



UNIVERSITY OF SEVILLA
Dpt. of Computer Science
and Artificial Intelligence

Modelling and simulation of real-life phenomena in Membrane Computing

A thesis submitted for the degree of
Doctor in Philosophy
to obtain the PhD degree
School of Computer Engineering
University of Sevilla

Manuel García-Quismondo Fernández

Approval of the Thesis Supervisors

Mario de Jesús Pérez Jiménez

Miguel Angel Martínez del Amor

November 12, 2013

*A mis padres y a mi abuela,
por ayudarme siempre y sacarme adelante.*

Agradecimientos

Esta tesis es el culmen de seis años de trabajo. Un trabajo que, aunque firmado por sólo una persona, en absoluto es individual, y sin cuya ayuda y apoyo nunca hubiera podido realizar. Es por eso que quiero dedicar esta página a expresar my gratitud a todos los que hicieron posible este trabajo.

Me gustaría comenzar por mis padres, Juanjo y Margari, por su ayuda desde mi nacimiento, así como a mi abuela Antonia, mi hermano Juan, mis tíos Yeye y Pedro y mis primos Pedro e Irene, por acogerme como “hijo prestado” durante incontables periodos de tiempo.

Así mismo, quiero expresar my agradecimiento a todos mis compañeros en el Departamento de Ciencias de la Computación de la Universidad de Sevilla. Especialmente, mi agradecimiento va hacia Mario de Jesús Pérez por su acogida bajo su tutela a a hora de completar mi tesis doctoral. Su generosidad y capacidad de sacrificio por el prójimo ha sido siempre una fuente de inspiración para mí. Seguidamente, me gustaría agradecer a Ignacio Pérez y a Daniel Díaz por haberme dado la oportunidad de incorporarme al mundo de la investigación. A Fran Romero, por tutelarme durante mi primera etapa como investigador “remunerado”, a Agustín Riscos y a Ana Ruiz, por su inestimable ayuda en temas docentes y burocráticos, a Miguel Angel Gutiérrez y a Fernando Sancho por sus anécdotas, a Carmen Graciani por su experiencia durante mi etapa como docente y a Alvaro Romero por su inestimable ayuda en temas de administración de sistemas en general. Asimismo, no podía faltar mi codirector de tesis Miguel Angel Martínez por enseñarme todo lo que sé sobre paralelismo, administración de servidores Linux, dispositivos GPU, lenguaje de programación CUDA y cocina con arroz bomba. Y, cómo no, mi más sincero agradecimiento a Luis Valencia y Luisfe Macías por compartir y animar tanísimas horas de despacho. Por último, nada más que extender mi agradecimiento a todo el Departamento de Ciencias de la Computación e Inteligencia Artificial.

También querría agradecer este trabajo a mis compañeros de carrera, en especial a José Manuel Pérez, Juan Sebastián Tinoco, David Fernández y Carlos Portero, por tantas horas de trabajos grupales compartidas. *I would like also to thank all people who fostered me during my research stays all over the world. Specifically, I would like to thank Martyn Amos, Angel Goñi, Javi Esquivel, Pete Harding, Matt Crossley and Andy Nisbet for taking me in and sharing with me their experience during my stay at the Novel Computation Group in Manchester Metropolitan University, in Manchester (U.K.). In addition, I*

would like to convey my most sincere acknowledge to my companions at the Image Processing and Intelligent Control Key Laboratory in Wuhan, Hubei (China), especially to Linqiang Pan, Tao Song, Xueming Liu, Yansen Su, Yuan Kong and Qing Wei, and to Gexiang Zhang for his homely reception in Chengdu, Sichuan (China). Last but not least, I would like to extend my acknowledgement to all who fostered me as another member in the Department of Biology at Tufts University in Medford, Massachusetts (U.S.A.), with special regards to Francie Chew, Michael Reed, Coco Gómez, Kelly Boisvert (who is finishing off her poster beside me while I write this acknowledgement), Anne Madden, Ed Rocha and Brian Holiday. Imi-as place asemenea sa da un mesaj de multumire la Ana Pavel pentru totul ajutor și sprijinu primit.

Finalmente, agradezco profundamente todo el apoyo económico ofrecido por la beca de Formación de Profesorado Universitario (FPU) del Ministerio de Educación, la beca de Formación de Personal Investigador (FPI) del IV Plan Propio de la Universidad de Sevilla, de la Junta de Andalucía, y por los proyectos de investigación nacionales I+D+i del Ministerio de Economía y Competitividad TIN2006-13425, TIN2009-13192 y TIN2012-37434; todos ellos cofinanciados por los fondos FEDER de la Unión Europea.

Contents

I	Preliminaries	7
1	Natural Computing	9
1.1	Paradigms in Natural Computing	10
1.2	Membrane Computing	13
1.3	Stochastic models and Probabilistic Models	26
2	Simulation of P Systems	31
2.1	Simulators in Membrane Computing	31
2.2	Standards in Membrane Computing	35
2.3	Parallel simulation of P systems	45
2.4	GPU Computing	49
2.5	Hardware specifications	56
II	Contributions	57
3	Enzymatic Numerical P systems	59
3.1	Numerical P Systems	60
3.2	Enzymatic Numerical P Systems	61
3.3	Simulation of Enzymatic Numerical P Systems	65
3.4	A GPU simulator for Enzymatic Numerical P systems	65
3.5	Performance analysis of the GPU simulator	74
4	Logic Network Dynamic P systems	85
4.1	Some antecedents of Gene Network models in Membrane Computing	85
4.2	Logic Networks	87
4.3	Population Dynamics P systems	96
4.4	A PDP-based model of Logic Networks	100

5	Probabilistic Guarded P Systems	107
5.1	Formal description of PGP systems	108
5.2	Simulation of PGP systems	114
5.3	Parallel simulation of PGP systems	119
5.4	Software environment	123
III	Results	131
6	Case studies	133
6.1	Modelling logic networks with LNPN systems: <i>Arabidopsis thaliana</i> , a case study	133
6.2	A PGP model on the ecosystem of <i>Pieris napi oleracea</i>	137
7	Conclusions	151
7.1	Summary by chapter	151
7.2	Thesis overview	152
7.3	Future work	156
	Appendices	160
A	Gene Network Data	163
B	PGP Model Data	167
	Bibliography	171

List of Figures

1.1	A cellular automaton state, depicting the concept of neighbourhood	11
1.2	A Petri net example	13
1.3	A classic Markov chain example	13
2.1	Output screen and code sample of SNPS	32
2.2	Snapshot of SimCM main screen	34
2.3	A diagram on P-Lingua software architecture	37
2.4	A MeCoSim-generated GUI application	43
2.5	Infobiotics (left) and MetaPlab (right) screenshots	45
2.6	Overview of Petreska and Teuscher's FPGA Membrane Computing simulator	47
2.7	General structure (left) and monitoring system (right) of a Membrane Computing simulator based on microcontroller technology	48
2.8	Input (left) and output (right) from Ciobanu and Guo's cluster simulator	49
2.9	The CUDA programming model	51
3.1	A numerical P system (left) and an enzymatic numerical P system (right)	63
3.2	An ENPS model for obstacle avoidance	64
3.3	Production function expression $(x_{1,2} + 7) + (x_{1,4} - 3)$	67
3.4	Workflow of the simulators	75
3.5	A sample of XML code to define ENPS systems	77
3.6	Dummy ENPS (left) and ENPS for function approximation (right)	80
3.7	Execution times and speed-up factors for dummy model	83
3.8	Execution times and speed-up factors for function approximation model	84
4.1	Behaviour of unary operations f_1^j	91
4.2	Behaviour of binary operations f_2^j	92
4.3	A graphical description of a PDP system	99
5.1	An example of PGP system. Flags are highlighted in red and probabilities equal to 1 are omitted.	110
5.2	Comparison of PGP systems and P systems with proteins	112

5.3	Thread distribution among blocks for the proposed GPU implementation of Algorithm 5.3.1. S , N_Γ , q , N_B and N_R denote the number of simulations and the alphabet size, degree, total number of blocks and total number of rules in the system, respectively. Block size was set to 256 because it offered the best performance results.	123
5.4	Main screen of MeCoGUI	129
5.5	Workflow for P-Lingua simulator (upper branch) and PGPC++ and PGPCUDA (lower branch) for MeCoGUI	130
6.1	Input Data on MeCoSim interface	135
6.2	Simulation Results from MeCoSim interface	136
6.3	Values predicted by the simulator. Solid and dashed lines represent average population levels and typical deviations (in individuals) among simulations, respectively. Red lines display values for phenotype Rr , green lines those of RR and blue lines those of rr	147
6.4	Simulation times (left) and acceleration factors (right) for PGPCUDA and PGPC++	149
A.1	Initial gene states in the <i>Arabidosis thaliana</i> gene network on the longday scenario taken as case study	164
A.2	Unary gene interactions present in the logic network associated to the behaviour of <i>Arabidosis thaliana</i> taken as case study	164
A.3	Binary gene interactions present in the logic network associated to the behaviour of <i>Arabidosis thaliana</i> taken as case study (1/2)	165
A.4	Binary gene interactions present in the logic network associated to the behaviour of <i>Arabidosis thaliana</i> taken as case study (2/2)	166
A.5	Final gene states in the <i>Arabidosis thaliana</i> gene network on the longday scenario taken as case study	166
B.1	Integer simulation parameters (left) and values for N_g and $Prop_{i,k}$ for the simulated ecosystem	168
B.2	Values for F_y and $Hat_{k,g}$ (left) and p_i , Efi , Hi and $Det_{k,g}$ (right) for the simulated ecosystem	168
B.3	Values for $P_{y,in,i}$ for the simulated ecosystem	169
B.4	Values for non-parametrized probabilities (left) and for O_y and M_y (right) for the simulated ecosystem	169

List of Tables

2.1	Hardware specifications of the laptop in which the simulations in this work have been carried out . . .	56
4.1	Parameters for LNDP systems	106
6.1	Parameters for <i>Pieris napi oleracea</i> model	146

Motivation

Natural Computing is a terminology introduced to encompass three classes of methods: (1) those that take inspiration from nature for the development of novel problem-solving techniques; (2) those that are based on the use of computers to synthesize natural phenomena; and (3) those that employ natural materials (e.g., molecules) to compute [60]. Paradigms inside this discipline are Artificial Neural Networks [121], Genetic Algorithms and Evolutionary Computing [230], Swarm Intelligence [235], Artificial Immune Systems [158] and DNA Computing [39], among others.

Membrane Computing is a bio-inspired branch of Natural Computing initiated by Gheorghe Păun which abstracts computing models from the structure and functioning of living cells and from the organization of cells in tissues or other higher order structures [190]. This is done by defining theoretical devices known as membrane systems or *P systems*. Although the foundational model (also known as *Transition P systems*) defines a membrane structure consisting on a hierarchical arrangement of membranes [188] in the form of a rooted tree, a wide range of modelling frameworks inside Membrane Computing has been introduced since its foundation. Among these models one can cite *Spiking Neural P systems* [163], *Tissue-like P systems* [162], *Numerical P systems* [193] and *Array-rewriting P systems* [40]. The syntactical ingredients of most P systems consist on an alphabet of symbols, called *objects*, a membrane structure composed of separate compartments or *membranes*, a (possibly empty) *multi-set* associated with each membrane and a set of rewriting rules per membrane which provide the evolution of the system throughout discrete time steps.

Time in P systems models is discrete and advances in finite steps [188]. In addition, a global clock marking the time for the system (i.e., for all compartments in the whole system) is assumed [194]. In this sense, the instantaneous description at any instant of a P system is known as *configuration*. Given a configuration C_1 , we say that we have a transition from C_1 to configuration C_2 by using the evolution rules of the system. A sequence of transitions between

configurations of a given P system is called a *computation* if it is maximal [188]. The majority of P system frameworks work in a *non-deterministic* and *maximally parallel* manner. The objects to evolve in a step and the rules by which they evolve are chosen in a non-deterministic manner, but in such a way that for each membrane we have a maximally parallel application of rules. This means that we assign objects to rules, non-deterministically choosing the rules and the objects assigned to each rule, but in such a way that after this assignation no further rule can be applied to the remaining objects [151].

Membrane Computing frameworks have been mainly employed as alternative approaches to tackle computationally hard problems, especially **NP**-complete ones. P systems are capable of solving **NP**-complete problems on polynomial time by trading time for space. That is, when Membrane Computing is used as an alternative to traditional approaches to solve problems from this complexity class, they make use of an exponential number of computing devices created in a natural manner in polynomial time. One can find examples of this approach in problems such as SAT [93], 3-Col [79] and Knapsack [174].

Another facet of P systems is their use as a modelling framework for real-life phenomena. When studying these processes, it is usually useful to build computing models. The degree of abstraction is a crucial aspect for the design of a model capable of capturing the dynamics of a process. That is, models must be kept as simple as possible, drawing away all unnecessary complexity and defining criteria to select which information is fed into it [86]. In this sense, P systems reveal themselves as formal abstractions of the phenomenon under study, filtering out aspects of lesser importance and including those which prove to be relevant throughout the iterative process of modelling, simulation and validation in a modular manner, i.e., in such a way that changing, deleting or including small pieces of knowledge in the system entails proportionally small changes in the model [199]. The modelled phenomena themselves are rather diverse, ranging from biochemistry [75, 199, 200] to image processing [226] and including robotics [221], economics [193], software [79], automatic music generation [72] and ecology [52].

Given a real-life process subjected to study and a model thereof, in order to verify that the behaviour of the model corresponds to that of the system it is usually necessary to *simulate* the system. Rather than simulating, one alternative approach would consist on *implementing* the models. However, when it comes to Membrane Computing currently it is not possible; P systems have not been implemented either *in vivo*, *in vitro* nor *in silico*. Traditionally, Membrane Computing simulators were completely *ad-hoc* applications for the

model at hand, simulator parameters were hard-coded and the code was not reusable because it was intended to work for a specific P system [202, 54]. In this sense, P-Lingua [71] pioneered the standardisation of P systems by implementing an open, plugin-based software architecture meant for its extension by third-party developers. P-Lingua is a software project which provides a specification language in which designers can define P systems, known as P-Lingua as well. This language can be easily extended when required. In addition, it also provides a set of Java [6] simulators, in such a way that users can select which simulator from those included suits better their needs. Moreover, *pLinguaCore* is a software application inside the P-Lingua framework which implements an extension mechanism in which new formats and simulators can be included in the framework.

Originally, Membrane Computing simulators were exclusively implemented on sequential architectures, such as standard personal computers, occasionally by using declarative languages such as Prolog [54]. However, this is an inherent limitation, since such devices do not match well with the parallel nature of P systems, so the quest for new technological approaches for simulation in Membrane Computing comes to the fore. In this sense, it is natural to resort to parallel architectures such as FPGA boards [223, 135, 151], computer clusters [46], microcontrollers [90, 89] and Graphic Processor Units (*GPUs*) [38, 139, 30, 113].

The aim of this thesis is the study and development of Membrane Computing modelling frameworks for real-life phenomena, as well as of simulators capable of capturing their semantics. Simulators included as results of this thesis are both sequential and parallel. Recent developments in Membrane Computing have enabled its application to model biochemical phenomena [199, 41, 23], and other areas such as ecosystems [52] or robotics [221] prove the versatility of this discipline as a tool to reproduce the behaviour of real-life systems. Some of these phenomena display a parallel structure *per se*. For instance, in collaborative robotics several robots with similar behaviour communicate with each other to perform common tasks [94], or integrate several components (sensors and actuators) to interact with the environment. Similarly, ecological models in Membrane Computing display a parallel structure because they are composed of a large number of elements (animals and plants) which interact with each other and with the environment. In this sense, parallel simulators prove their usefulness as software tools to simulate the dynamics of these models and, eventually, predict the future behaviour of the modelled systems.

This work starts with a GPU-based simulator for Enzymatic Numerical P

systems (ENPSs), a modelling framework in Membrane Computing designed to reproduce the behaviour of robotic systems such that they are composed of modular parts, each one with a specific function and able to interact with each other and with the environment to achieve objectives. This work continues introducing a Membrane Computing model for Gene Regulatory Networks known as *Logic Network Dynamic P* (LNDP) systems, which is a framework composed of modular elements (genes and operations over them) which communicate with each other. The consequence of this communication is the emergence of the dynamics of the network and its transition among states. Next, a modelling framework for ecosystems known as *Probabilistic Guarded P* (PGP) systems is proposed. This framework is complemented with two simulators, one sequential and the other GPU-based, to reproduce the dynamics of the framework. Finally, two case studies are provided. The first one is a case study on a Gene Regulatory Network associated with the behaviour of *Arabidopsis thaliana* in LNDP systems, whereas the second is a model on the ecosystem of butterfly *Pieris napi oleracea* in PGP systems, complemented with a performance analysis of its parallel simulator using this model as a benchmark.

Document structure

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

Part I: Preliminaries

Chapter I familiarizes the reader with the basics of Natural Computing, introducing some classical models in the discipline. Following, it delves into the state of the art of Membrane Computing. Some frameworks including the seminar *transition* model are discussed. Finally, it contrasts stochastic and probabilistic approaching when modelling real-life phenomena.

Chapter II discusses simulators in Membrane Computing and describes the project P-Lingua and the software tool *pLinguaCore*, which enable experts to describe and simulate P systems automatically. Moreover, some Membrane Computing simulators implemented on parallel platforms are described, with a special emphasis on those developed for Graphic Processing Units (*GPU*).

Part II: Contributions

Chapter III discusses Enzymatic Numerical P Systems (*ENPS*), a deterministic model for robotics, introducing its antecedents and sequential simulators. In addition, a parallel, GPU-based simulator is presented, including a performance analysis with some case studies and a methodology for repeated simulation of ENPSs.

Chapter IV discusses a model on Logic Networks (LNs), which are a specific type of Gene Regulatory Networks in which the combination of the states of a set of genes can influence another one. In addition, a model based on Population Dynamic P systems (PDP systems, for short) is finally presented, describing its semantics and its elements in detail.

Chapter V formalizes Probabilistic Guarded P Systems (*PGP Systems*, for short), a new modelling framework for ecology. The characteristic features of this approach are described, i.e., its spatial distribution and elements and its syntax and semantics. A parallel,

GPU-based simulator is described, as well as the integration of PGP systems into the P-Lingua framework.

Part III: Results

Chapter VI presents some case studies on the models and simulators described in part II, specifically, the modelling and simulation of a Logic Network involved in the flowering process of *Arabidopsis thaliana* by means of LNDP systems and the modelling and simulation of the ecosystem of White Cabbage Butterfly (*Pieris napi oleracea*).

Chapter VII focuses on the results compiled in this document, recapitulating the achievements and conclusions of this thesis and discussing some new lines of work resulting from it.

Part I
Preliminaries

Chapter 1

Natural Computing

In order to improve his quality of life, mankind has faced a variety of problems. The study of these problems paved the way for the study systematic procedures for their solution. The discovery of these procedural tasks was two-sided; on the one hand, it allowed knowledge transmission among peers, on the other hand, enabled mankind to build devices to carry out these tasks. Historical devices to automatically solve problems include the **abacus**, whose first appearance on historical registers dates back to 500 B.C. In 1801, Jacquard's introduced his **mechanical loom**, a semi-mechanical contraption which enabled automatic looming of patterns into fabric by means of a sequence of punch cards. Holes carved in these cards indicated the location of threads to achieve the desired pattern. In the decade of 1800s, Babbage's **differential engine** was a device with a hand-turned crank which produced successive terms in a mathematical series. His success pushed him forward towards the construction of the **analytical engine** (1847-1849), a sketch on a general-purpose computer, which was never built due to the fact that technology at that time did not allow its implementation. Finally, 1890 Hollerith devised an **automatic machine** to carry out censuses, though its was not significantly faster than performing the process by hand [211].

The first electronic computers (Z1, ABC and Eniac) revolved mathematics and computer science for good. The ensuing euphoria about the potential of computers to carry out calculations was curbed by Churchhouse's proof in 1936 about the physical limitations of processors based on electronic technology on computing speed. This breakthrough states that there exist an upper bound that calculations on electronic computers cannot overcome. Thus, independently of how much electronic processors are accelerated, there exist relevant instances of computationally hard problems which would take years (or even

centuries) to be solved. The only scenario in which these limitations could be surmountable is if the class of problems solvable on polynomial time by deterministic Turing machines is equal to class of problems solvable on polynomial time by non-deterministic ones. In the scientific community, there is a widely believed consensus about that this property (commonly known as the **P** \neq **NP** conjecture [53]) is not verified.

This chapter is structured as follows. Section 1.1 overviews some of the most widely used paradigms in Membrane Computing, including cellular automata [208], Petri nets [181, 182] and Markov chains [81]. Section 1.2 introduces the reader into the discipline of Membrane Computing. Finally, Section 1.3 compares two common approaches to handle randomness in the modelling of real-life phenomena, the stochastic approach and the probabilistic one.

1.1 Paradigms in Natural Computing

Since this question has been around for quite a lot of years, the quest for new computing paradigms capable of surpassing the limitations of electronic devices has come to the fore. In this context, Natural Computing emerges as a way forward. This discipline studies the simulation and implementation of dynamic processes taking place in nature and susceptible to be interpreted as calculations. Natural Computing is a terminology introduced to encompass three classes of methods: (1) those that take inspiration from nature for the development of novel problem-solving techniques; (2) those that are based on the use of computers to synthesize natural phenomena; and (3) those that employ natural materials (e.g., molecules) to compute [60]. Paradigms inside this discipline include Artificial Neural Networks [121], Genetic Algorithms and Evolutionary Computing [230], Swarm Intelligence [235], Artificial Immune Systems [158] and DNA Computing [39], among others.

Thus, Natural Computing defines an array of computational frameworks inspired on processes found in nature. Some of these frameworks, commonly applied as modelling tools in the field of ecology and population dynamics in general (which will be of special interest later in this document), are:

Cellular Automata: The introduction of cellular automata is attributed to Stanislaw Ulam and John von Neumann in the 1940s. A cellular automaton defines a system out of objects that have varying states over time. In short, a cellular automaton is a model of a system of "cell" objects with the following characteristics [208]:

- The cells live on a grid. This grid may have any finite number of dimensions.
- Each cell has a state, typically being the number of possible states finite. The simplest example defines possibilities 1 and 0, sometimes referred as *on* and *off* or *alive* and *dead*.
- Each cell has a neighbourhood, typically defined as a list of adjacent cells.

Although there exists a plethora of semantics for cellular automata, usually the state of a cell in time t is computed as a function over its neighbourhood, possibly including the cell itself. Figure 1.1 depicts an example of cellular automaton. Wolfram [233] remarks that cellular automata are not simply neat tricks, but are relevant to the study of biology, chemistry, physics, and all branches of science, thus revealing themselves as a consolidated modelling paradigm.

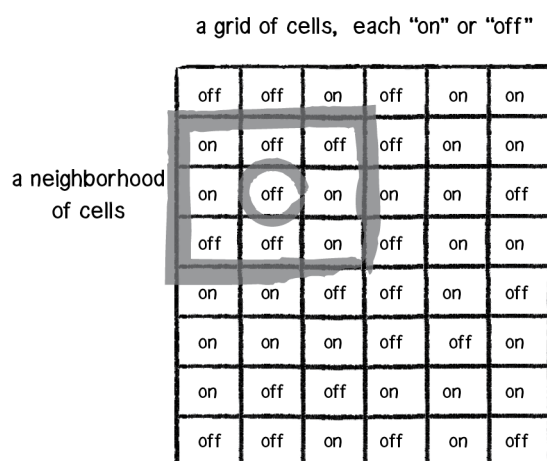


Figure 1.1: A cellular automaton state, depicting the concept of neighbourhood

Petri Nets: Petri nets are a promising graphical and mathematical tool for describing and studying information processing systems that are characterized as being concurrent, asynchronous, distributed, parallel, non-deterministic and/or stochastic [146]. Petri nets can be used as a visual-communication aid similar to flow charts and block and UML diagrams. Historically speaking, the concept of Petri net has its origins in Carl Adam Petri's dissertation in 1962 [181, 182]. A Petri net is composed of a set of **transitions**. In addition, there exists a set of **compartments** which contain a positive number of **tokens**. Transitions and

compartments are linked by means of weighted **arcs**. In this context, a compartment is said to be an **input** of a transition if there exists an arch for which the compartment is the source and the transition is the target. Likewise, a compartment is said to be an *output* of a transition if there exists an arch for which the compartment is the target and the transition is the source. Formally speaking, a Petri Net is a tuple $PN = (P, T, F, W, M_0)$, where:

- $P = \{p_1, p_2, \dots, p_m\}$ is a finite set of **compartments**
- $T = \{t_1, t_2, \dots, t_n\}$ is a finite set of **transitions**. P and T are disjoint sets (i.e. $P \cap T = \emptyset$).
- $F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relations)
- $W : F \rightarrow \mathbb{N}^+$ is a **weight function**
- $M_0 : F \rightarrow \mathbb{N}^+ \cup \{0\}$ is an **initial marking**

The state or **marking** of a Petri net is changed according to the following transition (*firing*) rule:

- A transition $t \in T$ is said to be *enabled* if each input compartment p of t is marked with at least $W(p, t)$ tokens, where $W(p, t)$ is the weight of the arc from p to t .
- An enabled transition might or might not fire.
- A firing of an enabled transition t removes $W(p, t)$ tokens from each input compartment p of t , and adds $W(t, p)$ tokens from each output compartment p of t .

