems, IEEE Transactions on*, 30(3):1182–1194, May 2015.

[186] Web-page. AMD Home Page. `http://www.amd.com/unleash/`.

[187] Web-page. Java's regular expressions specification.

`http://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html`.

[188] Web-page. Natural Computing Group of the Technical University of Madrid.

`http://www.lpsi.eui.upm.es/nncg`.

[189] Web-page. OpenCL standard webpage. `http://www.khronos.org/opencl`.

[190] Web-page. Pnet Lab: A Petri net tool.

`http://www.automatica.unisa.it/PnetLab.html`.

[191] Web-page. Research Group on Natural Computing – University of Seville.
`http://www.gcn.us.es`.

[192] Web-page. The CUDA Website.
`https://developer.nvidia.com/cuda-zone`.

[193] Web-page. The Daikon invariant detector. `http://plse.cs.washington.edu/daikon/`.

[194] Web-page. The GNU GPL Website.
`http://www.gnu.org/copyleft/gpl.html`.

[195] Web-page. The Java Website.
`https://www.java.com/`.

[196] Web-page. The JCUDA Website.
`http://www.jcuda.org/`.

[197] Web-page. The MeCoSim Website.
`http://www.p-lingua.org/mecosim/`.

[198] Web-page. The NVIDIA Website.
`http://www.nvidia.com/content/global/global.php`.

[199] Web-page. The P-Lingua Website.
`http://www.p-lingua.org/`.

[200] Web-page. The P Systems Webpage.
`http://ppage.psystems.eu/`.

[201] Web-page. The Spin model checker. `http://spinroot.com/spin/whatispin.html`.

[202] Web-resource. NVIDIA CUDA C Programming Guide 4.2. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`.

[203] Web-resource. The OpenMP API specification for parallel programming. `http://www.openmp.org`.

[204] F. Wen and Z. Han. Fault section estimation in power systems using a genetic algorithm. *Electric Power Systems Research*, 34(3):165 – 172, 1995.

[205] G. Xiong, D. Shi, and J. Chen. Implementing fuzzy reasoning spiking neural p system for fault diagnosis of power systems. In *Power and Energy Society General Meeting (PES), 2013 IEEE*, pages 1–8, July 2013.

[206] G. Xiong, D. Shi, L. Zhu, and X. HDuan. A new approach to fault diagnosis of power systems using fuzzy reasoning spiking neural p systems. *Mathematical Problems in Engineering*, 2013, 2013.

[207] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8(3):338–353, 1965.

[208] L. A. Zadeh. Knowledge representation in fuzzy logic. *IEEE Trans. Knowl. Data Eng.*, 1(1):89–100, 1989.

[209] C. Zandron, C. Ferretti, and G. Mauri. Solving np-complete problems using p systems with active membranes. In I. Antoniou, C. Calude, and M. Dinneen, editors, *Unconventional Models of Computation, UMC'2K*, Discrete Mathematics and Theoretical Computer Science, pages 289–301. Springer London, 2001.

[210] X. Zeng, H. Adorna, M. A. Martínez-del Amor, L. Pan, and M. Pérez-Jiménez. Matrix representation of spiking neural p systems. In M. Gheorghe, T. Hinze, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 6501 of *Lecture Notes in Computer Science*, pages 377–391. Springer Berlin Heidelberg, 2011.

[211] G. Zhang, H. Rong, F. Neri, and M. J. Pérez-Jiménez. An optimization spiking neural p system for approximately solving combinatorial optimization problems. *International Journal of Neural Systems*, 24:1–16, 2014.