Figure 1.2 depicts a Petri net. An exhaustive survey on Petri nets can be found in [146].

Markov Chains: Markov chains are not Natural Computing models, in the sense that they are not directly inspired by nature. However, they are included in this shortlist due to their salient relevance in ecology and population dynamics. In contrast to the aforementioned cases, Markov Chains [81] are probabilistic models in the sense that transitions between states are dictated by a set of probabilities. A Markov chains is composed of:

- A finite set of **states** $S = \{s_0, s_1, \dots, s_n\}$, where s_0 is known as the **initial state** of the system.

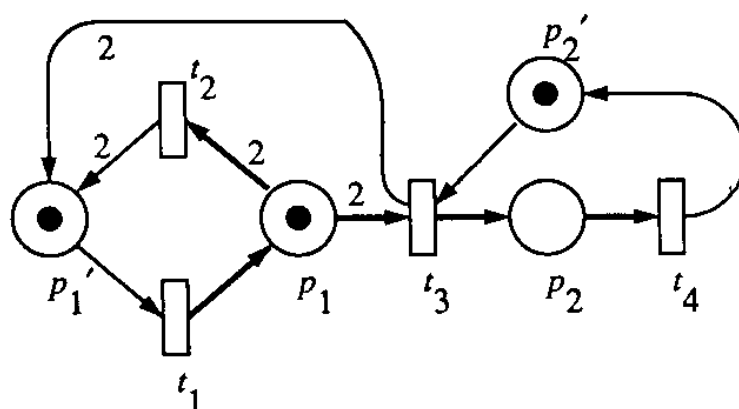


Figure 1.2: A Petri net example

- A $n \times n$ **transition matrix** usually noted as $P = P_{i,j}, 1 \leq i, j \leq n$, where $P_{i,j}$ determines the probability for a chain to move from state s_i to state s_j , and they are known as **transition probabilities**.

Figure 1.3 depicts a Markov chain. The apparent simplicity of Markov chains motivated its popularity as a modelling framework in different environments. In this sense, there exist Markov chain models on areas which range from ecology [240, 100, 82] to computer vision [234, 103, 225].

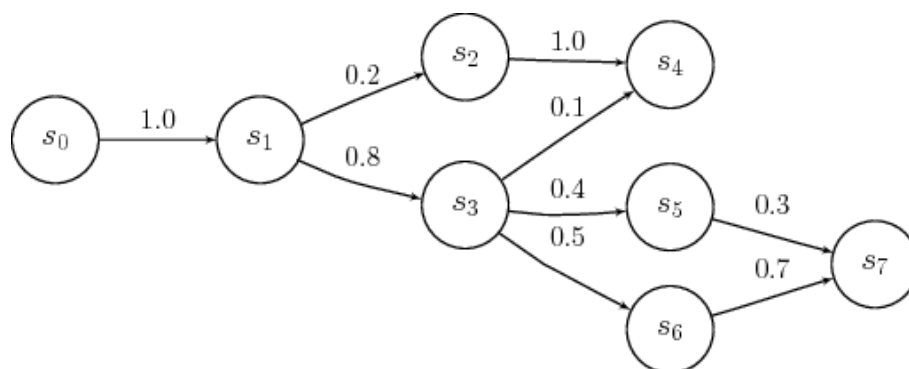


Figure 1.3: A classic Markov chain example

1.2 Membrane Computing

Membrane Computing is a quite active branch of Natural Computing, initiated by Gh. Păun at the end of 1998 [188]. Membrane Computing studies the

properties and applications of theoretical computing devices known as P systems, which are in some sense an abstraction of the structure and functioning of a living cell. Although there exists a large number of different definitions for P systems, most of them share some common features: a membrane structure composed of a number of regions or compartments, and an alphabet of symbols that allow to represent the objects placed in compartments and are able to evolve and/or travel through the membrane structure according to a set of rewriting rules, emulating the way substances undergo biochemical reactions in a cell. Many of the first P systems specifications that were investigated proved to be universal or computationally complete (i.e. equivalent in power to Turing Machines). Besides, the quest for efficiency has been another research direction, yielding in many cases polynomial-time cellular solutions to NP-complete problems, making a space-time trade-off and using the inherent massive parallelism of P systems. Another different approach is to concentrate on the evolution of the P systems itself, rather than focusing on the output of the computation and the number of steps [171]. In this context, P systems have been used to model biological phenomena within the framework of cellular systems and population biology presenting models of oscillatory systems [132, 68], signal transduction pathways [45], gene regulation control [41, 75, 101], quorum sensing [199], metapopulations [22], and real ecosystems [52, 33].

1.2.1 Preliminary concepts

Prior to delving any further into the concept of P systems, some preliminary concepts which will be used throughout this document are introduced [203]:

- An *alphabet* Γ is a non-empty set whose elements are called symbols.
- A *multiset* over an alphabet Γ is an application from Γ to the set N of natural numbers.
- $M(\Gamma)$ denotes the set of all the multisets over Γ .
- Given $u, u' \in M(\Gamma)$, $v = u \cap u'$ is defined as the multiset over Γ where $|v|_x = \min\{|u|_x, |u'|_x\}$, $\forall x \in \Gamma$.
- Given $u, v \in M(\Gamma)$, we say that v is contained in u and denote it by $v \subseteq u$, if and only if $|u|_x \geq |v|_x$, $\forall x \in \Gamma$.
- Given $u, u' \in M(\Gamma)$, $v = u + u'$ is defined as the multiset over A where $v(x) = |u|_x + |u'|_x$, $\forall x \in \Gamma$.

- Given $u, u' \in M(\Gamma)$, $u' \subseteq u$, $v = u - u'$ is defined as the multiset over Γ where $v(x) = |u|_x - |u'|_x$, $\forall x \in \Gamma$.
- Given $u \in M(\Gamma)$, $k \in \mathbb{N}$, $v = k \cdot u$ is defined as the multiset over Γ where $|v|_x = k \cdot |u|_x$, $\forall x \in \Gamma$.

A *membrane structure* is a rooted tree in which the nodes are called membranes, the root is called *skin*, and the leaves are called *elementary membranes*. The *degree* of a membrane structure is the number of membranes it contains (that is, the number of nodes of the tree).

Remark 1.1. *The concept of membrane structure is not the same in all Membrane Computing frameworks. In other cases, the membrane structure might be composed of a set of membranes disposed in a graph-like fashion.*

1.2.2 Transition P systems

Transition P systems are the foundational model originally introduced by Gh. Păun in 1998 [188]. Due to its simplicity, it has been widely chosen as a paradigm for its simulation.

Definition 1.1. *A transition P system of degree $m \geq 1$ without input is a tuple $\Pi = (\Gamma, \mu_\Pi, M_1, \dots, M_m, R_1, \rho_1, \dots, R_m, \rho_m)$ where:*

- Γ is the working alphabet of the system.
- μ_Π is a membrane structure of degree m . The membranes are labelled, in a one-to-one manner, from 1 to m .
- M_i is a multiset over Γ associated with membrane i , $1 \leq i \leq m$.
- R_i is a finite set of rewriting rules associated with membrane i , ($1 \leq i \leq m$). A rule is a pair (u, v) , usually written as $u \rightarrow v$, where u is a multiset over Γ and $v = v'$ or $v = v'\delta$, where v' is a string over $\Gamma \times (\{\text{here}, \text{out}\} \cup \{\text{in}_j | j \text{ is a membrane in } \mu_\Pi\})$.
- ρ_i is a strict partial order over the set of rules R_i .

An instantaneous description or *configuration* at any instant of a basic cell-like P system $\Pi = (\Gamma, \mu_\Pi, M_1, \dots, M_m, R_1, \dots, R_m)$, consists of a membrane structure and a family of multisets of objects over Π associated with each region of the structure. The *initial configuration* is (μ, M_1, \dots, M_m) . The rules

are chosen in a non-deterministic way, and in each region all objects that can evolve must do it. A configuration is a *halting configuration* if no rule of the system is applicable to it. In each time unit we can transform a given configuration in another configuration by applying the evolution rules to the objects placed inside the regions of the configurations, in a non-deterministic, and maximally parallel manner. In this way, we get *transitions* from one configuration of the system to the next one.

A computation, C , of a P system is a (finite or infinite) sequence of configurations, $\{C^i\}_{i < r}$, where:

- C^0 is an initial configuration of the system.
- $C^i \Rightarrow_{\Pi} C^{i+1}$, for every $i < r$
- Either $r \in \mathbb{N}, r \geq 1$ and C^{r-1} is a halting configuration (C is then a halting computation performing $r - 1$ steps), or $r = \infty$ (C is then a not halting configuration).

We say that Configuration C^1 yields configuration C^2 in one *transition step*, denoted by $C^1 \Rightarrow C^2$, if we can pass from C^1 to C^2 by applying the rules from R following the previous remarks.

The semantics of the model is explained in [203]. Prior to defining it, some concepts from [190] are introduced. A rule $r \in R_i, r = (u, v), 1 \leq i \leq m$ is *applicable* in configuration C^t if and only if u is contained in M_i at instant t . Likewise, A multiset of rules is *applicable* to the multiset of objects available in the respective region if and only if there are enough objects to apply the rules a number of times as indicated by their multiplicities. Finally, a multiset of rules is *maximal* if and only if no further rule can be added to it (no multiplicity of a rule can be increased) such that the obtained multiset is still applicable.

The application of an applicable rule $r = (u, v)$ in R_i at instant t is done as follows: first, the objects in u are removed from membrane i ; then, for every $(a, out) \in v$ an object a is put into the multiset associated with i 's first non-dissolved ancestor (or the environment if i is the skin membrane); for every $(a, here) \in v$ an object a is added to membrane i ; for every $(a, in_j) \in v$ an object ob is added to membrane j if and only if j is a child membrane of i ; otherwise, the rule cannot be applied. Finally, if $\delta \in v$, then membrane i is dissolved, that is, it is removed from the membrane structure. As a result, the objects associated with this membranes are collected by the first non-dissolved ancestor, and the rules are lost. An exception is the case of the skin membrane, which cannot be dissolved.

Let us consider a P system defined according to a Membrane Computing framework. The framework is said to be **maximally parallel** if, on every transition step of the P system, each chosen multiset of rules is maximal. Similarly, a configuration might have several following configurations because it is possible that there exists different maximal multisets of rules applicable to such a configuration, so P systems are non-deterministic devices.

The objects to evolve in a step and the rules by which they evolve are chosen in a non-deterministic manner, but in such a way that in each region we have a maximally parallel application of rules. This means that we assign objects to rules, non-deterministically choosing the rules and the objects consumed by each rule, but in such a way that after this consumption no further rule can be applied to the remaining objects.

1.2.3 Variants of P systems

As claimed above, Membrane Computing is not a monolithic discipline; rather, it encompasses a variety of computational paradigms, mostly devised to tackle computationally hard problems and/or model real-life phenomena. Apart from the transition model, some of these paradigms are listed below, which have been chosen either because they have been successfully simulated in parallel architectures or because of they have been used as inspiration for some of the models defined in the subsequent chapters. These models are *P systems with active membranes*, *Tissue-like P systems*, *Spiking Neural P systems* and *Kernel P systems*. Apart from the paradigms addressed in this chapter, others such as Enzymatic Numerical P systems (*ENPSs* [221]) and Population Dynamics P Systems (*PDP Systems*) [51]) will be later discussed in depth in Chapters 3 and 4, respectively.

1.2.3.1 P systems with active membranes

This model of P systems has been actively employed to solve computationally hard problems in polynomial time [173, 176]. Two features implemented by this model make it especially interesting for computational approaches trading time for space, as described by Păun in [189]. First, membranes can not only be dissolved, but they can also be duplicated by division. An elementary membrane can be divided by means of an interaction with an object from that membrane. The skin is never divided nor dissolved. Secondly, each membrane is supposed to have an “electrical polarization” or charge, one of the three possible: **positive** (+), **negative** (-), or **neutral** (0).

If in a non-elementary membrane there are two immediately lower membranes of opposite polarizations, one positive and one negative, then that membrane can also divide in such a way that the two membranes of opposite charge are separated; all membranes of neutral charge and all objects are duplicated and a copy of each of them is introduced in each of the two new membranes. Gh. Păun formalizes this model according to the following syntax:

Definition 1.2. A P system with active membranes of degree $m \geq 1$ is a construct: $\Pi = (\Gamma, H, \mu, M_1, \dots, M_m, \mathcal{R})$ where:

- Γ is an alphabet of objects
- H is a finite set of labels for membranes
- μ is a membrane structure, consisting of m membranes, labelled (not necessarily in a one-to-one manner) with elements of H ; all membranes in μ have neutral polarization at the initial configuration.
- M_1, \dots, M_m are multisets over Γ
- \mathcal{R} is a finite set of developmental rules of the following forms:
 - $[a \rightarrow v]_h^\alpha$, for $h \in H, \alpha \in \{0, +, -\}, a \in \Gamma, v \in M(\Gamma)$ (object evolution rules, associated with membranes and depending on the label and the charge of the membranes but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them).
 - $a[]_h^\alpha \rightarrow [b]_h^{\alpha'}$, for $h \in H, \alpha, \alpha' \in \{0, +, -\}, a, b \in \Gamma$ (in communication rules; an object is introduced in the membrane, and possibly modified during this process; also the polarization of the membrane can be modified, but not its label);
 - $[a]_h^\alpha \rightarrow []_h^{\alpha'} b$, for $h \in H, \alpha, \alpha' \in \{0, +, -\}, a, b \in \Gamma$ (out communication rules; an object is sent out of the membrane, and possibly modified during this process; also the polarization of the membrane can be modified, but not its label);
 - $[a]_h^\alpha \rightarrow b$, for $h \in H, \alpha \in \{0, +, -\}, a, b \in \Gamma$ (dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);

- $[a]_h^\alpha \rightarrow [b]_h^{\alpha'} [c]_h^{\alpha''}$, for $h \in H, \alpha, \alpha', \alpha'' \in \{0, +, -\}, a, b, c \in \Gamma$ (*division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label, and possibly of different polarizations; the object specified in the rule is replaced in the two new membranes possibly by new objects; the remaining objects are duplicated and may evolve in the same step by rules of the first type*).

Gh. Păun [189] also defines a semantic for P systems with active membranes according to the following principles:

- All rules are applied in parallel: in a step, evolution rules are applied to all objects to which they can be applied, all other rules are applied to all membranes to which they can be applied; an object can be used by only one rule, non-deterministically chosen (there is no priority relation among rules), but any object which can evolve by a rule of any form, should evolve. Moreover, on each transition and each membrane only one rule of type *dissolution*, *division*, *send-in* or *send-out* can be applied only once.
- If a membrane is dissolved, then all the objects in its region are left free in the first non-dissolved region immediately above it. Because all rules are associated with membranes, the rules of a dissolved membrane are no longer available at the next steps. The skin membrane is never dissolved.
- All objects and membranes not specified in a rule and which do not evolve are passed unchanged to the next step. For instance, if a membrane with the label h is divided by a division rule which involves an object a , then all other objects in membrane h which do not evolve are introduced in each of the two resulting membranes h . Similarly, the inner membrane structure is reproduced in each of the two new membranes with the label h , unchanged if no rule is applied to them. In particular, the contents of these neutral membranes is reproduced unchanged in these copies, providing that no rule is applied to their objects.
- If at the same time a membrane h is divided by a division rule and there are objects in this membrane which evolve by means of evolution rules, then in the new copies of the membrane the result of the evolution is introduced; that is to say, first the evolution rules are used, changing the objects, and then the division is produced, so that copies of the evolved objects are introduced in the two new membranes with label h .

Of course, this process takes only one step. This principle defines some sort of synchronization between rules applied at the same transition step.

- The rules associated with a membrane h are used for all copies of this membrane, irrespective whether or not this membrane is an initial one or it is obtained by division. This principle binds rules to labels rather than to membranes.
- The skin membrane can never divide although, as any other membrane, the skin membrane can be “electrically charged”.

1.2.3.2 Tissue-like P systems

Tissue-like P systems take inspiration from intercellular communication and communication between neurons. Consequently, instead of considering a hierarchical structure, membranes are placed at the nodes of a graph. Communication among cells is based on symport/antiport rules, which were introduced to P systems in [187]. Symport rules move objects across a membrane together in one direction, whereas antiport rules move objects across a membrane in opposite directions [162]. Martín-Vide *et al.* [134, 133] introduced the concept of tissue-like P systems as follows:

Definition 1.3. A tissue P system of degree $m \geq 1$, is a tuple $\Pi = (\Gamma, M_1, \dots, M_m, syn, i_{out})$, where

- Γ is a finite alphabet.
- $syn \subseteq \{1, 2, \dots, m\} \times \{1, 2, \dots, m\}$ (synapses among membranes).
- $i_{out} \in \{1, 2, \dots, m\}$ indicates an output membrane, in which the result of computations will be encoded.
- M_1, \dots, M_m are membranes of the form $m_i = (Q_i, s_{i,0}, w_{i,0}, P_i)$, $1 \leq i \leq m$, where:
 - Q_i is a finite set of states.
 - $s_{i,0} \in Q_i$ is the initial state.
 - $w_{i,0} \in E^*$ is the initial multiset of impulses.
 - P_i is a finite set of rules of the form $sw \rightarrow s'xy_{go}z_{out}$, where $s, s' \in Q_i$, $w, x \in M(\Gamma)$, $y_{go} \in M(E \times \{go\})$ and $z_{out} \in (E \times \{out\})$, being $z_{out} = \lambda$ for all $i \in \{1, 2, \dots, m\}$ different from i_{out} .

Consequently, the authors define a *configuration* in a tissue-like P system as a tuple of the form (s_1w_1, \dots, s_mw_m) , with $s_i \in Q_i$ and $w_i \in E^*$, $(1 \leq i \leq m)$. Similarly, they define $(s_{1,0}w_{1,0}, \dots, s_{m,0}w_{m,0})$ as the *initial configuration* of Π . Păun *et al.* [164] proposed a variant for this model, introducing cell division as a way of creating new membranes. They formalized such a model as follows:

Definition 1.4. A tissue-like P system with cell division of degree $m \geq 1$ is a tuple:

$\Pi = (\Gamma, \Sigma, M_1, \dots, M_m, \mathcal{E}, R, i_{in}, i_{out})$ where:

- Γ is a finite alphabet and $\Sigma \subset \Gamma$
- M_1, \dots, M_m are multisets over Γ .
- $\mathcal{E} \in \Gamma$ is an alphabet representing the set of objects in the environment in arbitrary copies of each.
- R is a finite set of rules of the following forms:

Communication rules: $(i, v/v, j)$, where $i, j \in \{0, 1, \dots, m\}$, $i \neq j$, $u, v \in M(\Gamma)$. The set $\{1, 2, \dots, q\}$ identifies the cells of the system, 0 is the environment; when applying a rule $(i, v/v, j)$ objects in u are sent from region i to region j and simultaneously the objects of the multiset v are sent from region j to region i .

Division rules: $[a]_i \rightarrow [b]_i [c]_i$, where $i \in \{1, 2, \dots, q\}$, $a, b, c \in \Gamma$ and $i \neq i_0$. When applying such a rule under the influence of object a , the cell with label i is divided in two cells with the same label; in the first copy the object a is replaced by b , in the second copy the object a is replaced by c ; all other objects are replicated and copies of them are placed in the two new cells.

- $i_{out} = 0$ denotes the environment and $i_{in} \in \{1, \dots, m\}$ denotes the input cell.

Communication rules define a non-directed graph connecting the cells in the system. In this model, rules are applied in a non-deterministic, maximally parallel way. It is noteworthy that there are not explicit evolution rules; instead, objects are interchanged with the environment i_0 , which has an arbitrary number of objects and, consequently, can provide the objects specified by the rule. In addition, if a cell is divided, then the division rule is the only one which is applied for that cell in that step, its objects do not evolve by means of communication rules.

1.2.3.3 Spiking neural P systems

Spiking neural P systems (*SN P Systems*, for short) is a type of P systems introduced by M. Ionescu *et al.* [111] which takes inspiration from the neurophysiological behaviour of neurons sending electrical impulses (spikes) along axons to other neurons [163]. SN P systems present some features which sharpen its dissimilarities with other models of P systems. Some of these are listed here. First, the alphabet is a *singleton*, that is, it only contains a type of object (usually noted as a and named *spike*). In addition, in their left-hand side, a regular expression is defined along with a multiset; for a rule to be applied (*fired*), its left-hand side expression must match the content of the membrane (*neuron*) in which it is applied. Besides, a rule can be applied after a number of steps (*delay*) after the conditions for its application is complied. Moreover, unlike Tissue-like P systems, in SN P systems links between neurons are explicit in the structure, rather than encoded on the rules.

Like in many P system types, there exist several variants on SN P systems. Here, as an example, a variant introduced by Pan and Pérez-Jiménez [163] is discussed. In their model, known as *SN P systems with division and budding*, new neurons can be produced by **division**, in whose case they are placed “in parallel”, or by **budding**, in whose case they are placed “serially”. It is important to remark that neurons produced by division can have labels different from each other and from the divided neuron, unlike in the previous models studied.

Definition 1.5. *A spiking neural P system with neuron division and budding of (initial) degree $m \geq 1$ is a construct of the form $\Pi = (O, H, syn, n_1, \dots, n_m, R_1, \dots, R_m, in, out)$ where:*

- $O = \{a\}$ is the singleton alphabet.
- H is a finite set of labels for neurons.
- $syn \in H \times H$ is a synapse dictionary, with $(i, i) \in syn$, for each $i \in H$.
- $n_i \geq 0$ is the initial number of spikes contained in neuron i , $i \in \{1, \dots, m\}$.
- R is a finite set of developmental rules, of one of the following forms:

extended firing rule (also called *spiking rule*) $[E/ac \rightarrow ap; d]_i$, where $i \in H$, E is a regular expression over a , and $c \geq 1, p \geq 0, d \geq 0$, with the restriction $c \geq p$.

neuron division rule $[E]_i \rightarrow []_j || []_k$, where E is a regular expression and $i, j, k \in H$

neuron budding rule $[E]_i \rightarrow []_i / []_j$, where E is a regular expression and $i, j \in H$

- $in, out \in H$ indicate the input and the output neurons of Π , respectively.

For such a model, the authors describe the following syntax [163]:

- If a neuron σ_i contains k spikes and $a^k \in L(E), k \geq c$, then the rule $[E/a^c \rightarrow a^p; d]_i$ is enabled and can be applied. This means consuming (removing) c spikes (thus only $k - c$ spikes remain in neuron σ_i); the neuron is *fired*, and it produces p spikes after d steps. If $d = 0$, then the spikes are emitted immediately; if $d = 1$, then the spikes are emitted in the next step, etc. If the rule is used in step t and $d \geq 1$, then in steps $t, t + 1, t + 2, \dots, t + d - 1$ the neuron is closed and cannot receive new spikes until d steps later.
- If (1) a neuron σ_i contains s spikes and $a^s \in L(E)$, and (2) there is no neuron σ_g such that the synapse (g, i) or (i, g) exists in the system, for some $g \in \{j, k\}$, then the division rule $[E]_i \rightarrow []_j || []_k$ can be applied. This means that after consuming all these s spikes the neuron σ_i is divided into two neurons, σ_j and σ_k . The new neurons contain no spike at the moment when they are created. They can have different labels, but they inherit the synapses that the parent neuron already has (if there is a synapse from neuron σ_g to neuron σ_i , then in the process of division one synapse from neuron σ_g to newly created neuron σ_j and another one from σ_g to σ_k are established; similarly, if there is a synapse from neuron σ_i to neuron σ_h , then one synapse from σ_j to σ_h and another one from σ_k to σ_h are established).
- If (1) a neuron σ_i contains s spikes, and $a^s \in L(E)$, and (2) there is no neuron σ_j such that the synapse (i, j) exists in the system, then the budding rule $[E]_i \rightarrow []_i / []_j$ is enabled and it can be applied. This means that consuming all the s spikes a new neuron is created, σ_j . Both neurons are empty after applying the rule. The neuron σ_i inherits the synapses going to it before using the rule. The new neuron σ_j created by budding by neuron σ_i inherits the synapses going out of σ_i before budding; that is, if there is a synapse from neuron σ_i to some neuron σ_h then a synapse from neuron σ_j to neuron σ_h is established. There is also a synapse (i, j) between neurons σ_i and σ_j .

Pan and Pérez–Jiménez point out that the model, although non–deterministic, is not maximal; only one rule per neuron can be applied (if possible) at each step. In addition, when a spiking rule is used, the state of neuron σ_i (open or closed) depends on the delay d . When a neuron division rule or neuron budding rule is applied, at this step the associated neuron is closed, it cannot receive spikes. In the next step, the neurons obtained by division or budding will be open and can receive spikes.

1.2.3.4 Kernel P systems

Last but not least, recently a new model aimed for formal verification of P systems has come into scene. This model, namely *Kernel P Systems* [77, 79, 112], defines P systems with a tissue–like membrane structure. The main novelty associated with this model is the definition of **guards**, i.e., conditions on the cardinality of objects which must be satisfied for a rule to be applied.

Definition 1.6. *Given a finite alphabet Γ , a guard over Γ can be recursively defined as follows:*

- *A guard is a tuple $\{a, op, n\}$, $a \in \Gamma \cup \bar{\Gamma}$, $op \in \{>, <, \leq, \geq, \neq, =\}$, $n \geq 0$, where $\bar{\Gamma}$ is an alphabet such that there exists a bijective relation between Γ and $\bar{\Gamma}$ and $\Gamma \cap \bar{\Gamma} = \emptyset$. Given such a relation, $\bar{a} \in \bar{\Gamma}$ represents the symbol related to $a \in \Gamma$*
- *Let g_1, g_n be two guards. Then, $g_1 \vee g_2$ is a guard*
- *Let g_1, g_n be two guards. Then, $g_1 \wedge g_2$ is a guard*

The semantics associated with guard $g = \{a, op, n\}$, $a \in \Gamma \cup \bar{\Gamma}$, $op \in \{>, <, \leq, \geq, \neq, =\}$, $n \geq 0$ over a membrane i at step $t \geq 0$ works as follows. Let $M_{i,t}$ be the multiset associated with membrane i at time t . Then, g is complied if the number of objects of type a in multiset $M_{i,t}$ is greater, lower, etc. than n according to the binary relation specified by op and $a \in \Gamma$. If $a \in \bar{\Gamma}$, then g is complied if and only if $g' = \{\bar{a}, op, n\}$ is not complied, where $\bar{a} \in \bar{\Gamma}$ is the symbol related to a . If g is of the form $g = g_1 \vee g_2 \vee \dots \vee g_s$, $s > 1$, then g is complied if any of the guards g_1, g_2, \dots, g_s are complied. Similarly, if g is of the form $g = g_1 \wedge g_2 \wedge \dots \wedge g_s$, $s > 1$, then g is complied if all guards g_1, g_2, \dots, g_s are complied.

Kernel P Systems define a rather complex model; they encompass a wide array of features from such a variety of P system models. Therefore, for the purpose of simplicity, a formalization on a subset of Kernel P systems known as

simple Kernel P systems (*skP systems*), as appeared in [79], is provided as an example.

Definition 1.7. *An skP system of degree $n \geq 1$ is a tuple*

$sk\Pi = (\Gamma, H, IO, C_1, \dots, C_n, \mu, i_0)$, where:

- Γ is an alphabet of symbols.
- H is an alphabet of labels.
- IO is a finite alphabet such that $IO \subseteq \Gamma$.
- C_1, \dots, C_n are compartments.
- $\mu = (V, E)$ is an undirected graph, where $V \subseteq H$ is the set of vertices and E is the set of edges.
- $i_0 \in L$.

An *skP* system, $sk\Pi = (A, L, IO, C_1, \dots, C_n, \mu, i_0)$, can be viewed as a set of n compartments, C_1, \dots, C_n , interconnected by edges from an undirected graph μ . IO is the alphabet of the environment objects. Each compartment is identified by a label of H and has initially a multiset over Γ , and a finite set of rules. The compartment receiving the result of a computation is denoted by i_0 ; in the sequel, this will always be the environment. An h -membrane division rule $[x]_l \rightarrow [y_1]_{l_1}, \dots, [y_h]_{l_h} \{g\}$ associated with a compartment $C = (l, w_0, R_l)$ is applicable at a given instant to the current multiset z if the guard g is evaluated true with respect to z and the object x is in the multiset z . When applying such a rule to x , the compartment labelled by l will be replaced by h compartments labelled l_1, \dots, l_h and x is replaced by multiset y_j in compartment l_j ; the content of l , but x , after using all the applicable rewriting and communication rules is copied in each of these compartments; all the links of l are inherited by each of the newly created compartments. A rewriting and communication rule $x \rightarrow (a_1, t_1), \dots, (a_h, t_h) \{g\}$ associated with a set of rules, R_l , of a compartment $C = (l, w_0, R_l)$, is applicable at a given moment to the current multiset z if the guard g is evaluated true, x is contained in z and the target $t_j \in L, 1 \leq j \leq h$, must be either the label of the current compartment, l , or the label of its existing neighbour ($\{l, t_j\} \in E$). When applying such a rule, objects a_j are sent to the compartment labelled by t_j , for each $j, 1 \leq j \leq h$. If a target t_j refers to a label that appears more than once, then one of the involved compartments will be non-deterministically chosen. When t_j indicates the label of the environment, the corresponding object a_j is sent

to the environment. In an *skP* system, the rules are applied in a maximally parallel way with the usual restriction that at most one membrane division rule can be applied per membrane in each step.

1.3 Stochastic models and Probabilistic Models

In natural phenomena, noise is present all across the board. Absolute certainty is an oddity in biology, and noisy and random processes are the norm. Therefore, when designing an biological experiment, it is simply impossible to predict the result with absolute confidence, the furthest the experimenter can go is to define a probability distribution to dictate how likely is to obtain each possible outcome of the experiment. In this section, two complementary approaches in capturing the inherent uncertainty of natural phenomena are presented.

1.3.1 Stochastic approach

Nature is, at its core, random. As a matter of fact, the vast majority of biological models implement some sort of random walk processes [47]. However, Bernoulli's law of large numbers states that, when a random experiment is repeated *ad infinitum*, the percentage difference between the expected and actual values goes to zero. As a consequence, the expected value is, by far, the most likely result, overshadowing other possible outcomes [24]. In addition, the well-known \sqrt{n} law introduced by Penrose [169] states that the influence of a particular individual in a system grows inversely proportional to the value of \sqrt{n} , where n is the number of individuals in a system. As a result, the particularities of each individual are eclipsed when the value of \sqrt{n} is large enough. Hence, deterministic models are of certain use to find the expectancy in a experiment, which is mostly relevant when the number of repetitions or the number of individuals in the experiment is considerably large.

However, when it comes to biochemical experiments with a small number of molecules, deterministic experiments might shed some light only in the case that the number of conducted experiments is large enough to overcome the limitations imposed by the sample size. If it is not the situation, the need for models which incorporate mechanisms to cope with randomness becomes primordial. Judson [114] claims that the term *stochastic method* encloses all methods which integrate, at some point, a certain degree of randomness. She

also differentiates two types of such algorithms: *Monte Carlo* methods, in which probabilities and the effect of random events are coarsely (but quickly) approximated, and *Las Vegas* methods, which take a more accurate picture of the state of the ecosystem at the expense of computational power. In practice, the boundary between these categories is rather blurred, so models are distributed over a continuum between these two edges rather than unequivocally associated with one of such types.

In this sense, the so-called *Gillespie Algorithm* [80, 232] (a.k.a. *Gillespie Direct Method*) is a stochastic method devised to simulate chemical phenomena which explores the state space associated with a system of ordinary differential equations (*ODE*) with as many equations as possible states exist in the system. This algorithm generates random events based on the *propensity* of biochemical reactions. Gillespie method makes the following assumptions [170]:

- The volume of the system under study is *constant* (V) and its temperature is *stable*.
- The molecules in the system can be classified in a finite, discrete series of $n \geq 1$ species, in such a way that the state of the system at time $t \geq 0$ can be described as a **vector** $X(t) = (X_1(t), \dots, X_n(t))$, where $X_i(t) \geq 0, 1 \leq i \leq n$, represents the number of molecules of type i at time t . This assumption shows similarities with the approach fostered by so-called *cohort models* [205, 86] widely used in ecology. These systems group individuals in a series of categories according to relevant properties (age, sex, body weight, etc.), in a similar fashion as described in this algorithm.
- The system is *autonomous*, that is, there exists no external input which influences the state of the system at any instant.
- There exist a finite number $M \geq 1$ of **chemical reactions** $\{r_1, \dots, r_m\}$ which transform each state $X(t)$ into $X(t+1)$. Each reaction $r_j, 1 \leq j \leq m$, is composed of:
 - A **state change vector** $v_j = (v_{1j}, \dots, v_{nj}), v_{ij} \in \mathbb{Z}$ which describes the effect of the application of reaction r_j upon the molecules in the system.
 - A **propensity function** $p_j(X(t))$ which indicates how likely is r_j to be applied on state $X(t)$. In this sense, $p_j(X(t))dt$ defines the probability of applying a reaction of type r_j in interval $[t, t + dt)$.

$p_j(X(t))$ is calculated out of c_j known as *stochastic constant*. Constant c_j is, in turn, computed from *kinetic constant* k_j , which is usually deduced from experimental data.

- An integer $h_j \geq 0$ specifying the number of **possible reactant combinations** upon the application of r_j .

The algorithm works as follows. On each step t , a chemical reaction is randomly chosen according to probabilities $p_j(X(t))$, $1 \leq j \leq m$. The chosen reaction is applied in time lapse $[t, t + \tau_m)$. τ_m is known as *waiting time*, and represents the time it takes for the reaction chosen in step t to be applied. A version of the algorithm specified in [170] is described in Algorithm 1.3.1:

Algorithm 1.3.1 Gillespie Direct Method

Input:

$T \geq 0$: iterations of the algorithm.

m : medium where the reaction takes place.

$X(0) = \{X_1(0), \dots, X_n(0)\}$: a vector representing the number of initial molecules of each species.

v_1, \dots, v_m , **where** $v_j = \{v_{1j}, \dots, v_{nj}\}$, ($1 \leq j \leq m$): j vectors describing the effect of the application of the reactions.

p_1, \dots, p_m , **where** p_j , ($1 \leq j \leq m$) **is a function of \mathbb{N}^n over $[0, 1]$** , that is, functions describing the probability for each reaction to be applied.

k_1, \dots, k_m : kinetic constants.

h_1, \dots, h_m : integer numbers greater or equal to 0 representing possible combinations of reactants upon the application of each reaction.

- 1: **for** $t = 0 \rightarrow T$ **do**
 - 2: $a_t \leftarrow \sum_{j=1}^m p_j$, where $p_j = h_j \cdot k_j$, $1 \leq j \leq m$, is the propensity function of r_j
 - 3: $b_0, b_1 \leftarrow$ two normally-distributed random numbers in $(0, 1)$
 - 4: $\tau_m \leftarrow \frac{1}{a_t} \cdot \ln(\frac{1}{b_1})$
 - 5: Choose the only j_t such as $\sum_{k=1}^{j_t-1} p_k < r_2 \cdot a_t < \sum_{k=1}^{j_t} p_k$
 - 6: $X(t + \tau_m) \leftarrow X(t) + v_{j_t}$
 - 7: $t \leftarrow t + \tau_m$
 - 8: **end for**
-

Gillespie Direct Method has been applied on a plethora of biochemical systems [232], ranging from cellular growth [129, 130, 34] to apoptosis [41].

1.3.2 Probabilistic approach

Probabilistic approaches propose an alternative way of handling noise and randomness when simulating real-life phenomena. Given a state in a system,

probabilistic conceptualisations define probability distributions which determine the likelihood of reaching a set of possible following states. Thus, evolving a state in a system is analogous to picking up a choice at random among the set of following states. These probability distributions can be obtained from experimental data or attuned through calibration methods [170].

Markov chains are a paradigmatic example of a probabilistic model [66]. They define a set of states such as transitions among them are dictated by probability distributions. However, some approaches bridging Markov chains and stochastic methods can be found in the literature [81]. Two probabilistic models in Membrane Computing will be later described in depth in this document. Population Dynamics P systems (*PDP systems*) are a novel and effective computational tool to model complex problems characterized by the ability to work in parallel [51]. In addition, Probabilistic Guarded P systems (*PGP systems*), a brand new model introduced in this thesis, will be formalized in Chapter 5.

Chapter 2

Simulation of P Systems

Given a real-life process subject to study and a model thereof, in order to verify that the behaviour of the model corresponds to that of the system it is usually mandatory to **simulate** the system. Rather than simulating, one alternative approach would consist on **implementing** the models. However, up to the present date, there has been no success in implementing P systems, whether on electronic or on biological substrates, mostly due to the inherent hurdles in exponential creation of computational devices. Therefore, in the current stage, simulation remains as the only way out.

This chapter is a general survey on simulators in Membrane Computing, describing representative examples since the introduction of the discipline. This chapter is structured as follows. First, a small introduction to origins in Membrane Computing simulation is provided. Section 2.1 focuses on the state of the art prior to the advent of P-Lingua and the change of paradigm it supposed in Membrane Computing. Section 2.2.1 discusses P-Lingua, a framework for the simulation of P systems. Section 2.3 lists some Membrane Computing simulators on parallel architectures. Finally, Section 2.4 narrows the field down to GPU simulators in the discipline.

2.1 Simulators in Membrane Computing

When talking about software, a *simulator* is a program such as, given a system and a simulator thereof, both return an equivalent output. It is essential to point out that returning an equivalent output does not necessarily mean that they both behave exactly in the same way. For instance, a P system with active membranes [173] is capable of creating an exponential number of computing devices in an efficient manner, whereas its simulator is restricted

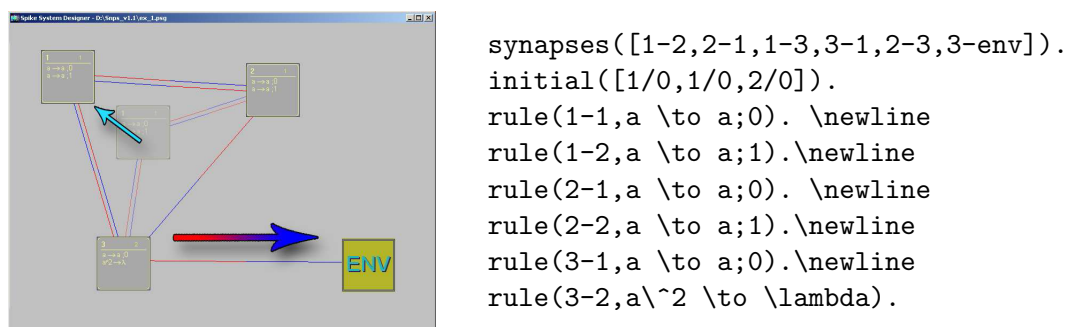


Figure 2.1: Output screen and code sample of SNPS

to the hardware constraints of the platform in which it is run. That is, it is unable to create such a number of computing devices because it can only access a finite number of processors. To overcome this limitation, the simulator might create *virtual* computing devices, but it would be constrained again by physical limitations: 1) the finite memory of the system and 2) the fact that the calculations carried out by these virtual devices are eventually run by the aforementioned finite processors.

The first Membrane Computing simulator was developed by Malița [131] with the intent to serve as a tool for educational and research assistance purposes [196]. This simulator takes advantage of rule-based dynamics of P systems and chose a declarative programming language like Prolog to capture best their semantics. In this simulator, the system to simulate was hard-coded in the application; it did not provide any means to input the system to simulate apart from modifying the code. Shortly, graphical representations became a common feature in Membrane Computing simulators. For instance, another primary simulator by Ciobanu and Paraschiv [44] already included a graphical interface to show the evolution of the simulated P system throughout time. Some authors developed software applications in which the input is given as raw text, whereas the system evolution is graphically displayed. For instance, Ramírez-Martínez *et al.* [196] developed SNPS, a simulator for Spiking Neural P systems in which the data is input on a simple, close-to- \LaTeX text format, whilst the output is a sophisticated graphical user interface (*GUI*) in which the evolution of the system can be seen as the simulator takes computation steps. Figure 2.1 displays the main screen of SNPS.

In Membrane Computing, simulators have been usually employed for three primary purposes, which are:

Educational purposes: These simulators address the question: *How does*

a P system work? It is noteworthy that, despite being computational models, P systems are different from silicon computers in many aspects. Like computers, P systems are discrete systems, they are composed of discrete entities (objects, membranes, rules, etc.) and time is discrete and advances in finite steps [188]. However, unlike conventional computers, P systems are non-deterministic devices. Of course, there exist exceptions to the rule, such as deterministic Enzymatic Numerical P Systems [221], as well as models which guide transition steps by using stochastic functions [177] or probabilities associated to rules [51], but in most cases the multiset of rules to apply on a given configuration is not unique. Therefore, the functioning of P systems might be not intuitive for the newcomer and, consequently, it is advisable to count on simulation tools which initiates students into the discipline.

Modelling assistance: Taming the semantics of Membrane Computing frameworks does not necessarily need to be straightforward for the designer. In this sense, it is primordial to know *what a P system does*. As the complex systems they are, they are capable of describing rather complex and intricate behaviours out of simple rewriting rules. When devising a P system, the designer has already in mind the desired functionality; it is possible (and likely) that this functionality gets enriched and evolves through time, but these are changes commonly due to decisions made during the design process, defining an iterative cycle based on **simulating** and **modifying** the P system. All in all, a Membrane Computing simulator must report the designer what is the state of a P system at every reached configuration and which are the rules selected for its evolution, so the designer knows how its P system behaves.

Core component of a larger application: P system users do not necessarily need to know that they are using a P system. In fact, sometimes they use applications as a black box, without guessing that the inference engine they are using is in fact a P system simulator. In this sense, the question to answer is *What can you tell me about my problem?* For instance, let us suppose the case of a mathematician trying to know if a problem is NP-complete. If this hypothesis is true, then it can be solved by a non-deterministic machine in polynomial time. In this situation, the mathematician formalizes its problems and obtains a results from the application, being this result computed by a hidden P system simulator. The same situation might take place when ecologists input parameters in an application and obtain the most likely outcome according to these

parameters. The ecologists want to know what is going to happen in their ecosystem, but they do not necessarily need to know that it is a P system which provides the answer.

Traditionally, Membrane Computing simulators were completely *ad-hoc* applications for the model at hand, simulator parameters were hand-coded and their code was not reusable, as it was intended to work for a specific P system [202, 54, 175]. Although they did suit the specific needs in each case, *ad-hoc* simulators lacked the needed standardisation. A preliminary attempt for this standardisation is SimCM [148], a software tool consisting on a simulation engine and a GUI to input the system to be simulated. The tool intends to move Membrane Computing closer to biologists, as a modelling framework for biochemical phenomena. It allows the modeller to create, save and load P systems and edit them with graphical *widgets*. It also displays P system computations graphically, highlighting branching configurations, and includes a debug environment which reports the user bugs in the designed system. Another interesting feature of SimCM is its sleek use of the well-known Model-View-Controller (*MVC*) pattern. This pattern encourages the design of software in three layers with as much independence as possible: layer **Model** stores the data (possibly on a database), layer **View**, usually consisting on a GUI, communicates the user with the application, and layer **Controller** provides the business logic, in our case, the simulation of the system. The application of this design principle permits the replacement of the simulation engine by other which suits better future releases, thus simplifying maintenance processes. Figure 2.2 displays SimCM main screen. Nevertheless, it

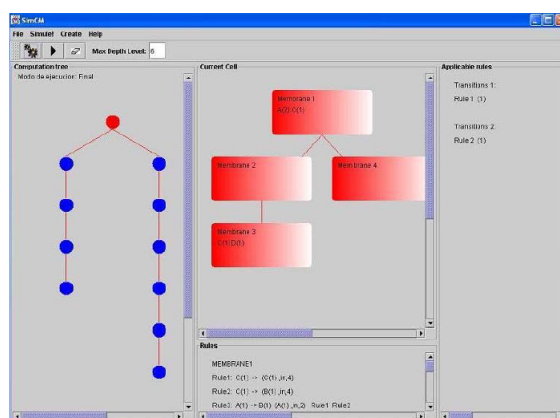


Figure 2.2: Snapshot of SimCM main screen

proved evident that defining P systems by graphical means lacked generality.

That is, one could develop a software tool for the simulation of P systems which suits the needs of an specific area. However, when it is necessary to extend this tool to new application areas, the extension proves tedious and time-consuming, usually requiring total restructuration of the whole tool. As a solution, Gutiérrez–Naranjo *et al.* [92] proposed a programming language for Membrane Computing. Literally, they claim that “the idea of a cellular programming language is possible”. Specifically, they proposed the development of a library of subroutines composed of rewriting rules. In this context, when a call to a subroutine is found, it should be replaced by its associated rules. After replacing all function calls, the authors intended to obtain a P system to solve a computationally hard problem simply by specifying parameters and programming standard function calls.

2.2 Standards in Membrane Computing

As the number of frameworks in Membrane Computing grew bigger with the passage of time, the need for standardization frameworks to avoid repeating the same code each time a new P system needs to be simulated. In this sense, P–Lingua is arguably the broadest framework in Membrane Computing, although other there exist other standards as well for specific applications.

2.2.1 P–Lingua Framework

P–Lingua [71] pioneered the standardisation of P systems by implementing an open, plugin-based software architecture meant for its extension by third-party developers. P–Lingua is a software tool which provides a specification language in which designers can define and describe P systems. This language can be easily extended when required. In addition, it also provides a set of Java [6] simulators, in such a way that users can select which simulator from those included suits better their needs. Moreover, P–Lingua implements an extension mechanism in which new formats and simulators can be enclosed in the framework.

P–Lingua was originally developed as a simulator for P systems with active membranes [63]. However, since its very beginning, further extension to cover new types of P systems was already on the horizon. The initial version of P–Lingua included many of the components present in today’s releases: an editor to define the P system to simulate, a simulator and a console to display the evolution of the P system throughout time (that is, the ongoing computation).

In fact, the raw input and output of P–Lingua has not considerably changed throughout the years, it just has been extended and generalized to take into consideration new models.

P–Lingua takes the principle of maintaining as much independence as possible between the user interface and the business logic one step beyond. It enables third–party developers to include their own formats and simulators for P system specification without changing the code, simply by programmatically implementing code snippets which define how an input should be parsed and how to simulate a type of P systems. That is, not only is the input detached from the simulator, is that even new inputs and simulators can be included without any alteration in the API. Moreover, it is possible to define new P system types in P–Lingua by combining pre–existing restrictions and creating new ones. This versatility is what gives P–Lingua its assets (complete freedom for external developers to define custom types of P systems) as a Membrane Computing standard.

2.2.1.1 Software architecture

P–Lingua consists on a Java standalone software API (namely *pLinguaCore*) which performs two operations: 1) simulate a P system and display the results and 2) translate a P system between two (presumably different) formats. The key feature of P–Lingua is that both formats and simulators are decoupled the system; the API does not know which format is being parsed or which simulation algorithm is being applied.

To customize a *pLinguaCore* release and incorporate new formats and simulation algorithms, the developer simply needs to register these components in a set of XML files. On a *pLinguaCore* simulation, the user specifies the location of the input, the format in which this input is encoded and the simulation algorithm to apply, as well as simulation parameters (number of steps, trace caching, etc.). Then, the API reads through these files looking for the implementation both of the format parser and of the simulation algorithm. If it proves unable to find any of these, *pLinguaCore* returns an error. Otherwise, the API reads the file by delegating on the format parser. First, it identifies the P system type defined in the input and checks if there exist a definition in its XML files for this type. If it does not exist, *pLinguaCore* outputs a failure message and halts. Otherwise, the API reads the input file. If this file contains any errors or does not comply with the restrictions defined for its type, the execution halts and error messages are output. This information is intended to serve as a guide for debugging the P system. On the other hand, if the file

specification correctly describes a P system, then warning messages (if any) are output and pLinguaCore simulates its P system according to the parameters above. On the contrary, on a parsing execution, pLinguaCore follows the steps listed above up to the point of simulation. Then, it checks for a format writer according to the output identifier given as parameter. If the API cannot find it, then an error message is output and the execution halts. Otherwise, the pLinguaCore delegates on the format writer the P system output in the specified file. Figure 2.3 summarizes this workflow. More detailed information about pLinguaCore architecture and execution modes can be found in [71].

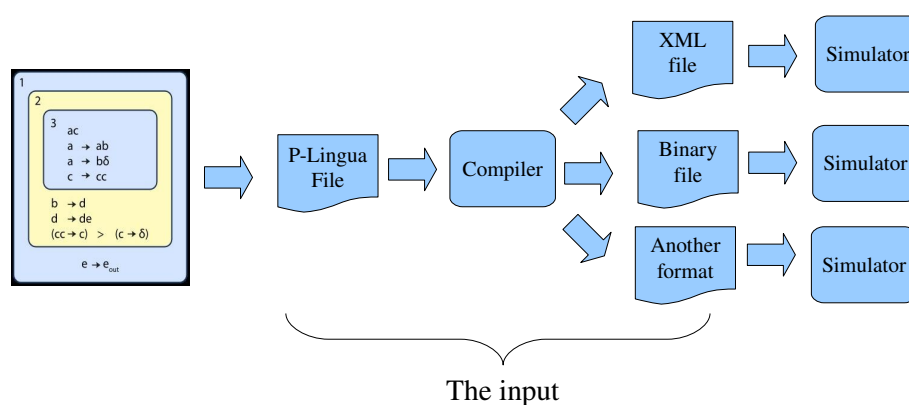


Figure 2.3: A diagram on P-Lingua software architecture

2.2.1.2 P-Lingua Language

Among the input formats available in pLinguaCore releases, P-Lingua provides a special programming language. In this language, P systems are specified on a syntax close to experts' syntax. In [196], a language to specify P systems was proposed. However, it was specifically designed for SN P systems and lacked generality to include new models. The main features of P-Lingua are its modularity, i.e., possibility to describe P systems by means of interchangeable rule modules and its similarities with the mathematical syntax used by Membrane Computing experts. P-Lingua includes various interesting features which ease the way in which P systems are specified: ability to group rules in parametric modules, decoupling of rules, multisets and initial membrane structures, numeric parameters to be instantiated at any point in the input and, above all, parametric rule patterns associated with iterators looping through

numeric variables. At the time of parsing, these patterns are unwrapped and sets of non-parametric rules are instantiated. Although P-Lingua grammar has incorporated noticeable changes since its initial release, current versions are backwards-compatible. In [71], P-Lingua language version 2 is specified, whereas version 4 is available at [12]. Some of the features of this language are the following:

- Comments are *C-like* (that is, delimited by `/**/`).
- The variant to simulate is selected with `@variant<selected_variant>`.
- Modules are defined with the reserved word `def`. There exists a main module (`main`) which is applied on each simulation and can call another ones.
- Rules are syntactically described in a similar way to the notation used by the experts.
- The initial membrane structure is defined with the reserved word `@mu`. Likewise, the initial multiset of each membrane is defined with the reserved word `@ms`.

Here, a small code snippet compliant with P-Lingua version 2 taken from [12] is included as an example. This code describes a P system which solves the SAT problem in polynomial time on a Membrane Computing framework with active membranes [178].

```
1  /*
2  * SAT.pli:
3  * This P-Lingua program defines a family of recognizer P systems
4  * to solve the SAT problem.
5  */
6
7  /* Module that defines a family of recognizer P systems
8  * to solve the SAT problem */
9  @model<membrane_division>
10 def Sat(m,n)
11 {
12  /* Initial configuration */
13  @mu = [[]'2]'1;
14
15  /* Initial multisets */
16  @ms(2) = d{1};
17
```

```

18  /* Set of rules */
19  [d{k}]'2 --> +[d{k}]-[d{k}] : 1 <= k <= n;
20
21  {
22    +[x{i,1} --> r{i,1}]'2;
23    -[nx{i,1} --> r{i,1}]'2;
24    -[x{i,1} --> #]'2;
25    +[nx{i,1} --> #]'2;
26  } : 1 <= i <= m;
27
28  {
29    +[x{i,j} --> x{i,j-1}]'2;
30    -[x{i,j} --> x{i,j-1}]'2;
31    +[nx{i,j} --> nx{i,j-1}]'2;
32    -[nx{i,j} --> nx{i,j-1}]'2;
33  } : 1<=i<=m, 2<=j<=n;
34
35  {
36    +[d{k}]'2 --> []d{k};
37    -[d{k}]'2 --> []d{k};
38  } : 1<=k<=n;
39
40  d{k}[]'2 --> [d{k+1}] : 1<=k<=n-1;
41  [r{i,k} --> r{i,k+1}]'2 : 1<=i<=m, 1<=k<=2*n-1;
42  [d{k} --> d{k+1}]'1 : n <= k <= 3*n-3;
43  [d{3*n-2} --> d{3*n-1},e]'1;
44  e[]'2 --> +[c{1}];
45  [d{3*n-1} --> d{3*n}]'1;
46  [d{k} --> d{k+1}]'1 : 3*n <= k <= 3*n+2*m+2;
47  +[r{1,2*n}]'2 --> -[r{1,2*n}];
48  -[r{i,2*n} --> r{i-1,2*n}]'2 : 1<= i <= m;
49  r{1,2*n}-[]'2 --> +[r{0,2*n}];
50  -[c{k} --> c{k+1}]'2 : 1<=k<=m;
51  +[c{m+1}]'2 --> +[c{m+1}];
52  [c{m+1} --> c{m+2},t]'1;
53  [t]'1 --> +[t];
54  +[c{m+2}]'1 --> -[Yes];
55  [d{3*n+2*m+3}]'1 --> +[No];
56
57  } /* End of Sat module */
58
59  /* Main module */
60  def main()
61  {
62    /* Call to Sat module for m=4 and n=6 */
63
64    call Sat(4,6);

```

```
65
66 /* Expansion of the input multiset */
67
68 @ms(2) += x{1,1}, nx{1,2}, nx{2,2}, x{2,3},
69          nx{2,4}, x{3,5}, nx{4,6};
70
71 /* To define another P system of the family, call the Sat
72    module with other parameters and expand the input
73    multiset with other values */
74
75 } /* End of main module */
```

Since its first version, P–Lingua language has been continuously extended to include new types of P systems. At the time of writing the present document, pLinguaCore (and, consequently, P–Lingua language) development history consists on the following versions:

1. Version 1.0 allows solely the definition and simulation of P systems with active membranes [63], a type of P systems in which membranes have associated electrical charges and create new membranes by division [174]. This first release of P–Lingua includes a GUI with a graphical canvas to display the current configuration of the P system simulated. As only active membrane P systems are allowed, this canvas is specifically to display the structure of such systems.
2. Version 2.0 incorporates the aforesaid extension mechanism [71]. Henceforth, every new format parser and writer, P system type and simulation algorithm are incorporated by drawing on this mechanism. The types included in this release are:
 - Transition P systems [179].
 - Symport/antiport P systems [69].
 - P systems with active membranes [189].
 - P systems with active membranes and membrane creation rules [91].
 - Stochastic P systems [210].
 - Probabilistic P systems [141].
 - Kernel P Systems [79].

These types are defined by programmatically combining restrictions. For instance, type *Active membranes* does not allow membrane creation, that

is, explicit creation of a new membrane inside another, whereas *Transition P systems* does not allow membrane division, which consists on dividing a membrane into two or more. This release also includes simulation algorithms for these types and two new formats: XML (both parser and writer) and binary (only parser). P-Lingua language was extended to include the features defined by these P system types. For instance, probabilities and constants associated to rules are characteristics of stochastic and probabilistic P systems, whereas membrane creation rules is a feature of P systems with active membranes and membrane creation rules. In addition, a new sentence `@model<system_type>` was integrated, so as to stipulate the type of P system to parse among those available.

3. Version 2.1 incorporates tissue-like P systems [191] into the set of available types, along with a simulator compliant with its semantics [140]. This extension entails introducing tissue-like membrane structures (a directed graph) into P-Lingua language, along with tissue-like, swapping rules. The syntax of these rules is $[u]'label_1 \leftrightarrow [v]'label_2$, being u and v multisets over the system's alphabet. This syntax might be misleading; any of such rules requires for its application that $label_1$ contains multiset u and membrane $label_2$ contains multiset v . Then, rather than consuming objects in u or v , these objects are swapped, in such a way that, after the application of the rule, objects in u are now in membrane $label_2$ and objects in v are now in membrane $label_1$.
4. Version 3.0 incorporates two new P system types and two simulators thereof: Spiking Neural P systems (SN P systems) [108] and Probabilistic Dynamic P Systems (PDP systems) [51]. Regarding SN P systems, the ensuing extension of P-Lingua language proved to be quite a laborious task. These systems present some specificities which break away from the types introduced so far; like tissue-like P systems, their structure consists on a directed graph. However, unlike tissue-like P systems, this graph is not defined by the rules but rather it is embedded in the membrane structure. Moreover, its rules define regular expressions which dictate when rules are fired, in contrast to types mentioned so far in which the only requirement for firing a rule is that there are enough objects and that the required polarization matches. In addition, in these systems the input is not static; it is encoded on a sequence (train) of spikes which, consequently, had to be included in P-Lingua syntax. On top of it, the version of SN P systems incorporated defines membrane division and

budding [163], features which define rules creating both new membranes and links to previously existing ones.

On the other hand, PDP systems are an active research line as models for ecosystems. They are capable of capturing the randomness inherent in ecological processes to predict the evolution of species and another ingredients according to pre-existing literature data. This release also includes the Direct Non-Deterministic Probabilistic algorithm (*DNDP*) [141], which is inspired on the Direct Non-Deterministic (*DND*) algorithm [151] and the Direct distribution based on Consistent Blocks Algorithm (*DCBA*) [139, 138]. PDP systems are covered in detail in Chapter 4.

5. Finally, version 4.0 incorporates Kernel P Systems [77], a framework for numerical problems in Membrane Computing which has been specifically designed for formal verification by applying model checking techniques on P systems. Moreover, this version includes an implementation of the Direct distribution based on Consistent Blocks Algorithm (*DCBA*) [138].

2.2.1.3 MeCoSim

pLinguaCore is a standalone API with a command-line interface. It provides not GUI whatsoever. The idea is that, for each specific field of application, a GUI is developed and connected to pLinguaCore. After some applications of pLinguaCore (mainly in the field of ecology [52]), it became evident that most GUIs share common features which are independent from the field. However, although some software components could be reused, every time that pLinguaCore was applied to a new field, the specificities of each case made necessary the development of new a GUI from scratch. MeCoSim [172] addresses that problem by generating GUIs out of configuration files. Essentially, MeCoSim reads a spreadsheet with the settings of a specific application (input parameter fields, output charts, statistical treatment of the data, etc) and instantiates a new GUI tailored for the problem at hand. Application settings in MeCoSim are a programming language on their own, as the variety of possible applications is quite diverse (ranging from formal verification of P systems [112, 79] to simulation of gene network dynamics [75] and including ecosystems [7, 50, 49]). In addition, MeCoSim enables a mechanism to incorporate plugins in the application, so that developers can define their own functionality to interact with the GUI.

MeCoSim GUIs define two views, each one addressing the needs of a specific

user profile: the *end user* and the *designer user*. Following the philosophy from Section 2.1, the end user does not need to know anything about Membrane Computing, and the program behaves as a black box for him. The goal of the end user is to develop virtual experiments on the phenomenon under study, thus allowing him to define the simulation parameters, simulate scenarios and generate charts with meaningful statistical information. On the contrary, the designer user does need to know about the P system used, being responsible for designing, debugging and validating the family of P systems used by the program. Thus, the role of the designer user is to validate the designed P system by comparing its simulations with data available from the phenomena under study. For this purpose, MeCoSim facilitates designer users the same functionalities than those available for end users plus a few more related to the design process of P systems: edition, compilation, simulation and selection of number of steps per application cycle. An application automatically generated by MeCoSim is shown on Figure 2.4. More information about MeCoSim and downloadable releases are available on [7].

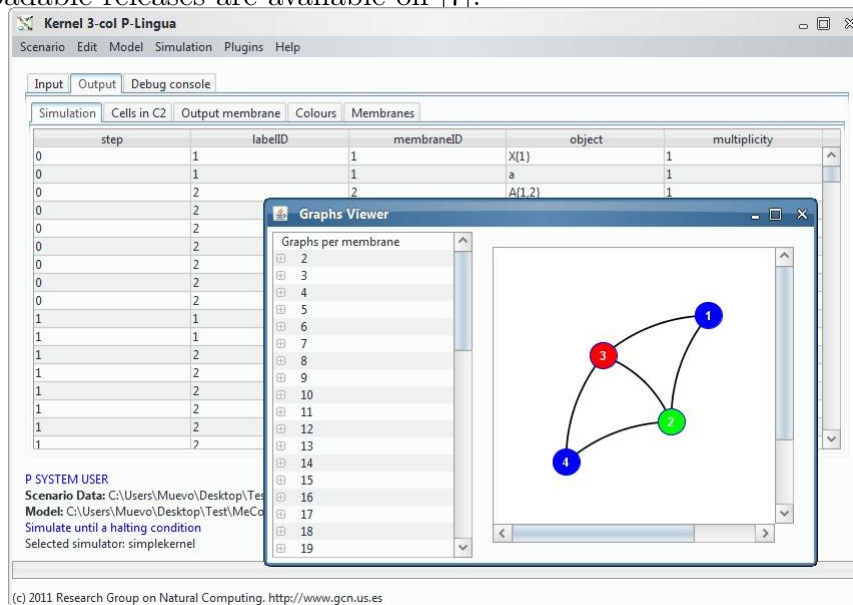


Figure 2.4: A MeCoSim-generated GUI application

2.2.2 Other standards in Membrane Computing

Apart from P-Lingua, there exist other standards for the specification and simulation of P systems. Although, to the best of the author's knowledge, P-Lingua is the only standard which intends to address a broad spectrum of

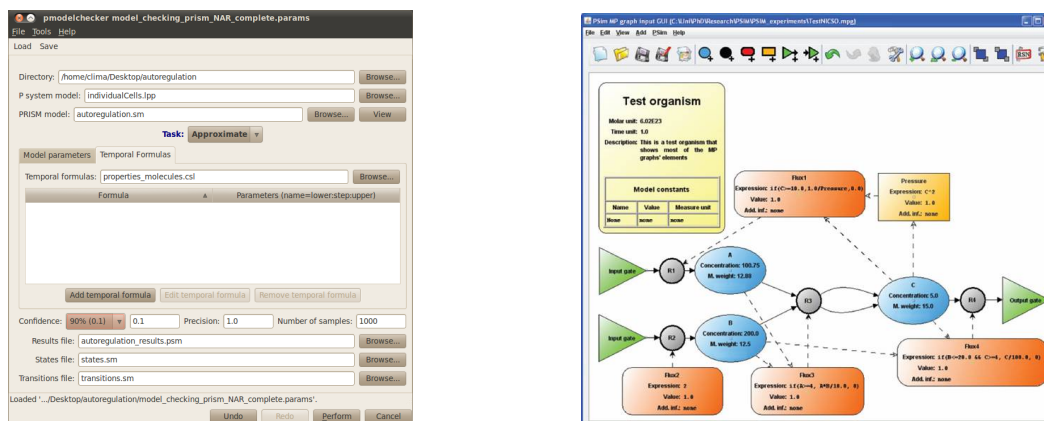
types of P systems, these other standards are software projects under active development and do serve as modelling and simulation tools on specific areas. Some of these standards are *Infobiotics* [23] and *MetaPlab* [35], whose main screen can be seen on Figure 2.5.

2.2.2.1 Infobiotics

The Infobiotics Workbench [23] is an integrated software suite incorporating model specification, simulation, parameter optimization and model checking for Systems and Synthetic Biology. Infobiotics enables two simulation algorithms: stochastic simulation or numerical integration, as well as visualization in time and space. In Infobiotics, inputs can be specified in two complementary model representation languages: *mcss-SBML*, an extension of the Systems Biology Markup Language [105], and a domain specific language, implementing a type of P systems known as *lattice population P systems* [201]. *mcss-SBML* can be visually edited on well-known editing tools such as CellDesigner [70]. Infobiotics implements two different algorithms for model simulation: ODE solvers provided by GNU scientific library [84] and a set of multicompartmental algorithms [201] based on widespread Gillespie Direct Method [80]. In addition to simulation, Infobiotics enables two interesting features: model checking and parameter and structural optimization. The former is achieved by delegating on two model checking tools: PRISM [122] and MC2 [65], whereas the latter implements several parameter optimization algorithms (i.e. differential evolution [212] and covariance adaptation) and a structural optimization method based on evolutionary computing [32]. Infobiotics is available on [5].

2.2.2.2 MetaPlab

MetaPlab [35] is a virtual laboratory developed to assist biologists in understanding internal mechanisms of biological systems and to forecast their response to external stimuli, environmental conditions alterations and structural changes. Meta P Lab features a type of P systems known as Metabolic P systems (*MP systems*) [132]. These P systems proved to be effective in modelling biological phenomena related to metabolism. MetaPlab implements a deterministic simulation algorithm based on the *mass partition principle*, which defines the transformation rate of object populations according to a suitable generalization of biochemical laws. MetaPlab enables graphic visualization and design of MP systems based on constructs named *MP graphs*. To store these systems, MetaPlab defines a type of data structures known as *MP stores*. MP system processing is achieved by *MP plugins*, combinable software modules



Model checking on Infobiotics

An MP system simulation on
MetaPlab

Figure 2.5: Infobiotics (left) and MetaPlab (right) screenshots

which define the operations to perform on these systems. Finally, *MP vistas* zooms relevant aspects on the system at hand, thus enabling feature analysis on the designed models. MetaPlab can be downloaded from [8].

2.3 Parallel simulation of P systems

Originally, Membrane Computing simulators have been exclusively implemented on sequential architectures, such as standard personal computers, occasionally by using declarative languages such as Prolog [54]. However, the inherent performance limitations of such devices is that they do not match well with the parallel nature of P systems, so the quest for new technological approaches comes to the fore. Therefore, it is natural to resort to parallel architectures such as FPGA boards [223, 135, 152, 151], computer clusters [46], microcontrollers [90, 89] and Graphic Processor Units (*GPU*) [38, 139, 30, 113]. Their main features are next explained.

2.3.1 FPGA boards

A Field Programmable Gate Array (FPGA) circuit [204, 218, 224] 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 [218]. The seminar work on parallel simulation of P systems on FPGA boards is a Transition P

system simulator attributed to Petreska and Teuscher [180]. In their work, they deploy each membrane as 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. Here, 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 $u \rightarrow v(v_1, in_i), (v_2, out)$, where:

- i corresponds to the label of any of the membrane children.
- u, v, v_1, v_2 are multisets over the system's alphabet.

Upon the application of a rule, multiset u evolve into v , multiset v_1 is sent to membrane i and multiset v_2 is sent to the membrane's parent. 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 system's alphabet. On every step, each applicable rule issues an *applicable* signal. Let w be the multiset associated with the rule's membrane. A rule is applicable if cardinalities in u are lower or equal than those in w and no rule with higher priority is applicable at the same step. Then, 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, then looks for a disabled membrane and enables it, copying all information into the membrane registers and setting its status flag to enabled. Later on, it sets its label to that in the rule's created membrane label. When all applicable rules are applied, then 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 represented on Figure 2.6.

Another interesting work on the FPGA simulation of P systems is authored by Nguyen *et al.* [152]. In their work, they present Reconfig-P, a

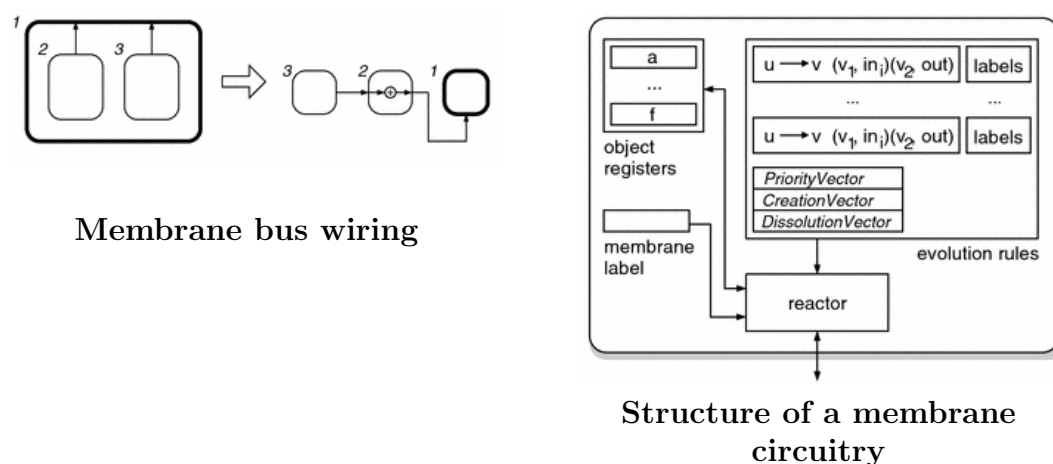
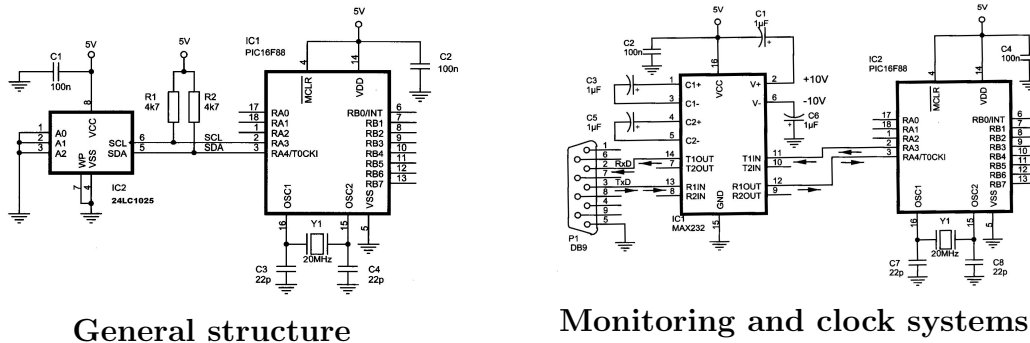


Figure 2.6: Overview of Petreska and Teuscher's FPGA Membrane Computing simulator

Java application which generates a FPGA circuit description out of Handel-C code. Handel-C [128] 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) [153] and conflict resolution strategies to resolve object competition (time-oriented and space-oriented) [149, 150, 151, 153]. 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.

2.3.2 Microcontrollers

Other parallel platforms, such as microcontrollers, have been used to simulate P systems. For instance, Gutiérrez *et al.* [90] made use of these relatively simple microcomputers to simulate P systems. In their work, they 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



General structure

Monitoring and clock systems

Figure 2.7: General structure (left) and monitoring system (right) of a Membrane Computing simulator based on micro-controller technology

used to store data. In their solution, they suggest devices of type 24LC1025 for memory storage. Due to their Harvard architecture [209, 214], 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, the authors opt for connecting the microcontrollers 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 which ensures that information concerning modification in the cardinality of objects due to the application of rules is properly transmitted. Finally, they estimate the whole cost of a system composed of 1000 membranes at about \$10000, a much more affordable solution than cluster-based platforms. Figure 2.7 displays the architecture of the simulator.

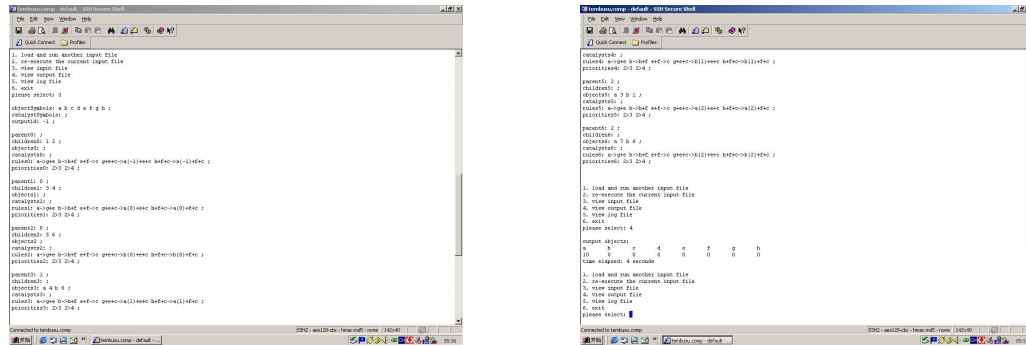
2.3.3 Computer clusters

Distributed simulation on computer clusters adds up to the parallel approaches considered to simulate P systems. In their work, Ciobanu and Guo [46] present a simulator for P systems on C++ which runs on a computer grid. Workload distribution was achieved by using Message Passing Interface (*MPI*) [161]. *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, like in aforesaid approaches, an outputs another file which contains the current configuration upon the halting of the simulator.

In their implementation, the authors allocate the simulation of each membrane 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 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 2.8. For more information, please refer to [46].

Moreover, there are also some works on the distributed simulation of P systems with *Hadoop* [64], a popular framework for Parallel Computing based on the *Map-Reduce* pattern [227]. This framework applies in parallel an operation *Map* to each data and unifies the results of these operations by applying operation *Reduce*.



```

1: load and run membrane input file
2: simulate the current input file
3: view input file
4: view output file
5: view log file
6: exit
please select: 3

membrane0: a b c d e f g h ;
objects: 1 1
membrane1: 2 ;
objects: 1 2 ;
membrane2: 3 ;
objects: a b c d e f g h ;
membrane3: 4 ;
objects: 1 2 ;
membrane4: 5 ;
objects: a b c d e f g h ;
membrane5: 6 ;
objects: 1 2 ;
membrane6: 7 ;
objects: a b c d e f g h ;
membrane7: 8 ;
objects: 1 2 ;
membrane8: 9 ;
objects: a b c d e f g h ;
membrane9: 10 ;
objects: 1 2 ;

output object:
a 1 0 0 0 0 0 0 0
b 0 1 0 0 0 0 0 0
c 0 0 1 0 0 0 0 0
d 0 0 0 1 0 0 0 0
e 0 0 0 0 1 0 0 0
f 0 0 0 0 0 1 0 0
g 0 0 0 0 0 0 1 0
h 0 0 0 0 0 0 0 1

1: load and run membrane input file
2: simulate the current input file
3: view input file
4: view output file
5: view log file
6: exit
please select: 4

output object:
a 0 0 0 0 0 0 0 0
b 0 1 0 0 0 0 0 0
c 0 0 1 0 0 0 0 0
d 0 0 0 1 0 0 0 0
e 0 0 0 0 1 0 0 0
f 0 0 0 0 0 1 0 0
g 0 0 0 0 0 0 1 0
h 0 0 0 0 0 0 0 1

please select: 5

```

Figure 2.8: Input (left) and output (right) from Ciobanu and Guo's cluster simulator

2.4 GPU Computing

Along with FPGA boards, the parallel technology on which many simulators have been developed is allegedly Graphic Processing Units (GPU). Originally,

GPUs were devised as auxiliary computers to assist the main Central Processing Unit (CPU) on carrying out graphical computations. This way, graphical data was streamlined into the GPU memory by using Direct Memory Address (DMA) [67] circuitry, thus unburdening the CPU from anything related to graphical data. 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 (also known as *Data Parallelism*). Originally, GPU processors were solely conceived to process graphic data. What is more, GPU integrate auxiliary hardware to perform common graphical tasks, such as raytracing [195], antialiasing [144, 58] and pixel shading [56]. However, it became evident that the parallel architecture of GPUs could be successfully applied for other parallel applications. 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 [160]. Nevertheless, the lack of appropriate development frameworks to implement general-purpose parallel algorithms on GPUs was a hindrance for this field of application to flourish; 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 [160].

2.4.1 CUDA programming model

The appearance of GPGPU Computing software development kits (*SDKs*) into scene did away with this requirement. In this sense, in 2007 NVIDIA announced CUDA (*Compute Unified Device Architecture*), a programming model specifically 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.

A NVIDIA GPU is composed of a set of cores or *Streaming Processors* (SPs), reaching 512 in model NVIDIA Tesla M2090 [10]. 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 [10].

In CUDA programming model [119, 160, 154], 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 on Figure 2.9.

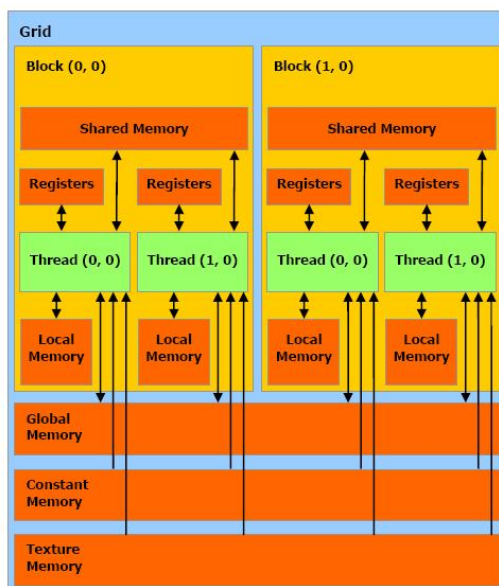


Figure 2.9: The CUDA programming model

On runtime, CUDA dynamically assigns bundles of threads or *warps* to idle SMs. Each one of these threads executes the same code on different, 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 [154]. 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 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 [38]. 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). This way, both parts of the program can cooperate in order to obtain a global result [154].

2.4.2 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 to other platforms such as FPGA boards. Unlike CUDA, OpenCL is a free standard independent from any particular company [216, 145, 11]. OpenCL is also compatible with AMD devices, such as AMD Fusion cards or Accelerated Processing Unit (APU) [1], 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 [11].

2.4.3 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 a decent simulation platform for P systems. This work has been initiated in [136] with all the simulators listed below with exception of the last one, which are available in [13].

2.4.3.1 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.* [38]. In their implementation, each step on a simulation consists on two stages: a *selection* stage and an *execution* stage. On every step, the selection stage selects which rules are to be applied according to membrane polarization and object availability, whereas in the execution stage the selected rules are applied. The simulation halts when the selection stage detects that no more rules can be applied.

In P systems with active membranes, rules can be classified in five types: *evolution* rules, *send-in* rules, *send-out* rules, *division* rules and *dissolution* rules [174]. Consequently, a kernel computes the application of each type of rules. The first kernel (that of evolution rules) also performs rule selection.

Moreover, a simulator specifically developed for a model of P systems with Active Membranes solving the SAT problem is presented in [37, 36]. 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 comply 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*, i.e., given any input for the system, all computations return the same output. In their work, they 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.

2.4.3.2 Spiking Neural P systems

Spurred by the success obtained by Cecilia *et al.*, Cabarle *et al.* [31, 29] developed a simulator of Spiking Neural P systems on the same technology. The specificities of Spiking Neural P systems propose a demanding challenge; for a rule to be applied, the spikes in the rule's associated membrane must match a regular expression. In addition, the neuron structure is explicit; links between membranes are not encoded on the rules, but are a part of the structure itself. Moreover, rules might define a time delay which states the number of step cycles after the rule issues spikes to its neighbours [108].

Due to the inherent difficulty in simulating the model, the authors started by

tackling the simulation of SN P systems without delays. The simulator is based on a matrix representation of SN P systems [236], 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 [113]

2.4.3.3 Population Dynamic P Systems

Another model of P systems simulated on GPU platforms is Population Dynamics P (PDP) systems [51], which are a framework devised to model ecosystems. Informally speaking, 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 each 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. PDP systems and associated concepts (such as rule blocks) will be further addressed in this work.

In their work, Martínez-del-Amor *et al.* [139] first developed a C++ simulator, which was further improved with OpenMP [16]. 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 on a parallel implementation of DCBA algorithm [138]. In this implementation, selection phase is divided in three stages. In the first stage, the simulator calculates a number of applications for each block in parallel by distributing objects in the rule's membrane among objects consumed by the block. As it will be reviewed later, this approach is a clear inspiration for PGP parallel simulation algorithm. In the second stage, the algorithm cal-

culates a random order in which objects are sequentially assigned to blocks, accordingly consuming block objects in a maximal way. In the third stage, the rules to be applied inside each block are chosen according to a randomly generated multinomial distribution. Finally, the execution stage generates the objects produced by the applied rules in parallel.

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.

2.4.3.4 Kernel P Systems

Recently, a simulator on simple Kernel P Systems has been developed by Ipaté *et al.* [112] The specificity of this model is that, in addition to being enough objects available, for a rule to be applied a condition over a set of objects must be satisfied. In their work, the authors test their simulator on a Kernel P System model of the Subset Sum problem [62]. The simulator executes two phases: the first stage checks which rules are applicable and selects rules to be applied among those, consuming the objects indicated by the rules, whereas the second produces the objects generated by the applied rules. However, the authors fail to specify how they implement non-determinism in their simulator, though 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.

2.4.3.5 Tissue P Systems

Last but not least, Martínez-del-Amor *et al.* [137, 61] developed a GPU-based simulator for a specific solution to *SAT* with tissue P systems with cell division. Their implementation consists on five separate stages, as follows. Their simulator consists on 5 phases: **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 on simulating a P system family which solves the SAT problem (we refer to [191] 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.

2.5 Hardware specifications

All simulations reported in this work have been performed on a laptop with a NVIDIA GTX 460M card and an Intel 7 as CPU processor (see Table 2.1).

Feature	Value
CPU Processor	Intel i7
CPU RAM Memory	8 GB DDR3
CPU Cache Memory	6 MB
CPU Clock Frequency	1.6 GHz
GPU Model	NVIDIA GTX 460M
Number of GPU Cores	192
GPU Clock Frequency	1.35 GHz
GPU Memory	1.5GB DDR5, shared
GPU Memory Clock Frequency	1.25 GHz

Table 2.1: Hardware specifications of the laptop in which the simulations in this work have been carried out

Part II
Contributions

Chapter 3

Enzymatic Numerical P systems

Enzymatic Numerical P Systems (**ENPSs**, for short) are a type of P systems which differs in some crucial aspects regarding those introduced so far in this document. Like the models described up to the present point, ENPSs are parallel systems. In addition, their membrane structure consists on a rooted (*cell-like*) tree. However, the following features are specific of this kind of P systems:

- Membranes do not have associated multisets. Instead, they contain a set of numerical, real-valued variables which evolve due to the application of **programs** in lieu of rewriting rules.
- The application of programs is **deterministic**. Each program might be associated with a variable (known as *enzyme*) which acts as a switch for the program.

ENPSs are based on Numerical P Systems (NPSs), a non-deterministic, parallel model originally aimed to model the underlying uncertainty of economical processes. To familiarize the reader with the concepts of ENPSs, first NPSs are introduced. The parallel structure of ENPSs makes them appropriate for their simulation in parallel platforms, such as GPUs, in order to simulate massive ENPS instances. In this sense, the contributions of this work to ENPSs consist on a GPU-based simulator (namely **ENPSCUDA**) and a C++ one (namely **ENPSC++**) for these systems, being developed the latter to measure the performance gain obtained by the former in comparison to sequential ENPS simulators.

This chapter is structured as follows. Sections 3.1 and 3.2 describe Numerical P Systems and Enzymatic Numerical P Systems, respectively. Section

3.3 describes sequential algorithms and software for the simulation of ENPSs. Section 3.4 describes a simulator for ENPSs for GPU platforms. Finally, Section 3.5 analyses the performance of this simulator, characterizing acceleration peaks.

3.1 Numerical P Systems

Numerical P Systems [193, 61] are a kind of P system introduced by Gheorghe and Radu Păun in 2006 [193]. 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. Although tissue-like NPSs are discussed in [221], in the original version of NPSs the membrane structure is a tree-nested, cell-like hierarchy, so no new membrane architecture is introduced in this model.

Formally speaking, a numerical P system of degree $m \geq 1$ is a tuple

$$\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \dots (Var_m, Pr_m, Var_m(0)))$$

where:

- H is an **alphabet** with m symbols used as labels of the m membranes of the system. Elements in H are the labels of the membranes in Π .
- μ is a **membrane structure**, a rooted tree, with m membranes.
- $Var_i = \{x_{1,i} \dots x_{k_i,i}\}$ is the finite set of **variables** associated with compartment i , $1 \leq i \leq m$.
- $Var_i(0) = (\lambda_{1,i} \dots \lambda_{k_i,i})$ are numerical values (*real numbers*) for the variables in Var_i . These values are considered as initial values; at instant 0 of the system evolution we have $x_{j,i} = \lambda_{j,i}$, $1 \leq i \leq m, 1 \leq j \leq k_i$.
- $Pr_i = Pr_{1,i} \dots Pr_{q_i,i}$ is the set of programs from compartment i of μ , $1 \leq i \leq m$. The l -th program $Pr_{l,i}$ from compartment i is of the form $Pr_{l,i} = (F_{l,i}(x_{1,i}, \dots, x_{k_i,i}), c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$ where $F_{l,i}(x_{1,i}, \dots, x_{k_i,i})$ is the l -th **production function** from compartment i and $c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i}$ describes the **repartition protocol**. Objects $c_{l,j}$ and v_j , $1 \leq j \leq n_i$, are, respectively, integer numbers and variables in compartment i , its parent or any of the children of compartment i .

The production function $F_{l,i}(x_{1,i}, \dots, x_{k_i,i})$ from compartment i is a real function having as variables those from this compartment. The expression $c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i}$ describes the repartition protocol which has the following meaning: let $v_1 \dots v_{n_i}$ be the set of variables from compartment i , from the parent membrane of i and for all compartments corresponding to children of compartment i . The coefficients $c_{l,1}, \dots, c_{l,n_i}$ are natural numbers that specify the proportion of the current production distributed to each variable v_1, \dots, v_{n_i} .

More precisely, at any instant $t \geq 0$, a program $Pr_{l,i}$ on each set Pr_i , $1 \leq i \leq m$, is non-deterministically chosen. Then, $F_{l,i}(x_{1,i}(t), \dots, x_{k_i,i}(t))$ and $C_{l,i} = \sum_{j=1}^{n_i} c_{l,j}$ are computed. The values of all variables on which $F_{l,i}$ depends are *consumed* and reset to 0. The value $q = \frac{F_{l,i}(x_{1,i}(t), \dots, x_{k_i,i}(t))}{C_{l,i}}$ represents the “unitary portion” to be distributed to variables v_1, \dots, v_{n_i} , according to coefficients $c_{l,1}, \dots, c_{l,n_i}$ in order to obtain the values of these variables at time $t + 1$. Specifically, variable $v_{l,j}$ will receive $q \cdot c_{l,j}$, $1 \leq j \leq n_i$, from compartment i . If a variable receives such “contributions” from several neighbouring compartments, then they are added in order to produce the value of the variable at time $t + 1$.

This model of computation was initially aimed to capture the nature and behaviour of economic processes [193]. There had been some previous works on the modelling of economic processes by means of Membrane Computing [192], proposing the application of NPSs them. In addition, a characterization of NPSs in terms of computational completeness can be found in [193], which proves that a rather restrictive type of NPSs is Turing complete, i.e., equivalent to a Turing machine in terms of computational power [155].

3.2 Enzymatic Numerical P Systems

As it is usual on Membrane Computing models, a new kind of P systems has risen as an extension of NPSs. This model is known as *Enzymatic Numerical P Systems* (ENPSs). Although this parallel model of computation has many points in common with Numerical P Systems, there are some aspects which differentiate both models. This way, in contrast to Numerical P Systems, Enzymatic Numerical P Systems describe 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.

An Enzymatic Numerical P System of degree $m \geq 1$ is a tuple

$$\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \dots (Var_m, Pr_m, Var_m(0)))$$

where:

- H , μ and $(Var_1, Var_1(0)) \dots (Var_m, Var_m(0))$ have the same meaning than in Numerical P Systems described in Section 3.1.
- Pr_i is the set of programs associated with membrane i . Each l -th program in set Pr_i may have one of the following forms:
 - $Pr_{l,i} = (F_{l,i}(x_{1,i}, \dots, x_{k_i,i}), c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$
 - $Pr_{l,i} = (F_{l,i}(x_{1,i}, \dots, x_{k_i,i}), (e_{l,i} \rightarrow), c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$

In both forms, all values which also appear in Section 3.1 have the same meaning, with $e_{l,i}$ being a variable in Var_i . This variable is known as the *enzyme-like* variable associated with $Pr_{l,i}$ and its value cannot be consumed by this program. Enzyme-like variables are exclusive ingredients of ENPSs. That is, they do not appear in NPSs.

The main novelty introduced by ENPSs has to do with the use of enzyme-like variables to control the execution flow of programs. This way, each program may have an associated enzyme-like variable which controls its application. If a program is to be applied at instant t , then this program is *active* at that instant. On each computation step, all active programs in each membrane are applied in parallel. Programs in ENPSs are applied the same way than in NPSs. However, a program is active only in the following cases:

- The program does not have an associated enzyme.
- The program has an associated enzyme and the value of this enzyme is greater than the minimum of the values of the variables consumed by the program.

In [165], Pavel *et al.* make a point about the numerical nature of enzymes in their model, claiming that it must be clear that the enzymatic mechanism was inspired by biological processes, but ENPSs themselves do not aim at modelling chemical reactions. So the proposed enzymatic mechanism will not constrain the computational model by taking in consideration all the real biological facts. In this sense, variable values can be real numbers (also negative) and production functions may be polynomials of any degree. A chemical reaction could be modelled only by first degree polynomials, but in ENPSs it is

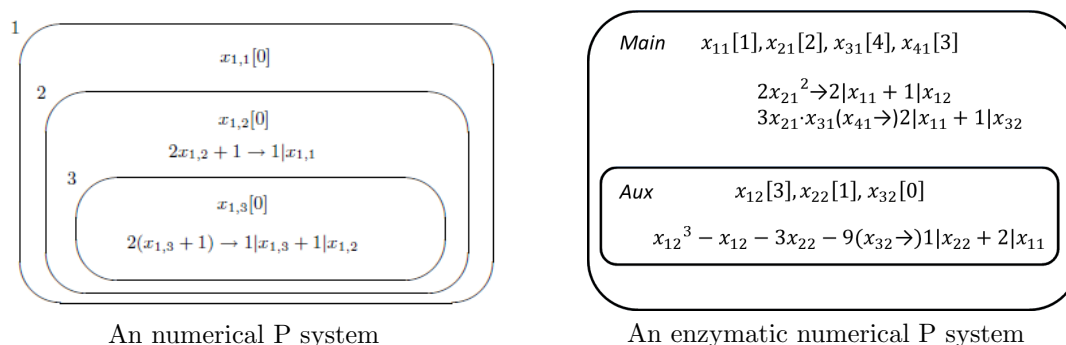


Figure 3.1: A numerical P system (left) and an enzymatic numerical P system (right)

possible to define production function consisting on polynomials with degree greater than 1. The enzymatic mechanism can be generalized by a boolean expression associated with a production function. Based on the value of the boolean expression, the production function would be active (selected) or inactive.

Both NPSs and ENPSs have been successfully applied for modelling robot controllers [28, 166, 168]. Pavel *et al.* [167] identified two advantages of using ENPS in comparison to traditional P systems with associated multisets of objects:

- Variables can be assigned real numbers. Consequently, there is no need for extra effort to simulate floating point operations.
- Membranes and variables inside are the same throughout all the computation; the memory required to store membranes and variables does not grow. That is to say, it is not possible that new objects enter or exit any membrane, nor that new membranes are created.

Moreover, Pavel *et al.* [166] also identified several advantages in using ENPS models in comparison to NPS models, some of which are:

- Membrane representation is very efficient for designing robotic behaviours. Consequently, a fewer number of programs than in NPSs is required to attain the same behaviour.
- Enzyme variables, if existent, control the program flow and detect any existing termination condition.

ENPS models have proved to be universal [221], with improved results in [222] and [124]. Moreover, ENPS-based models for deterministic mobile robot

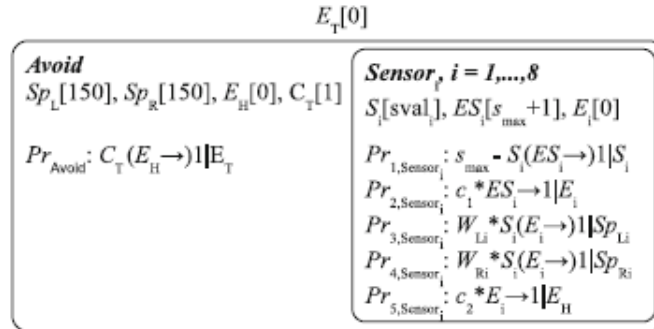


Figure 3.2: An ENPS model for obstacle avoidance

controllers have been successfully used to model obstacle avoidance [166] and odometric localization [168]. Both models were simulated with SNUPS [157, 27], a graphical, interactive Java simulator which reads an ENPS description on XML format and outputs the result of the simulation, querying the user for inputs when required. Here, the first case proposed by Pavel *et al.* [166] is presented as an illustrative example. This model tackles the question of obstacle avoidance in a closed circuit, a well-known problem in robotics. In their example, the authors suppose a robot with 8 sensors and 2 motors, 1 per side of the robot, although they claim that it can be easily adapted for any kind of robot. To design an obstacle avoidance behaviour, the data received from the proximity sensors is processed and stored in a variable. This data measures the proximity from any nearby obstacle. The motor speed is modified according to the value of this variable in order to avoid the detected obstacles. In this sense, if an object is sensed more with the sensors on the right side, the speed of the right motor should be greater than the speed of the left motor. Consequently, the robot should avoid the collision by turning left. Figure 3.2 depicts the ENPS designed to model this process. The Observe–Decide–Act (*ODA*) loop modelled by the authors is at the core of classic control theory [168]. In addition, the incremental tuning approach of the system variables bears a strong resemblance with Proportional–Integral–Derivative (*PID*) controllers, in which a set of variables is iteratively adjusted to reach stability in the presence of external disturbances [126]. However, in contrast to differential *PID* controllers and due to the modular nature of P systems [41], small changes in the desired behaviour entail proportionally small adjustments in the model.

3.3 Simulation of Enzymatic Numerical P Systems

As mentioned in Section 3.2, SNUPS [157] is a Java simulator for ENPSs. The algorithm implemented by SNUPS to simulate the dynamics of ENPSs is schemed in Algorithm 3.3.1. In contrast to other types of P systems, when it comes to simulation the membrane structure in ENPSs is merely a scaffold. That is to say, ENPS models describe a set of variables intercommunicated with each other by means of programs. Membranes are of certain use to define which pairs of variables are allowed to communicate. However, in the standard notation for ENPS variable names have assigned sub-indexes which depend on their membrane. Therefore, if this notation is employed, it is possible to uniquely identify a variable by its name, without need to explicitly address its membrane.

3.4 A GPU simulator for Enzymatic Numerical P systems

ENPSs present some features which differ from the types of P systems that have been presented so far. These features have a strong influence on the development of parallel simulators for ENPS models:

- ENPSs are deterministic models. Therefore, the repeated simulation of the same ENPS would be of little use, as every simulation would monotonously repeat the same configurations. This is not the case of non-deterministic P systems, as each simulation would reflect a different computation.
- Values associated with membranes are real numbers. Operations on real numbers take more time to compute than on integer numbers. As a matter of fact, the computational power of a computer is measured in *megaflops*, i.e., millions of floating-point operations per second. Moreover, GPUs are optimized to work with floating point numbers.
- Production functions are general arithmetic expressions of unbounded depth with binary operations. Therefore, the most natural approach to compute them is to apply some type of *divide-and-conquer* algorithm. That is to say, to compute a binary operation first compute its operands. This characteristic will display its importance when the GPU simulator

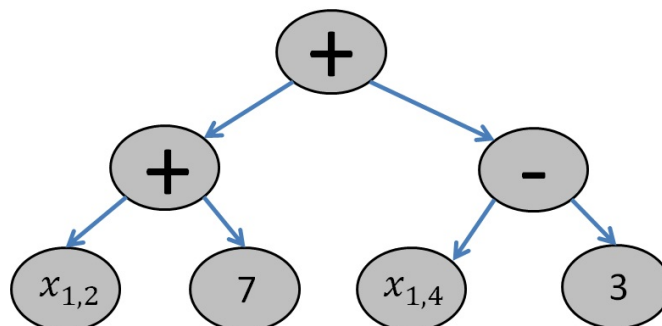
Algorithm 3.3.1 Algorithm for simulation of ENPS models**Input:**

- T : an integer number $t \geq 1$ indicating the iterations of the simulation.
- $\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \dots (Var_m, Pr_m, Var_m(0)))$: ENPS of degree $m \geq 1$.

```

1: for  $t \leftarrow 0$  to  $T$  do
2:   for  $i \leftarrow 1$  to  $m$  do
3:     for  $j \leftarrow 1$  to  $k_i$  do
4:        $x'_{j,i} \leftarrow x_{j,i}(t)$ 
5:     end for
6:   end for
7:   for  $i \leftarrow 1$  to  $m$  do
8:     for  $l \leftarrow 1$  to  $q_i$  do
9:       if  $Pr_i$  is of the form  $Pr_{l,i} = (F_{l,i}(x_{1,i}, \dots, x_{k_i,i}), (e_{l,i} \rightarrow), c_{l,1}|v_1 + \dots + c_{l,n_i}|v_{n_i})$ 
       then
10:         $apply\_program_{l,i} \leftarrow e_{l,i} > \min(x'_{j,i})$  such as  $F_{l,i}(x'_{1,i}, \dots, x'_{k'_i,i})$  consumes
         $x'_{j,i}$ 
11:      else
12:         $apply\_program_{l,i} \leftarrow true$ 
13:      end if
14:      if  $apply\_program_{l,i} = true$  then
15:         $y_{l,i} \leftarrow F_{l,i}(x'_{1,i}, \dots, x'_{k'_i,i})$ 
16:      end if
17:    end for
18:  end for
19:  for  $i \leftarrow 1$  to  $m$  do
20:    for  $l \leftarrow 1$  to  $q_i$  do
21:       $x'_k \leftarrow 0$ , where  $F_{l,i}(x'_{1,i}, \dots, x'_{k'_i,i})$  consumes  $x'_k$ 
22:    end for
23:  end for
24:  for  $i \leftarrow 1$  to  $m$  do
25:    for  $l \leftarrow 1$  to  $q_i$  do
26:      for  $k \leftarrow 1$  to  $n_i$  do
27:        if  $apply\_program_{l,i} = true$  then
28:           $x'_k \leftarrow x'_k + y_{l,i} \cdot \frac{c_{l,k}}{\sum_{g=1}^{n_i} c_{l,g}}$ , where  $(c_{l,k}|x'_k)$  is a pair (constant, variable)
          in the repartition protocol of program  $Pr_{l,i} \wedge c_{l,k} \neq 0$ 
29:        end if
30:      end for
31:    end for
32:  end for
33:  for  $i \leftarrow 1$  to  $m$  do
34:    for  $j \leftarrow 1$  to  $k_i$  do
35:       $x_{j,i}(t+1) \leftarrow x'_{j,i}$ 
36:    end for
37:  end for
38: end for

```

Figure 3.3: Production function expression $(x_{1,2} + 7) + (x_{1,4} - 3)$

is described. Figure 3.3 depicts an example of a production function expression.

The aim of the simulator is to capture the semantics of ENPS models, performing operations in parallel whenever possible. What follows is a hands-on description of the simulator, regarding both the data structures used and the execution of a simulation.

3.4.1 Data structures in the simulator

As it is the norm in GPU computing [38], the data representing the model to simulate is stored by means of arrays, which are:

V (Variables): an array of real numbers of dimension n , where $n \geq 1$ is the total number of variables in the system. Each element V_i , $1 \leq i \leq n$, represents a variable in the system.

PNT (Production Node Types): an array of characters of dimension o , where $o \geq 1$ is the total number of constants, variables and operators in all production functions in the system. Each element PNT_j , $1 \leq j \leq o$, can encode one of the following constant values: **CONSTANT**, **VARIABLE** or any of those: $+$, $-$, $*$, $/$ and \wedge .

PNV (Production Node Values): an array of real numbers of dimension o . Each element PNV_j , $1 \leq j \leq o$, can encode one of the following, depending on the value of PNT_j :

- If $PNT_j = \text{CONSTANT}$, then PNV_j encodes a real constant.

- If $PNT_j = VARIABLE$, then $PNV_j \leq n$ represents a variable in array *variables*.

Otherwise, PNV_j has no particular meaning.

PNLO (Production Node Left Operands): an array of integer numbers of dimension o . If PNT_j is **CONSTANT** or **VARIABLE**, then $PNLO_j$, $1 \leq j \leq o$, has no meaning. Otherwise, $PNLO_j$ addresses the left operand of the operation represented by index j .

PNRO (Production Node Right Operands): an array of integer numbers of dimension o . If PNT_j is **CONSTANT** or **VARIABLE**, then $PNRO_j$, $1 \leq j \leq o$, has no meaning. Otherwise, $PNRO_j$ addresses the right operand of the operation represented by index j .

RPV (Repartition Protocol Variables): a matrix of integer numbers of dimension $s \times q$, where $s \geq 1$ is the total number of programs in the system and $q \geq 1$ is maximum number of pairs (*constant, variable*) in all repartition protocols in the system. Each element $RPV_{l,k} \leq n$, $1 \leq l \leq s, 1 \leq k \leq q$, represents a variable in the system. If the repartition protocol of Program l has fewer than q pairs, then $RPV_{l,k} = -1$.

RPC (Repartition Protocol Constants): a matrix of integer numbers of dimension $s \times q$. Each element $RPC_{l,k}$, $1 \leq l \leq s, 1 \leq k \leq q$, represents a repartition constant. Initially, each value in $RPC_{l,k}$ is an integer, but it can store real numbers. If the repartition protocol of Program l has fewer than q pairs, then $RPC_{l,k} = 0$.

E (Enzymes): an array of integer numbers of dimension s . Each element E_l , $1 \leq l \leq s$, represents the enzyme from Program l . If F_l is of the form $(F_l(x_{1,u}, \dots, x_{k_u,u}), c_{l,1}|v_1 + \dots + c_{l,n_u}|v_{n_u})$, where u is the membrane associated with the production function from Program l , then $e_l = -1$.

In addition to the structures used to represent the system, other arrays are used as well to store temporary data necessary for simulations. These are:

PFR (Production Function Results): an array of real numbers of dimension s which stores the results of the calculations of the computed production functions.

AP (Applicable Program): an array of characters of dimension s which stores the markers to set if programs are applied on the current step of computation. These markers can be *Active* (true) or *Inactive* (false).

3.4.2 Execution of a simulation step

As described in the former subsections, the execution of a simulation step consists of the checking and application of programs for a predefined number of steps. This number of steps, as well as the model to simulate, are specified as inputs to the simulator. In the case that the model simulated defines a number of steps, then this number prevails over the one given as input. This simulation algorithm is described in Algorithm 3.4.1.

In order to simulate ENPS models in parallel, the same algorithm is launched on different *threads*. In CUDA programs, threads are arranged in *blocks*. In these algorithms the number of threads to launch is explicitly indicated by the programmer as a parameter, but not the number of blocks. The reason is that different block sizes (i.e. number of threads per block) might yield different performance results among different graphic cards, so the block size for each algorithm call is omitted. Moreover, unlike blocks, the concept of thread is ubiquitous in parallel computing, so by omitting threads the algorithm can be directly implemented on different parallel platforms such as MPI clusters [161], in which threads do not necessarily need to be bundled in blocks.

Algorithm 3.4.1 Algorithm for parallel simulation of ENPS models

Input:

- T : an integer number greater or equal to 1 indicating the iterations of the simulation.
 - Data structures indicated in Subsection 3.4.1.
- 1: $s \leftarrow$ total number of programs in all membranes
 - 2: $q \leftarrow$ number of pairs (*variable, constant*) in all repartition protocols
 - 3: Normalize repartition protocol constants on $s \times k$ threads (see Algorithm 3.4.2)
 - 4: **for** $t \leftarrow 0$ **to** T **do**
 - 5: Check program applicability on s threads (see Algorithm 3.4.3)
 - 6: Calculate production functions of applicable programs on s threads (see Algorithm 3.4.5)
 - 7: Set to 0 the variables consumed by applicable programs on s threads (see Algorithm 3.4.7)
 - 8: Distribute the results of production functions of applicable programs on q threads (see Algorithm 3.4.8)
 - 9: **end for**
-

3.4.2.1 Repartition protocol normalization

The first stage of the implemented algorithm is known as *repartition protocol normalization*. This stage normalizes the repartition protocol constants of each

repartition protocol, so later the results of production functions are easier to distribute. Program checking launches s threads, where s is the total number of programs of the simulated model. Each thread l , $1 \leq l \leq s$, runs Algorithm 3.4.2.

Algorithm 3.4.2 Repartition protocol normalization

Input:

- RPC : data structure from subsection 3.4.1.
- l, k : two integer numbers $1 \leq l \leq s$, $1 \leq k \leq q$, being $s \geq 1$ the total number of programs in all membranes and i the membrane where the program of the repartition protocol is applied. l and k identify the thread in which the algorithm is applied.

$$1: RPC_{l,k} \leftarrow \frac{RPC_{l,k}}{\sum_{g=1}^q RPC_{l,g}}$$

3.4.2.2 Program checking

The second stage is known as *program checking*, and checks whose programs are applicable by comparing the program's enzyme-like variable (if existent) with the minimum of all variables consumed by its production function. If the program is not of enzymatic form, then it is applied in every step. Like the repartition protocol normalization, the program checking launches s threads, where s is the total number of programs of the simulated model. Each thread l , $1 \leq l \leq s$ performs Algorithm 3.4.3.

Algorithm 3.4.3 Program checking

Input:

- $V, PNT, PNV, PNLO, PNRO, e$ and AP : see Subsection 3.4.1.
- l : an integer number $1 \leq l \leq s$, being $s \geq 1$ the total number of programs in all membranes. l identifies the thread in which the algorithm is applied.

```

1: if  $E_l = -1$  then
2:    $AP \leftarrow true$ 
3: else
4:    $mv \leftarrow$  returned value by call to Algorithm Calculate minimum value
      (see Algorithm 3.4.4) with input parameters  $V, PNT, PNV, PNLO, PNRO$ 
      and  $l$ 
5:    $evar \leftarrow e_l$ 
6:    $eval \leftarrow variable_{ev}$ 
7:    $AP \leftarrow eval > mv$ 
8: end if

```

Algorithm 3.4.4 Calculate minimum value

Input:

- V , PNT , PNV , $PNLO$ and $PNRO$: see Subsection 3.4.1.
- l : an integer number $1 \leq l \leq s$, being $s \geq 1$ the total number of programs in all membranes. l identifies the thread in which the algorithm is applied.

```

1: if  $PNT_l = VARIABLE$  then
2:    $vindex \leftarrow PNV_l$ 
3:    $vval \leftarrow V_{vindex}$ 
4:   return  $vval$ 
5: else
6:   if  $PNT_l = CONSTANT$  then
7:     return  $\infty$ 
8:   else
9:      $lnode \leftarrow PNLO_l$ 
10:     $rnode \leftarrow PNRO_l$ 
11:     $min\_lhs \leftarrow$  returned value by call to Algorithm Calculate minimum value with
        input parameters  $V$ ,  $PNT$ ,  $PNV$ ,  $PNLO$ ,  $PNRO$  and  $lnode$ 
12:     $min\_rhs \leftarrow$  returned value by call to Algorithm Calculate minimum value with
        input parameters  $V$ ,  $PNT$ ,  $PNV$ ,  $PNLO$ ,  $PNRO$  and  $rnode$ 
        return  $\min(min\_lhs, min\_rhs)$ 
13:   end if
14: end if

```

3.4.2.3 Production function computation

The third stage is known as *production function computation*, and computes the production function of applicable programs by recursively traversing them as a binary tree. Production function computation launches s threads as well, where s is the total number of programs of the simulated model. Each thread l , $1 \leq l \leq s$, performs Algorithm 3.4.5.

Algorithm 3.4.5 Production function computation

Input:

- $V, PNT, PNV, PNLO, PNRO, AP$ and PFR : see Subsection 3.4.1.
- l : an integer number $1 \leq l \leq s$, , being $s \geq 1$ the total number of programs in all membranes the total number of programs in all membrane. l identifies the thread in which the algorithm is applied.

```

1: if  $AP_l = true$  then
2:   if  $PNT_l = VARIABLE$  then
3:      $vindex \leftarrow pnv_l$ 
4:      $vval \leftarrow variables_{vindex}$ 
5:      $PFR_l \leftarrow variable\_value$   $\triangleright$  Set the production function result according to its
        type
6:   else
7:     if  $PNT_l = CONSTANT$  then
8:        $PFR_l \leftarrow production\_node\_values_l$ 
9:     else
10:       $lnode \leftarrow PNLO_l$ 
11:       $rnode \leftarrow PNRO_l$ 
12:       $lhsr \leftarrow$  returned value from call to Algorithm Production function
        computation with input parameters  $V, PNT, PNV, PNLO, PNRO, AP,$ 
         $PFR$  and  $lnode$ 
13:       $rhsr \leftarrow$  returned value from call to Algorithm Production function
        computation with input parameters  $V, PNT, PNV, PNLO, PNRO, AP,$ 
         $PFR$  and  $rnode$ 
14:      return returned value by call to Algorithm Apply operation
        (see Algorithm 3.4.6) with input parameters  $lhsr, rhsr$  and  $PNT_l$ 
15:    end if
16:  end if
17: end if

```

3.4.2.4 Variable clearing

The fourth stage is known as *variable clearing*, and sets to 0 all variables consumed by the production function of applicable programs. Like the production function computation, the variable clearing traverses production functions of

Algorithm 3.4.6 Apply operation

Input:

- lhs and rhs : two real values.
 - ot (*operation type*): one of the following: '+', '-', '*', '/', 'pow'.
- 1: **if** $ot = '+'$ **then return** $lhs + rhs$
 - 2: **else if** $ot = '-'$ **then return** $lhs - rhs$
 - 3: **else if** $ot = '*'$ **then return** $lhs \cdot rhs$
 - 4: **else if** $ot = '/'$ **then return** lhs/rhs
 - 5: **else if** $ot = 'pow'$ **then return** lhs^{rhs}
 - 6: **end if**
-

applicable programs, clearing visited variables. Variable clearing launches s threads as well, where s is the total number of programs of the simulated model. Each thread l , $1 \leq l \leq s$ runs Algorithm 3.4.7.

Algorithm 3.4.7 Variable clearing

Input:

- V , PNT , PNV , $PNLO$, $PNRO$ and AP : see Subsection 3.4.1.
 - l : an integer number $1 \leq l \leq s$, being $s \geq 1$ the total number of programs in all membranes. l identifies the thread in which the algorithm is applied.
 - $check$: a logic value (*true* or *false*) which indicates if it is necessary to check program applications.
- 1: **if** $check = false \vee ap_l = true$ **then**
 - 2: **if** $PNT_l = VARIABLE$ **then**
 - 3: $vindex \leftarrow pnv_l$
 - 4: $V_{vindex} \leftarrow 0$
 - 5: **else if** $PNT_l \neq CONSTANT$ **then**
 - 6: $lnode \leftarrow PNLO_l$
 - 7: $rnnode \leftarrow PNRO_l$
 - 8: Call to algorithm *Clear variables* with input parameters V , PNT , PNV , $PNLO$, $PNRO$, AP , $lnode$ and *false*
 - 9: Call to algorithm *Clear variables* with input parameters V , PNT , PNV , $PNLO$, $PNRO$, AP , $rnnode$ and *false*
 - 10: **end if**
 - 11: **end if**
-

3.4.2.5 Results distribution

The last stage of the implemented algorithm is known as *results distribution*, and distributes the results of the production function of applicable programs.

This is arguably the less computationally consuming stage, as it has no recursive calls nor loops. Variable clearing launches $s \times q$ threads, where s is the total number of programs of the simulated model and q is the maximum number of pairs (*variable, constant*) in all repartition protocols in the system. Each thread (l, k) , $1 \leq l \leq s$, $1 \leq k \leq q$, runs Algorithm 3.4.8.

Algorithm 3.4.8 Results distribution

Input:

- V , RPV , RPC , PFR and AP : see Subsection 3.4.1.
- l, k : two integer numbers $1 \leq l \leq s$, $1 \leq k \leq n_i$, being $s \geq 1$ the total number of programs in all membranes and i the membrane where the program of the repartition protocol is applied. l and k identify the thread in which the protocol is applied.

```

1: if  $AP_l = true$  then
2:    $vindex \leftarrow RPV_{l,k}$ 
3:    $V_{vindex} \leftarrow RPC_{l,k} \cdot PFR_l$ 
4: end if

```

3.5 Performance analysis of the GPU simulator

What follows is a performance analysis of the GPU simulator for ENPS models described in this chapter. In addition, set-up operations and considerations about the simulations and simulators used for a fair comparison are discussed. Finally, acceleration factors and elapsed times are displayed, so as to give a graphical overview on the acceleration gained in each case.

3.5.1 Parallel simulator workflow

In order to ease the parallel simulation of ENPS models, the simulator takes an input file describing an ENPS in XML format. The XML format used is the one accepted by SNUPS [157], a previously existent sequential simulator for ENPSs. This way, the reusability of the models is improved, as the same file can be used with independence to the selected simulator, be it SNUPS and on the GPU-based one introduced, without any change in the XML file format.

In order to simulate an ENPS, one needs to encode it on the same XML format as it is required on SNUPS. Once this P system is encoded, the resulting file can be parsed by the GPU simulator. After the parsing process, the simulation is

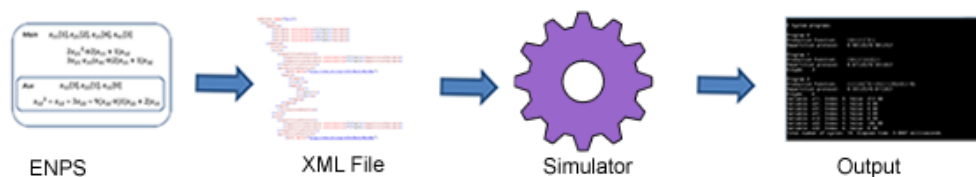


Figure 3.4: Workflow of the simulators

performed. Eventually, the information is displayed on the command prompt. Figure 3.4 represents this process.

3.5.2 Analysis settings

In [73], some problems about an extensive analysis of the simulator are discussed. One of them has to do with the fact that the existing ENPS models have too few programs for parallel simulations to pay off. The reason is that the settings operations computed at the beginning of parallel simulations take a long time, in comparison to the whole sequential computation runtime. In order to overcome this difficulty, some ENPS reference models were replicated several times, thus obtaining models with an on-demand number of programs. The process of *replicating* an ENPS model Π consists on creating new ENPS models with the same structure, variables and programs than Π , but with initial variable values and repartition constants taking random values. Finally, all replicated models are enclosed in a new membrane with no variables nor programs. Algorithm 3.5.1 explains this process. Henceforth, these reference models (one per case study) will be addressed as *seeds*, so the more times the seed models are replicated, the more programs will compose the resulting models. When the number of replications given as input is large enough, the GPU simulation does pay off in terms of execution time.

This replication process has been performed by using Java [6]. For the purposes of parsing the seed model and writing the resulting models, an extension of P-Lingua [71] has been developed. Specifically, a new input and a new output format has been included into the P-Lingua framework. The input format delegates the parsing process to SNUPS [157]. The output format generates an XML description of the model. A sample of this XML code is depicted on Figure 3.5. This description is encoded on the common format accepted by all ENPS simulators (Java, C++ and CUDA/C++). This extension proves the versatility of P-Lingua as a useful, assisting tool for a wide variety of Membrane Computing-related tasks; in this case, for analysing the performance of

Algorithm 3.5.1 Algorithm for model replication of ENPS models**Input:**

- N : an integer $N > 1$ representing the number of copies to perform.
 - $\Pi = (H, \mu, (Var_1, Pr_1, Var_1(0)) \dots (Var_m, Pr_m, Var_m(0)))$: an ENPS (seed) of degree $m \geq 0$ to replicate.
- 1: $\Phi \leftarrow \emptyset$
 - 2: **for** $o = 1$ **to** N **do**
 - 3: $\Pi_o = (H_o, \mu_o, (Var_{1,o}, Pr_{1,o}, Var_{1,o}(0)) \dots (Var_{m,o}, Pr_{m,o}, Var_{m,o}(0)))$, where:
 - $\forall o (1 \leq o \leq N), H_o = \{\{1, o\} \dots \{m, o\}\}$
 - $\forall i, o (1 \leq i \leq m, 1 \leq o \leq N), Var_{i,o} = \{x_{1,i,o} \dots x_{k_i,i,o}\}$
 - $\forall i, o (1 \leq i \leq m, 1 \leq o \leq N), E_{i,o} \subseteq Var_{i,o}$ is the set of all enzyme-like variables associated with programs in $Pr_{i,o}$.
 - $\forall i, o (1 \leq i \leq m, 1 \leq o \leq N), Pr_{i,o} = Pr_{1,i,o} \dots Pr_{q_i,i,o}$, where:
 - $Pr_{l,i,o} = (F_{l,i,o}(x_{1,i,o}, \dots, x_{k_i,i,o}) \rightarrow c_{l,1,o}|v_{o,1} + \dots + c_{l,n_i,o}|v_{o,n_i})$ if $Pr_{l,i}$ is in non-enzymatic form.
 - $Pr_{l,i,o} = (F_{l,i,o}(x_{1,i,o}, \dots, x_{k_i,i,o})(e_{l,i,o} \rightarrow c_{l,1,o}|v_{o,1} + \dots + c_{l,n_i,o}|v_{o,n_i}))$ if $Pr_{l,i}$ is in enzymatic form.
 - $\forall i, o 1 \leq i \leq m, 1 \leq o \leq N, Var_{i,o}(0) = \{\lambda_{1,i,o} \dots \lambda_{k_i,i,o}\}$
 - 4: $\Phi \leftarrow \Phi \cup \{\Pi_o\}$
 - 5: **for** $o = 1$ **to** N **do**
 - 6: **for** $i = 1$ **to** m **do**
 - 7: **for** $j = 1$ **to** k_i **do**
 - 8: $\lambda_{j,i,o} \leftarrow$ a random value $rand \in \{1, \dots, 10\}$
 - 9: **end for**
 - 10: **for** $l = 1$ **to** q_i **do**
 - 11: **for** $j = 1$ **to** n_i **do**
 - 12: $c_{l,j,o} \leftarrow$ a random value $rand \in \{1, \dots, 10\}$
 - 13: **end for**
 - 14: **for each** $ck_{l,i,h,o} \in F_{l,i,o}$ **do**
 - 15: $ck_{l,i,h,o} \leftarrow$ a random value $rand \in \{1, \dots, 10\}$
 - 16: **end for**
 - 17: **end for**
 - 18: **end for**
 - 19: **end for**
 - 20: **end for**
 - 21: **return** $\Pi' = (H', \mu', (Var_{1,1}, E_{1,1}, Pr_{1,1}, Var_{1,1}(0)) \dots (Var_{m,N}, E_{m,N}, Pr_{m,N}, Var_{m,N}(0)), (Var_{skin}, E_{skin}, Pr_{skin}, Var(0)_{skin}))$, where:
 - $H' = \cup_{o=1}^N H_o \cup \{skin\}$
 - $\mu' = [\mu_1, \dots, \mu_N]_{skin}$
 - $Var_{skin} = \emptyset$
 - $E_{skin} = \emptyset$
 - $Pr_{skin} = \emptyset$
 - $Var(0)_{skin} = \emptyset$


```

<membrane name="main">
  <region>
    <memory>
      <variable initialValue="1">x11</variable>
      <variable initialValue="2">x21</variable>
      <variable initialValue="4">x31</variable>
      <variable initialValue="3">x41</variable>
    </memory>
    <rulesList>
      <rule>
        <repartitionProtocol>
          <repartitionVariable contribution="1">x11</repartitionVariable>
          <repartitionVariable contribution="1">x12</repartitionVariable>
        </repartitionProtocol>
        <productionFunction>
          <math xmlns="http://www.w3.org/1998/Math/MathML">
            <apply>
              <times/>
              <cn>2</cn>
              <apply>
                <pow/>
                <ci>x21</ci>
                <cn>2</cn>
              </apply>
            </apply>
          </math>
        </productionFunction>
      </rule>
      <rule>
        <repartitionProtocol>
          <repartitionVariable contribution="2">x11</repartitionVariable>
          <repartitionVariable contribution="1">x32</repartitionVariable>
        </repartitionProtocol>
        <productionFunction>
          <math xmlns="http://www.w3.org/1998/Math/MathML">

```

Figure 3.5: A sample of XML code to define ENPS systems

a GPU simulator.

3.5.3 Performance analysis

All parallel parts of the algorithm are executed with a degree of parallelism at least equal to the number of programs in the simulated model. The degree of parallelism can be even greater when the repartition protocol stage is applied. Hence, a theoretical acceleration of at least the number of programs in the model could be reached, if compared to the runtime of sequential simulators. What follows is an overview of some of the performance analysis conducted to assess the acceleration obtained by the GPU simulator.

As a contribution of this work, two new simulators for ENPS models have been developed on CUDA/C++ and C++, respectively. These simulators will be referred as ENPSCUDA and ENPSC++. Simulation times were compared from ENPSCUDA, ENPSC++ and SNUPS [157]. The purpose in using a C++ simulator is to measure the performance gain against a simulator developed in a low-level programming language. Then, the resulting execution times were compared with the ones obtained by ENPSCUDA, in order to get an approximate speed-up.

On the other hand, SNUPS is an ENPS simulator developed in Java language. Java programs are executed over a middleware between the actual device and the software, known as Java Virtual Machine (*JVM*) [6]. JVM ensures that Java programs can be executed on any device in which JVM is installed, thus guaranteeing complete compatibility among different hardware architectures. However, this virtual machine approach comes at a cost. Firstly, the programmer loses control of the way in which the memory is managed. For instance, memory objects cannot be freed directly. Instead, an execution thread named *garbage collector* checks which objects are not referenced anymore in the program and frees the allocated memory. Secondly, the translations from JVM instructions to assembly instructions are performed in runtime. Thus, an overhead in the execution time is produced as a result of these translations. All in all, the programmer cannot control directly the execution flow of Java programs. Therefore, in cases where efficiency is required, Java is not, in most cases, in the same league as low-level languages such as C++.

3.5.3.1 A C++ sequential simulator

In order to carry out an analysis of the performance and runtimes obtained from ENPSCUDA, ENPSC++ has been developed. The aim of this simulator is to compare simulation times obtained from a sequential, low-level simulator to those from the GPU simulator. Nevertheless, it can also be used for the efficient simulation of Enzymatic Numerical P Systems in those architectures in which no NVIDIA card is available. Besides, this simulator takes as input a file which describes an ENPS in the same format that the ENPSCUDA and SNUPS. Therefore, the same files can be used for all three simulators, hence sparing time on translations between formats.

For a fair comparison between execution times, no memory allocation is performed after the setup stage. This feature is compulsory on ENPSCUDA, because all computation steps are implemented by means of kernel calls and all memory in the GPU can only be allocated from the host code [10]. The

importance of this feature rises from the fact that memory allocation in C++ is a time-consuming instruction. Therefore, if there were a significant number of memory allocations on each computational step the performance of the C++ sequential simulator would be severely hindered. This would result on an even larger speed-up factor due to a bad design of the sequential simulator, instead of a good design of its GPU counterpart.

3.5.3.2 Dummy model

The first seed model is a dummy one with no particular purpose apart from this performance analysis. This model is an ENPS which consist of 2 membranes: $\Pi_1 = (H, \mu, (Var_1, E_1, Pr_{1,1}, Var_1(0)), (Var_2, E_2, Pr_{1,2}, Var_2(0)))$, where:

- $H = \{1, 2\}$
- $\mu = [[]_2]_1$
- $Var_1 = \{x_{1,1}, x_{2,1}, x_{3,1}\}$
- $E_1 = \{x_{3,1}\}$
- $Pr_{1,1} = \{3 \cdot x_{1,1}(x_{3,1} \rightarrow)2|x_{1,1} + 1|x_{2,1}\}$
- $Var_1(0) = \{1, 2, 3\}$
- $Var_2 = \{x_{1,2}\}$
- $E_2 = \emptyset$
- $Pr_{1,2} = \{2 \cdot x_{1,2} \rightarrow 2|x_{1,2}\}$
- $Var_2(0) = \{1\}$

3.5.3.3 Function approximation

On a second case study, the seed model performs a function approximation. Mathematical functions like trigonometric functions, exponential functions, etc. are often used in control algorithms in robotics. Therefore, in the following example an ENPS model which computes e^x is presented. The proposed GPU simulator also allows the computation of rational production functions as well as polynomials, which is an important advantage for the modelling process of complex membrane systems. In order to approximate e^x , the following power series is used:

$$e^x \approx \sum_{n \geq 0} \frac{x^n}{n!} \quad (3.1)$$

The partial sum of this power series is the next sequence:

$$s_n = \sum_{k \geq 0}^n \frac{x^k}{k!} \quad (3.2)$$

Sequence s_n can be written in a recurrent form, as follows:

$$s_n = s_{n-1} + a_n \quad (3.3)$$

where a_n is:

$$a_n = \frac{x^n}{n!} \quad (3.4)$$

Sequence a_n can also be written in a recurrent form, by computing $\frac{a_n}{a_{n-1}}$ as follows:

$$a_n = \frac{x}{n} \cdot a_{n-1} \quad (3.5)$$

Formula 3.5 can be implemented as a rational production function, as shown in Figure 3.6 (rule $Pr_{1,2}$ right).

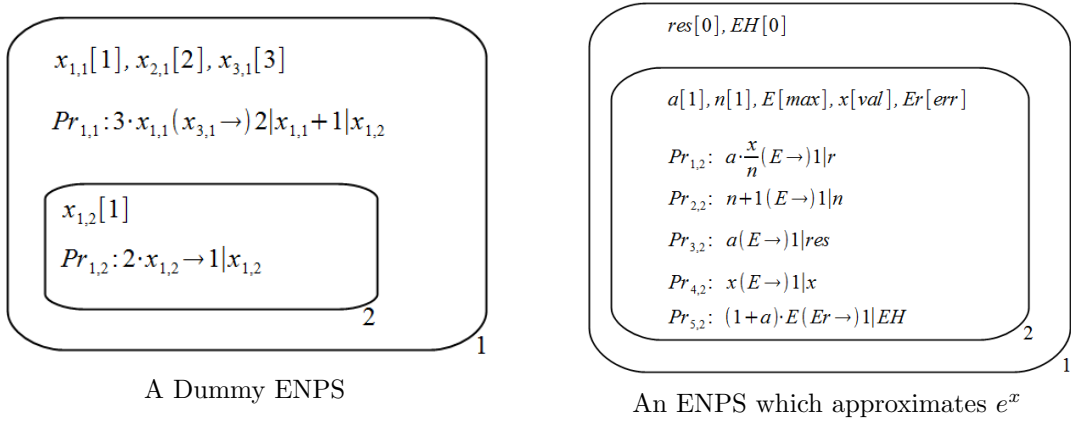


Figure 3.6: Dummy ENPS (left) and ENPS for function approximation (right)

As it is shown in Figure 3.6, two membranes were used in order to approximate the exponential function e^x . The skin membrane (membrane 1) contains

a non enzyme-like variable *res* which represents the result of the computation and an enzyme-like variable, *EH*, which is a stop enzyme. Stop enzymes are used in order to test the halting condition of the computation. Therefore, the number of computational steps is different for different arguments of the function. The computation finishes when the value of the term added to the sum is lower than a given value, *err*.

The child membrane (membrane 2) is responsible for the approximation. It contains the following variables:

- *a* stores the next term of the a_n sequence and has an initial value equal to 1.
- *n* is a counter variable and has the initial value equal to 1.
- *x* represents the argument of the function e^x .
- *E* is an enzyme-like variable which controls the program flow. It allows the execution of the valid production functions and it is consumed when the computation finishes; the initial value of the enzyme is given as input *max*; *max* must be a value greater than the maximum possible value of *x*.
- *Er* is an enzyme-like variable used in the halting condition.

Er receives an input value, $err = 10^{-10}$; when the term of the series is lower than *err*, the computation stops.

Membrane 2 has five rules responsible for the following tasks:

- $Pr_{1,2}$ computes the next term in the series; if the value of *E* is greater than *a*, *x* or *n*, the rule is active.
- $Pr_{2,2}$ produces the incrementation of *n*.
- $Pr_{3,2}$ accumulates the terms in variable *res*, which will be the final result.
- $Pr_{4,2}$ copies the value of *x*, which was consumed and must be stored.
- $Pr_{5,2}$ causes the computation to halt when $a < Er$.

The value of *a* is decreasing because the sequence a_n is convergent and decreases as *n* grows. When $Pr_{5,2}$ is activated, the stop enzyme *EH* receives a positive value and *E* is consumed, so the other production functions become inactive. A condition outside the membrane system tests if the stop enzyme,

EH , is greater than 0 and if that happens, the simulation halts.

It is important to highlight that the model obtained from replicating this exponential approximation model might not approximate e^x . Nevertheless, the purpose of this work is the development of a GPU simulator for ENPS models, not the modelling of numerical functions within that framework, and the replication process has been provided merely to obtain ENPS models with a sufficiently large number of membranes.

3.5.4 Acceleration analysis

The graphic card used is a domestic, commercial one. Thus, it is not designed for intensive, parallel computations. In contrast, Tesla models contain more RAM memory and a larger number of processors, as they are specifically engineered for intensive High Performance Computing [10]. For instance, NVIDIA Tesla C1060 graphic cards contain 240 streaming multiprocessors and 4 GB RAM memory [10]. Thus, the speed-up factors obtained on one of these cards is expected to be larger than in the ones obtained in this study.

The models have been simulated on an *NVIDIA GeForce GTX 460M* card whose technical characteristics are described in Chapter 6, Section 2.5. This model supports the new Fermi technology. Fermi cards allow programmers to make use of new features impossible (or at least very hard) for previous models. Some examples of these features are the computation of recursive functions and atomic operations with float-type numbers [10]. The impact of these features on the simulator code is rather important. Thus, by employing these features, the development process is eased and the simulator code is much clearer. For instance, in order to calculate production functions in programs, recursion comes as a straightforward approach. That is because these general mathematical expressions can be easily represented as tree-like structures, which are usually traversed by using recursive algorithms. Another important brand-new feature on Fermi technology is the use of atomic operations on floating-point numbers. In order to add up the contributions from repartition protocols, these instructions have been used, as the values of contributed variables can be modified by different CUDA threads. However, the use of these features comes at a cost. Specifically, ENPSCUDA can only be run on Fermi NVIDIA cards, as previous models do not support these features. An improvement on the code in order to add compatibility for previous graphic cards is thus left as a future work.

For each seed, a total of 36 models have been simulated. The number of programs range from 1 to 120000. Each model has been run for 100 steps with

a block size of 256 threads per block for each kernel, as it was the one which gave the best results on the device of choice. Figure 3.7 displays the execution times and acceleration factors obtained when simulating the dummy and function approximation models. The reason to ignore SNUPS in some charts is to display cleaner statistics on the execution times and speed-up factors obtained from the most efficient studied simulators.

By examining the dummy model charts, one can observe that there is a large difference between the execution times from SNUPS and ENPSC++/ENPSCUDA. Thus, a maximum speed-up factor of about 90x is reached on the comparison between ENPSCUDA and SNUPS, taking place in the interval between 4000 and 8000 programs per model. However, the maximum speed-up factor obtained between ENPSCUDA and ENPSC++ is only 6.5x. The maximum speed-up factor on the simulation of the e^x model is about 49x on the SNUPS vs ENPSCUDA comparison and about 10x on the ENPSC++ vs ENPSCUDA comparison.

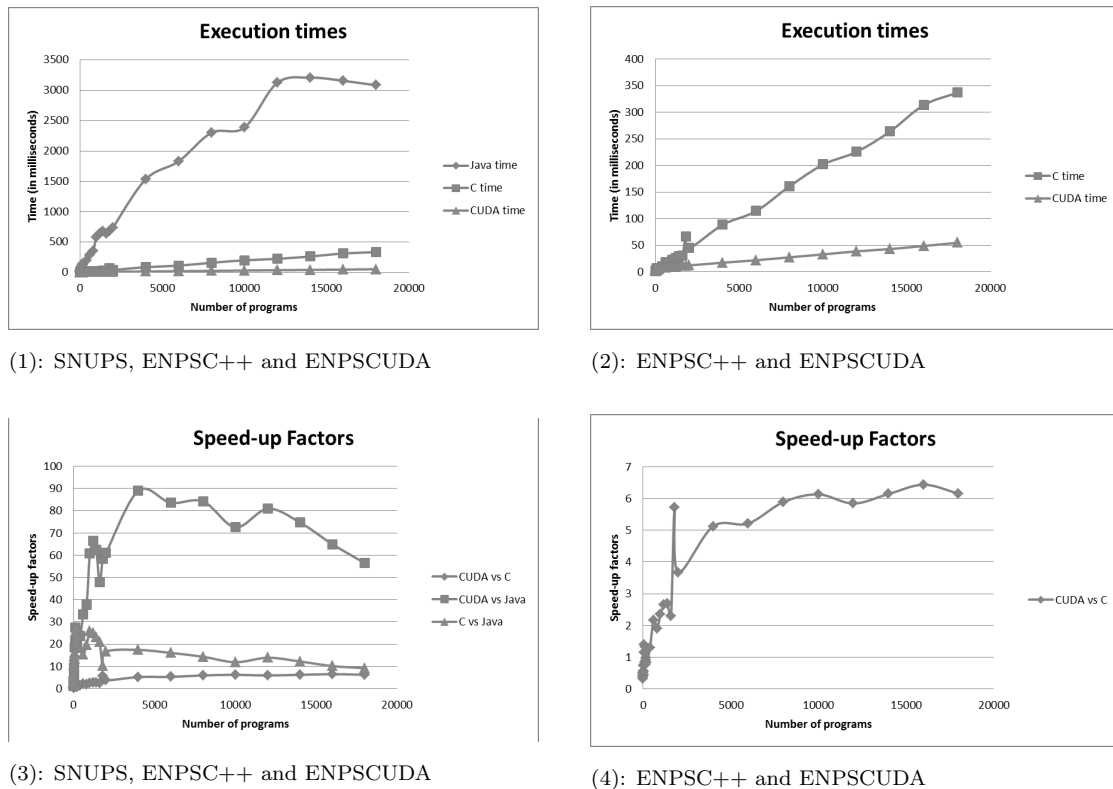
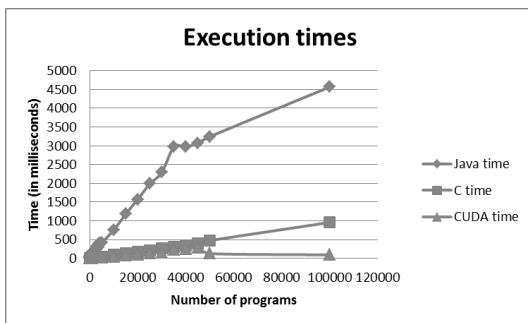
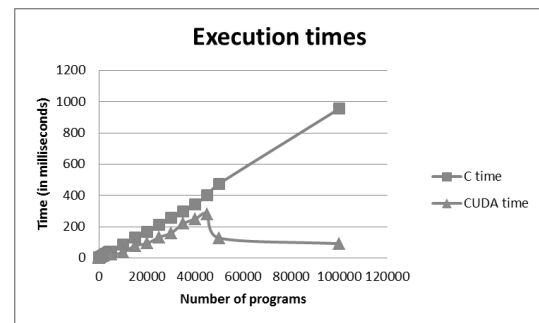


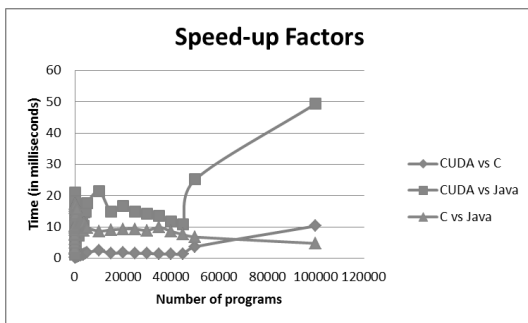
Figure 3.7: Execution times and speed-up factors for **dummy** model



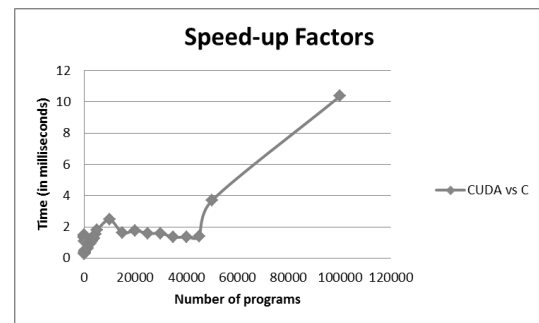
(5): SNUPS, ENPSC++ and ENPSCUDA



(6): ENPSC++ and ENPSCUDA



(7): SNUPS, ENPSC++ and ENPSCUDA



(8): ENPSC++ and ENPSCUDA

Figure 3.8: Execution times and speed-up factors for **function approximation** model

Chapter 4

Logic Network Dynamic P systems

Since its inception, Membrane Computing has been an useful modelling framework for biochemical phenomena. As a natural evolution of this research field, genetic networks have also been modelled by means of P systems. In this chapter, a Membrane Computing model for a specific type of gene networks known as Logic Networks, is discussed. This model is designed within the framework of *Population Dynamics P* (PDP) systems, which has been successfully applied on the modelling of ecosystems and is also introduced in this chapter. The aim of this model is to reproduce the dynamics of Logic Networks by emulating the behaviour of the improved *Logic Analysis of Phylogenetic Profiles* (LAPP) method, which is an algorithm specifically designed to capture the dynamics of these networks.

This chapter is structured as follows. Section 4.1 overviews some Membrane Computing models on gene regulatory networks. Section 4.2 formalizes the concept of logic networks. Section 4.3 is devoted to present Population Dynamics P systems and their application as a modelling framework for Logic Networks. Finally, Section 4.4 describes a Membrane Computing model based on the improved LAPP method for the simulation of Logic Network dynamics.

4.1 Some antecedents of Gene Network models in Membrane Computing

Since its introduction by Gheorghe Păun [190], Membrane Computing has been applied as a modelling framework for biological phenomena at a microscopical

level. One of its main features is its capability to model different compartments by means of *membranes* interconnected by a hierarchical structure. The idea is that reactions may differ according to the compartment in which they occur. Some traditional approaches such as Ordinary Differential Equations (*ODEs*) already enabled this feature. For instance, Kawai [116] proposed a multidimensional stochastic ODE system. This system describes the evolution of the concentration of chemical drugs inside biological tissues such as liver, guts and muscles. This concentration varies because of the decay of the drug molecules, as well as the inflow of the drug substance among the tissues. In this system, Kawai models each one of the different tissues as compartments. The concentration of the modelled drug inside each compartment is represented by a variable, which is modified on the stochastic evolution of the ODE system.

Although ODEs are a well-known framework for biomolecular systems, they require some assumptions on the system to be modelled. Specifically, they require the differential in the concentration of substances within each compartment to be constant. In addition, their accuracy fails when the number of molecules taken into account is too small or the reactions are not fast. This is due to their continuous nature, that is, the number of molecules in the system is approximated to a real number. This approximation works well when the number of molecules is large (a sufficiently large number of molecules is to be at least thousands of them), but it does not appropriately reflect reality on scenarios which consider only a few molecules. A different approach from the field of Membrane Computing can help sort out these constraints. In contrast to ODEs, P systems do not need to make these assumptions. That is, they faithfully reproduce scenarios with few molecules and non-constant concentration differentials. There also exists another advantage of P systems over ODEs known as *modularity* of the system. A system is considered to be *modular* if a small change in the behaviour of the modelled system, usually entails a relatively small change in the model, whilst a slight modification in the behaviour of ODEs usually requires a complete restructuration [41, 199]. Cheruku *et al.* demonstrated this property by simulating FAS-induced apoptosis by using P systems [41].

The Gillespie algorithm is a well-known Monte Carlo algorithm for the stochastic simulation of molecular interactions taking place inside a compartment, a well-mixed and fixed volume. This algorithm takes into account that all reactions do not occur at the same time. That is, there is not a global clock to synchronize the reactions of every substance. In contrast, substances react in a stochastic manner. Specifically, reaction probabilities are calculated as a function over their concentrations. This well-tested algorithm is the core engine

of *Infobiotics* [23], a software tool for systems biology within the framework of Membrane Computing. This software simulates biochemical processes by mapping them into P systems. Simulations are carried out by implementing a multicompartimental version of the Gillespie algorithm. These biochemical systems range from simple reaction systems to more complicated and structured ones, such as Gene Regulatory Networks (*GRNs*).

Informally speaking, GRNs are directed graphs in which vertices and edges represent genes and interactions, respectively. The dynamics of these networks are heavily influenced by the fluctuations in the concentrations of the biochemical substances interacting with the genes, such as proteins. These fluctuations have been especially studied within the field of Membrane Computing, in order to understand the evolution of GRNs. For instance, Hinze *et al.* [102] proposed a P system model for GRNs. In this model, the timing of biochemical reactions is modelled by Hill kinetics, which formulate the intensity of gene interactions by means of continuous, sigmoid-shaped threshold functions. These functions quantify the production rate of gene products. By including Hill kinetics into the field of Membrane Computing, they obtain a new framework known as *Hill P systems*. Hill P systems combine the discrete nature of P system rules and the continuous dynamics of Hill kinetics functions to regulate the application of these rules.

The idea of mixing discrete Membrane Computing rules and continuous functions to regulate their application is relatively common. For instance, Profir *et al.* [184] proposed another Membrane Computing model of GRNs. Their model uses the properties of P systems to reflect the discrete aspects of gene regulation, such as the interaction between DNA and other biomolecules, while describing the internal state of each cell in a continuous form. The model does not describe rules for direct interaction between genes, but between DNA and another biomolecules. This way, gene interaction is described in two steps: an interaction between the first gene and a biomolecule, and an interaction between that biomolecule and the second gene. Gene interactions are synchronized by using Linear Temporal Logic constructs, giving place to P transducers [45].

4.2 Logic Networks

In this section, the concept of Logic Network is discussed. Informally speaking, a Logic Network is a directed graph in which nodes are influenced by other nodes either directly or by boolean operations over them. In this section, a

formalization of Logic Networks, its dynamics and a method for constructing logic networks out of raw data are presented, so as to introduce the model which will be later conceptualised under the paradigm of Membrane Computing.

4.2.1 Formalization of Logic Networks

In this chapter, Gene Regulatory Networks are considered as boolean networks in which, at any instant, genes can be either active or inactive. What follows is a formal definition of a Logic Network (LN), including its syntax and semantics.

4.2.1.1 Syntax of Logic Networks

An *alphabet* is a non-empty set. Given a finite alphabet Γ , we denote $\bar{\Gamma} = \{\bar{x} : x \in \Gamma\}$ and $\bar{x} = x, x \in \Gamma \cup \bar{\Gamma}$, where $\Gamma \cap \bar{\Gamma} = \emptyset$.

Definition 4.1. A gene g over a finite alphabet Γ is an element in Γ . The behaviour of g is a mapping φ_g from \mathbb{N} into $\{0, 1\}$. The state of g at any instant $t \in \mathbb{N}$ is $\varphi_g(t)$. If $\varphi_g(t) = 1$ (respectively, $\varphi_g(t) = 0$) it is said that gene g is active (respectively, inactive) at instant t .

Given a finite alphabet Γ and a gene g over Γ the application $\varphi_{\bar{g}}$ is defined as follows: $\varphi_{\bar{g}} = 1 - \varphi_g$, that is, for each $t \in \mathbb{N}$, $\varphi_{\bar{g}}(t) = 1 - \varphi_g(t)$. It is important to highlight that if g is a gene over Γ , then \bar{g} is **not** a gene over Γ . For each alphabet Γ the mapping l_Γ from $\Gamma \cup \bar{\Gamma}$ into $\{0, 1\}$ is defined as follows: $l_\Gamma(x) = 1$, if $x \in \Gamma$, and $l_\Gamma(x) = 0$, otherwise. That is to say, $l_\Gamma(x) = 1$ means that x is a gene over Γ .

Definition 4.2. A Logic Network of size $n \geq 1$ over an alphabet Γ such that $|\Gamma| \geq n$, is a tuple (Γ, f_1, f_2) where:

1. $|\Gamma| = n$ (Γ is the set of genes of the network).
2. $f_1 = (f_1^j, \dots, f_1^{\alpha_1})$ such that for each $j, 1 \leq j \leq \alpha_1, f_1^j = (g_1^{j,1}, g_1^{j,2}, \omega_1^j, op_1^j)$, where:
 - $g_1^{j,1}, g_1^{j,2} \in \Gamma$.
 - ω_1^j is a real number in $[0, 1]$ which represents the certainty of unary interaction f_1^j (see [229], noting that ω_1^j is equivalent to $U(B|A)$, with $B = g_1^{j,2}$ and $A = g_1^{j,1}$).
 - op_1^j is a mapping from \mathbb{N} into $\{-1, 0, 1\}$ which can be of one of the following types: sp^j, si^j, wp^j, wi^j . These functions are defined as follows: for each $t \in \mathbb{N}$:

- $sp^j(t) = \varphi_{g_1^{j,1}}(t) - \varphi_{\bar{g}_1^{j,1}}(t)$ (strong promotion)
- $si^j(t) = -sp_1^j(t)$ (strong inhibition)
- $wp^j(t) = \varphi_{g_1^{j,1}}(t)$ (weak promotion)
- $wi^j(t) = -wp_1^j(t)$ (weak inhibition)

These operations provide the contribution from f_1^j to gene $g_1^{j,2}$ in conjunction with ω_1^j in order to know its state at instant $t + 1$.

3. $f_2 = (f_2^j, \dots, f_2^{\alpha_2})$ such that for each $j, 1 \leq j \leq \alpha_2$, $f_2^j = (g_2^{j,1}, g_2^{j,2}, g_2^{j,3}, \omega_2^j, op_2^j)$, where:

- $g_2^{j,1}, g_2^{j,2}, g_2^{j,3} \in \Gamma \cup \bar{\Gamma}$.
- ω_2^j is a real number in $[0, 1]$ which represents the certainty of binary interaction f_2^j (see [229], noting that ω_2^j is equivalent to $U(C|f(A, B))$, with $C = g_2^{j,3}$, $A = g_2^{j,1}$, $B = g_2^{j,2}$ and $f = f_2^j$).
- op_2^j is a mapping from \mathbb{N} into $\{-1, 1\}$ which can be of one of the following types: and^j, or^j, xor^j . These functions are defined as follows: for each $t \in \mathbb{N}$,

$$and^j(t) = [\varphi_{g_2^{j,1}}(t) \cdot \varphi_{g_2^{j,2}}(t) - \overline{\varphi_{g_2^{j,1}}(t) \cdot \varphi_{g_2^{j,2}}(t)}] \cdot (2 \cdot l_\Sigma(g_2^{j,3}) - 1)$$

$$or^j(t) = \frac{[\varphi_{g_2^{j,1}}(t) + \varphi_{g_2^{j,2}}(t) - \varphi_{g_2^{j,1}}(t) \cdot \varphi_{g_2^{j,2}}(t) - \overline{\varphi_{g_2^{j,1}}(t) + \varphi_{g_2^{j,2}}(t) - \varphi_{g_2^{j,1}}(t) \cdot \varphi_{g_2^{j,2}}(t)}]}{\varphi_{g_2^{j,1}}(t) + \varphi_{g_2^{j,2}}(t) - \varphi_{g_2^{j,1}}(t) \cdot \varphi_{g_2^{j,2}}(t)} \cdot (2 \cdot l_\Sigma(g_2^{j,3}) - 1)$$

$$xor^j(t) = \frac{[(1 - \varphi_{g_2^{j,1}}(t)) \cdot \varphi_{g_2^{j,2}}(t) + (1 - \varphi_{g_2^{j,2}}(t)) \cdot \varphi_{g_2^{j,1}}(t) - \overline{(1 - \varphi_{g_2^{j,1}}(t)) \cdot \varphi_{g_2^{j,2}}(t) + (1 - \varphi_{g_2^{j,2}}(t)) \cdot \varphi_{g_2^{j,1}}(t)}]}{(1 - \varphi_{g_2^{j,1}}(t)) \cdot \varphi_{g_2^{j,2}}(t) + (1 - \varphi_{g_2^{j,2}}(t)) \cdot \varphi_{g_2^{j,1}}(t)} \cdot (2 \cdot l_\Sigma(g_2^{j,3}) - 1)$$

where \bar{b} denotes $1 - b$, for each $b \in \{0, 1\}$.

These operations provide the contribution from f_2^j to gene $g_2^{j,3}$ in conjunction with ω_2^j in order to know its state at instant $t + 1$. The development of these formulae is described as follows. Let us start with the $and_2^j(t)$ operation over genes $G_2^{j,1}(t)$ and $G_2^{j,2}(t)$. The idea is that it behaves like a logic *and* gate, in which both the inputs and the output can be negated. The contribution to gene $g_2^{j,3}$ is positive if $G_2^{j,1}(t) = 1 \wedge G_2^{j,2}(t) = 1$ and $l(G_2^{j,3}) = 1$, or $G_2^{j,1}(t) = 0 \vee G_2^{j,2}(t) = 0$ and $l(G_2^{j,3}) = 0$, and is negative otherwise. If $G_2^{j,1}(t) = 1 \wedge G_2^{j,2}(t) = 1$, then $G_2^{j,1}(t) \cdot G_2^{j,2}(t) = 1$ and $\overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)} = 0$. Likewise, if $G_2^{j,1}(t) = 0 \vee G_2^{j,2}(t) = 0$, then $G_2^{j,1}(t) \cdot G_2^{j,2}(t) = 0$ and $\overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)} = 1$.

Therefore, let us suppose that $G_2^{j,1}(t) = 1 \wedge G_2^{j,2}(t) = 1$. Then:

$$\begin{aligned} & [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] = 1 \\ & \overline{[G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}]} - G_2^{j,1}(t) \cdot G_2^{j,2}(t) = \\ & -1 \cdot [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] = -1 \end{aligned}$$

If $l(G_2^{j,3}) = 1$, then

$$\begin{aligned} & [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) = 1 \\ & 1 - l(G_2^{j,3}) = 0 \end{aligned}$$

Therefore,

$$\begin{aligned} & [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) \\ & - [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) = 1 \end{aligned}$$

However, if $l(G_2^{j,3}) = 0$, then

$$\begin{aligned} & [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) \\ & - [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) = \\ & - [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) = -1 \end{aligned}$$

Likewise, if $G_2^{j,1}(t) = 0 \vee G_2^{j,2}(t) = 0$ and $l(G_2^{j,3}) = 1$ then

$$\begin{aligned} & [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) \\ & - [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) = -1 \end{aligned}$$

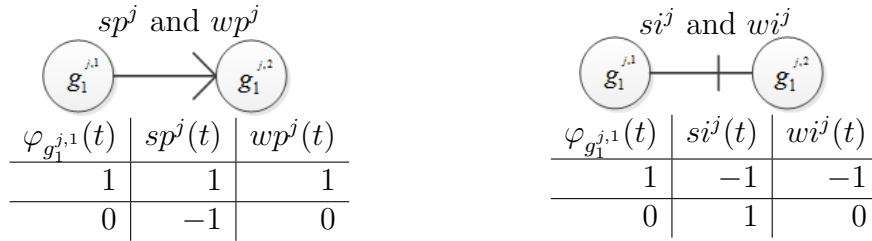
(1 for $l(G_2^{j,3}) = 0$). By simplification, we obtain the following expression:

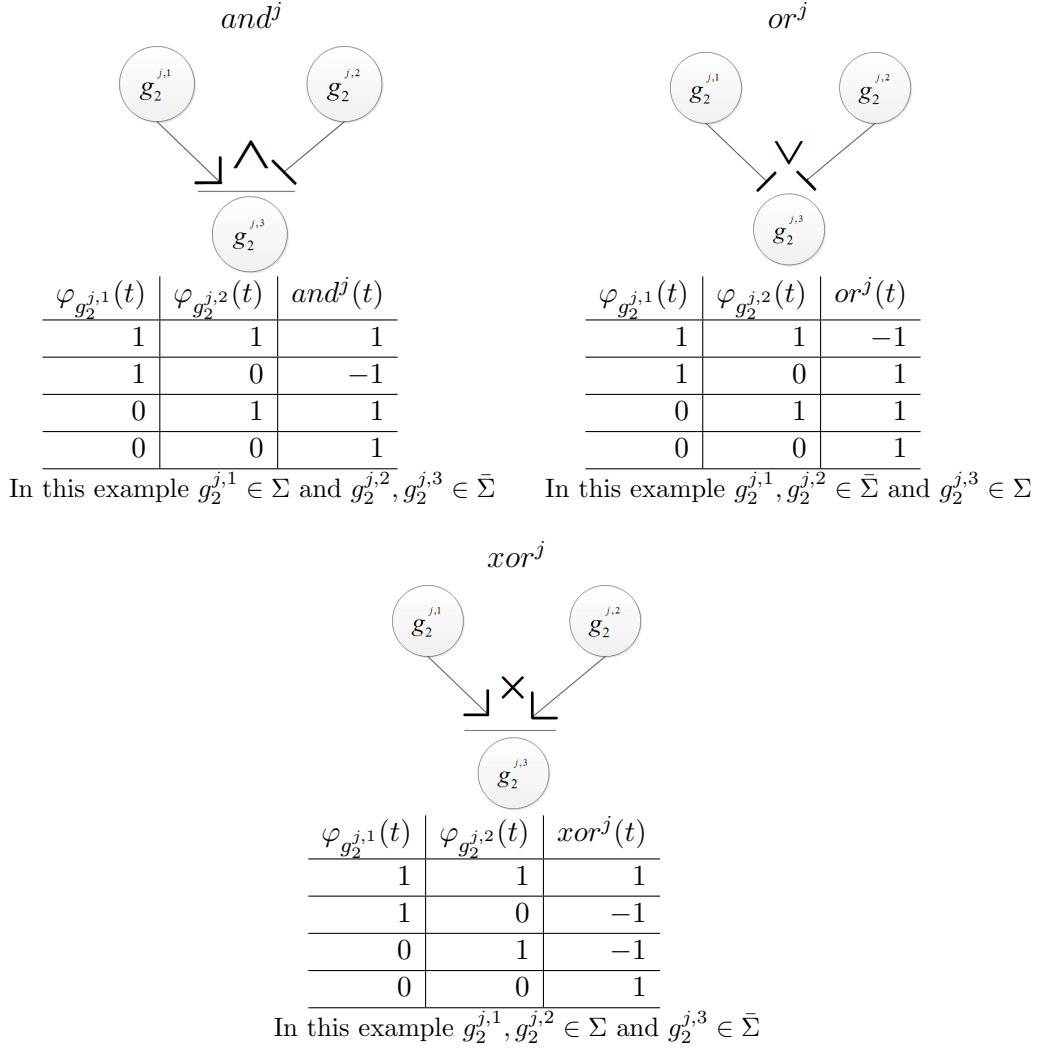
$$\begin{aligned} \text{and}_2^j(t) &= [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) \\ &\quad - [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) \\ &= [G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \overline{G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (2 \cdot l(G_2^{j,3}) - 1) \end{aligned}$$

The same reasoning is applied to deduce the formulae for operations $or_2^j(t)$ and $xor_2^j(t)$:

$$\begin{aligned}
or_2^j(t) &= [G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \\
&\quad \overline{G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot l(G_2^{j,3}) \\
&\quad - [G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \\
&\quad \overline{G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (1 - l(G_2^{j,3})) \\
&= [G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t) - \\
&\quad \overline{G_2^{j,1}(t) + G_2^{j,2}(t) - G_2^{j,1}(t) \cdot G_2^{j,2}(t)}] \cdot (2 \cdot l(G_2^{j,3}) - 1) \\
xor_2^j(t) &= [(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t) - \\
&\quad \overline{(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t)}] \cdot l(G_2^{j,3}) \\
&\quad - [(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t) \\
&\quad \overline{(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t)}] \cdot (1 - l(G_2^{j,3})) \\
&= [(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t) \\
&\quad \overline{(1 - G_2^{j,1}(t)) \cdot G_2^{j,2}(t) + (1 - G_2^{j,2}(t)) \cdot G_2^{j,1}(t)}] \cdot (2 \cdot l(G_2^{j,3}) - 1)
\end{aligned}$$

Next, operations f_1^j and f_2^j are informally described in Figures 4.1 and 4.2, respectively. Let us point out that, in graphic representations of operations f_1^j , only genes in Γ appear. Likewise, the membership of $g \in \Gamma$ (respectively, $g \in \bar{\Gamma}$) is translated into the arrow-type operation \rightarrow (respectively, \dashv). If $g_2^{j,3} \in \bar{\Gamma}$, $\bar{\cdot}$ is denoted upon Gene $g_2^{j,3}$.

Figure 4.1: Behaviour of unary operations f_1^j

Figure 4.2: Behaviour of binary operations f_2^j

4.2.1.2 Semantics of logic networks

Next, a semantics for Logic Networks is introduced. Let $\Pi = (\Gamma, f_1, f_2)$ be a Logic Network of size $n \geq 1$ over an alphabet $\Gamma = \{g_1, \dots, g_n\}$ with n nodes (genes) according to Definition 4.2. A *configuration* of the Logic Network Π at instant t is a tuple $(\varphi_{g_1}(t), \dots, \varphi_{g_n}(t))$ which describes the state of every gene g_i at that instant.

In order to define a *transition step* from t to $t + 1$ in a Logic Network Π ,

the value $\varphi_{g_i}(t+1)$, $1 \leq i \leq n$, is computed from the configuration of Π at any instant t . For such a purpose, some concepts and notations are discussed.

- Let $f_1^j = (g_1^{j,1}, g_1^{j,2}, \omega_1^j, op_1^j)$ be a unary operation by which node $g_1^{j,1}$ acts on node $g_1^{j,2}$. In this case, $g_1^{j,1}$ and $g_1^{j,2}$ are genes. The action of $g_1^{j,1}$ on $g_1^{j,2}$ at instant t , denoted by $action(g_1^{j,2}|g_1^{j,1})(t)$, is defined as follows:

$$action(g_1^{j,2}|g_1^{j,1})(t) = op_1^j(t) \cdot \omega_1^j$$

- Let $f_2^j = (g_2^{j,1}, g_2^{j,2}, g_2^{j,3}, \omega_2^j, op_2^j)$ be a binary operation by which nodes $g_2^{j,1}$ and $g_2^{j,2}$ act on node $g_2^{j,3}$. It is noteworthy that, in the case of binary operations, $g_2^{j,k}$, $1 \leq k \leq 3$, is whether a gene or, otherwise, $\bar{g}_2^{j,k}$ is a gene. In order to compute the contribution to the state of the gene associated with an instant $t+1$, the action of $g_2^{j,1}$ and $g_2^{j,2}$ on $g_2^{j,3}$ at instant t , is denoted by $action(g_2^{j,3}|g_2^{j,1}, g_2^{j,2})(t)$, as follows:

$$action(g_2^{j,3}|g_2^{j,1}, g_2^{j,2})(t) = op_2^j(t) \cdot \omega_2^j$$

- Based on the aforesaid definitions, the total effect of the action on gene i is defined as follows:

$$\begin{aligned} Action(g_i, t) &= Action_1(g_i, t) + Action_2(g_i, t), \text{ being} \\ Action_1(g_i, t) &= \sum_{\substack{1 \leq j \leq \alpha_1 \\ g_1^{j,2} = g_i}} action(g_1^{j,2}|g_1^{j,1})(t) \\ Action_2(g_i, t) &= \sum_{\substack{1 \leq j \leq \alpha_2 \\ g_2^{j,3} = g_i}} action(g_2^{j,3}|g_2^{j,1}, g_2^{j,2})(t) \end{aligned}$$

Then, $g_i(t+1)$ is defined as follows:

$$\varphi_{g_i}(t+1) = \begin{cases} 1, & \text{if } \varphi_{g_i}(t) + Action_1(g_i, t) + Action_2(g_i, t) \geq 0.5, \\ 0, & \text{otherwise} \end{cases}$$

This manner, given a configuration $\mathcal{C}_t = \{\varphi_{g_1}(t), \dots, \varphi_{g_n}(t)\}$ at any instant t we can compute the configuration $\mathcal{C}_{t+1} = \{\varphi_{g_1}(t+1), \dots, \varphi_{g_n}(t+1)\}$ at instant $t+1$, taking one transition step.

Considering this concept of Logic Network and its associated dynamics, a model within the framework of Membrane Computing to reproduce the behaviour of these networks is presented in Section 4.4, providing a formalization and defining its semantics.

4.2.2 Construction of Logic Networks

In this subsection, a method for constructing logic networks out of raw data known as *Logic Analysis of Phylogenetic Profiles* (*LAPP*, for short) is presented. LAPP [25] is an approach to identify logic interactions among a set of elements (e.g., genes, proteins) in uncertain scenarios. This approach is based on the expression profiles of these elements. Given m_0 samples, the expression profiles of element A are $(v_1, v_2, \dots, v_{m_0})$, where v_i , $1 \leq i \leq m_0$ is the expression value of element A in the i^{th} sample. Unary interactions involve two elements (three for binary). In order to illustrate this concept, two types of unary interactions are considered (strong promotion and strong inhibition), as well as a total of ten types of binary interactions based on *AND*-like, *OR*-like and *XOR*-like logic gates combining negated inputs and outputs.

The existence likelihood of unary interactions f_1^i , $1 \leq i \leq 2$ from A to B can be computed as described in Formula 4.1.

$$U(B|f_1^i(A)) = \frac{H(f_1^i(A)) + H(B) - H'(f_1^i(A), B)}{H(B)}, \quad (4.1)$$

where:

- $f_1^i(A) = (f_1^i(v_1), \dots, f_1^i(v_{m_0}))$ is the result of the expression profile of A , *i.e.* $(v_1, v_2, \dots, v_{m_0})$ under the reaction of f_1^i .
- $H(B) = -\sum_{t \in \{0,1\}} p_t \log(p_t)$ is the entropy of the expression profile of B , *i.e.* $(v'_1, v'_2, \dots, v'_{m_0})$, where p_t is the number of element t in $(v'_1, v'_2, \dots, v'_{m_0})$.
- $H'(f_1^i(A), B) = -\sum_{(t_1, t_2) \in \Lambda} p_{(t_1, t_2)} \log(p_{(t_1, t_2)})$ is the joint entropy of $f_1^i(A)$ and the expression profile of B , where $\Lambda = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$, $p_{(t_1, t_2)}$ is the number of element (t_1, t_2) in $((f_1^i(v_1), v'_1), \dots, (f_1^i(v_{m_0}), v'_{m_0}))$.

In addition, we consider $U(B|A) = \max\{U(B|f_1^1(A)), U(B|f_1^2(A))\}$ as the 1-order U from A into B .

Similarly, the existence likelihood of binary interactions f_2^j ($j \in \{1, 2, \dots, 10\}$) from A and B to C can be computed as described in Formula 4.2.

$$U(C|f_2^j(A, B)) = \frac{H(f_2^j(A, B)) + H(C) - H'(f_2^j(A, B), C)}{H(C)}, \quad (4.2)$$

where:

- $f_2^j(A, B) = (f_2^j(v_1, v'_1), \dots, f_2^j(v_{m_0}, v'_{m_0}))$ is the result of the expression profile of A and B , i.e. $(v_1, v_2, \dots, v_{m_0})$ and $(v'_1, v'_2, \dots, v'_{m_0})$ under the reaction of f_2^j .
- $H(C) = -\sum_{t \in \{0,1\}} p_t \log(p_t)$ is the entropy of the expression profile of C , i.e. $(v''_1, v''_2, \dots, v''_{m_0})$, where p_t is the number of element t in $(v''_1, v''_2, \dots, v''_{m_0})$.
- $H'(f_2^j(A, B), C) = -\sum_{(t_1, t_2) \in \Lambda} p_{(t_1, t_2)} \log(p_{(t_1, t_2)})$ is the joint entropy of $f_2^j(A, B)$ and the expression profile of C , where $\Lambda = \{(0, 0), (0, 1), (1, 0), (1, 1)\}$, $p_{(t_1, t_2)}$ is the number of element (t_1, t_2) in $((f_2^j(v_1, v'_1), v''_1), \dots, (f_2^j(v_{m_0}, v'_{m_0}), v''_{m_0}))$.

Similarly to the case of $U(A|B)$,

$$U(C|A, B) = \max\{U(C|f_2^1(A, B)), \dots, U(C|f_2^{10}(A, B))\}$$

is called the 2-order U from A and B into C . u_1 and u_2 are set to the threshold values of unary and binary interactions, respectively. Then, those which satisfy existing conditions are obtained. Unary interactions only exist between two genes with greater difference of 1-order U s in two opposite directions. The direction of $U(B|A)$ is from A to B . The 1-order U with the opposite direction to that of $U(B|A)$ is $U(A|B)$, and its direction is from B to A . The relative difference of $U(B|A)$ and $U(A|B)$ is defined as

$$Differ(U(B|A), U(A|B)) = \frac{|U(A|B) - U(B|A)|}{\frac{1}{2}(U(A|B) + U(B|A))}$$

Let the relationship between $U(B|A)$ and $U(A|B)$ be $U(B|A) \geq U(A|B)$ and the threshold of relative difference be δ_0 . If $Differ(U(B|A), U(A|B)) > \delta_0$, then the relationship from A to B may exist. If $U(B|A) > u_1$, then the unary interaction from A to B is f_1^i . The existing conditions of unary interactions are listed as follows:

$$\begin{aligned} U(B|A) &= U(B|f_1^i(A)) \\ U(B|A) &> U(A|B) \\ \delta &= \frac{|U(A|B) - U(B|A)|}{\frac{1}{2}(U(A|B) + U(B|A))} > \delta_0 \\ U(B|A) &> u_1 \end{aligned}$$

It is noteworthy that binary interactions from A and B to C are considered only if unary interactions from A to C and from B to C do not exist. The existing conditions of binary interactions are listed as follows:

$$\left. \begin{array}{l}
 U(C|A, B) = U(C|f_2^j(A, B)) \\
 U(C|A, B) > U(C|A) \\
 U(C|A, B) > U(C|B) \\
 U(C|A) < u_1 \\
 U(C|B) < u_1 \\
 u_2 > u_1
 \end{array} \right\} \quad 1 \leq i \leq 2$$

Taking into account these concepts, a logic network can be established as follows:

- For each $U(B|f_1^i(A))$, $1 \leq i \leq 2$, nodes A , B and $f_1^i(i)$ and edges $A \rightarrow f_1^i(i)$ and $f_1^i(i) \rightarrow B$, are included in the logic network.
- For each $U(F|f_2^j(D, E))$, $1 \leq j \leq 10$, nodes D , E , F and $f_2^j(j)$ and edges $D \rightarrow f_2^j(j)$, $E \rightarrow f_2^j(j)$ and $F \rightarrow f_2^j(j)$ are included in the logic network. In this chapter, threshold values for unary and binary interactions are set to 0.25 and 0.7, respectively.

4.3 Population Dynamics P systems

In this section, *Population Dynamics P systems (PDP systems)* are introduced. PDP systems are a kind of P systems used in this chapter as a modelling framework for LNs. PDP systems can be regarded as P systems with tissue-like structure in which each node is an environment, where each of them contains the same cell-like membrane structure [52, 51]. In short, a PDP system is composed of (1) a set of connected environments describing a directed graph, (2) a cell-like structure denoting the internal membrane hierarchy associated with each environment, (3) a working alphabet of objects whose elements represent species of the modelled system and (4) a set of rules which describe how objects evolve and move inside and among environments [51].

4.3.1 Population Dynamics P systems – Formal framework

Definition 4.3. *A Population Dynamics P system of degree (q, m) with $q, m \geq 1$, taking T time units, $T \geq 1$, is a tuple*

$(G, \Gamma, \Sigma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{ij} : 1 \leq i \leq q, 1 \leq j \leq m\})$
where:

- $G = (V, S)$ is a directed graph. Let $V = \{e_1, \dots, e_m\}$ whose elements are called environments;
- $\Gamma \cup \Sigma$ is the working alphabet;
- T is a natural number that represents the simulation time of the system;
- \mathcal{R}_E is a finite set of communication rules among environments of the form

$$(x)_{e_j} \xrightarrow{p(x,j,j_1,\dots,j_h)} (y_1)_{e_{j_1}} \dots (y_h)_{e_{j_h}}$$

where $x, y_1, \dots, y_h \in \Gamma$, $(e_j, e_{j_l}) \in S$ ($l = 1, \dots, h$) and $p_{(x,j,j_1,\dots,j_h)}(t) \in [0, 1]$, for each $t = 1, \dots, T$. If $p_{(x,j,j_1,\dots,j_h)}(t) = 1$, for each t , then we omit the probabilistic function. These rules verify the following:

- ★ For each environment e_j and for each object x , the sum of functions associated with the rules from \mathcal{R}_E whose left-hand side is $(x)_{e_j}$ coincides with the constant function equal to 1.
- μ is a membrane structure (i.e. a rooted tree) consisting of q membranes, with the membranes injectively labelled by $1, \dots, q$. The skin membrane is labelled by 1. We also associate electrical charges from the set $EC = \{0, +, -\}$ with membranes.
- \mathcal{R} is a finite set of evolution rules of the form $r : u[v]_i^\alpha \rightarrow u'[v']_i^{\alpha'}$ where $u, v, u', v' \in M(\Gamma)$, $i \in \{1, \dots, q\}$, and $\alpha, \alpha' \in EC$.
- For each $r \in \mathcal{R}$ and for each j , $1 \leq j \leq m$, $f_{r,j}$ is a computable function whose domain is $\{1, \dots, T\}$ and its range is $[0, 1]$, verifying the following:
 - ★ For each $u, v \in M(\Gamma)$, $i \in \{1, \dots, q\}$ and $\alpha, \alpha' \in EC$, if r_1, \dots, r_z are the rules from \mathcal{R} whose left-hand side is (i, α, u, v) and the right-hand side have polarization α' , then $\sum_{j=1}^z f_{r_j}(t) = 1$, for each t , $1 \leq t \leq T$.
 - ★ If $(x)_{e_j}$ is the left-hand side of a rule $r \in \mathcal{R}_E$, then none of the rules of \mathcal{R} has a left-hand side of the form $(1, \alpha, u, v)$, for any $u, v \in M(\Gamma)$ and $\alpha \in EC$, having $x \in u$.
- For each j ($1 \leq j \leq m$), $\mathcal{M}_{1j}, \dots, \mathcal{M}_{qj} \in M(\Gamma)$, initially placed in the q regions of μ within environment e_j .

A system as described in Definition 4.3 can be viewed as a set of m environments e_1, \dots, e_m linked between them by the arcs in the directed graph G . Each environment e_j contains a P system, $\Pi_j = (\Gamma, \mu, \mathcal{R}, \mathcal{M}_{1j}, \dots, \mathcal{M}_{qj})$, of degree q , such that $\mathcal{M}_{1j}, \dots, \mathcal{M}_{qj}$ are the initial multisets for this environment, and every rule $r \in R$ has a computable function $f_{r,j}$ (specific for environment j) associated with it. Figure 4.3 graphically describes a PDP system.

The tuple of multisets of objects present at any moment in the m environments and at each of the regions of each Π_j , together with the polarizations of the membranes in each P system, constitutes a *configuration* of the system at that moment. At the initial configuration of the system, all environments are assumed to be empty and all membranes have a neutral polarization.

A global clock is considered, marking the time for the whole system, that is, all membranes and the application of all rules (both from \mathcal{R}_E and \mathcal{R}) are synchronized in all environments.

The PDP system can pass from one configuration to another by using the rules from $\mathcal{R} = \mathcal{R}_E \cup \bigcup_{j=1}^m \mathcal{R}_{\Pi_j}$ as follows: at each transition step, the rules to be applied are selected according to the probabilities assigned to them, and all applicable rules are simultaneously applied in a maximal way.

When a communication rule between environments $(x)_{e_j} \xrightarrow{p(x,j_1,\dots,j_h)} (y_1)_{e_{j_1}} x \dots (y_h)_{e_{j_h}}$ is applied, object x passes from e_j to e_{j_1}, \dots, e_{j_h} possibly modified into objects y_1, \dots, y_h , respectively. At any moment t , $1 \leq t \leq T$, for each object x in environment e_j , if there exist communication rules whose left-hand side is $(x)_{e_j}$, then one of these rules will be applied. If more than one communication rule can be applied to an object, the system selects one randomly, according to their probability which is given by $p_{(x,j,j_1,\dots,j_h)}(t)$.

For each j ($1 \leq j \leq m$) there is just one further restriction, concerning the consistency of charges: in order to apply several rules of \mathcal{R}_{Π_j} simultaneously to the same membrane, all the rules must have the same electrical charge on their right-hand side.

4.3.1.1 Some definitions on the model

In order to clarify the dynamics of PDP systems, some definitions are introduced here. These definitions will be used throughout this document, but are intrinsically related with the concept of PDP system.

Definition 4.4. For each $e_j \in V$ and $x \in \Gamma$, $B_{e_j,x}$ denotes the block of communication rules having $(x)_{e_j}$ as left-hand side.

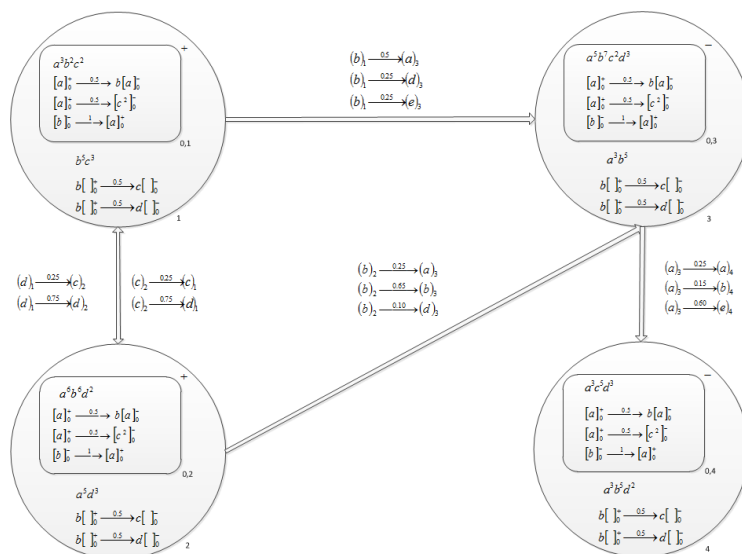


Figure 4.3: A graphical description of a PDP system

Definition 4.5. For each $u, v \in M(\Gamma)$, $1 \leq i \leq q$ and $\alpha, \alpha' \in EC$, $B_{i,\alpha,\alpha',u,v}$ denotes the block of evolution rules having $u[v]_i^\alpha$ as left-hand side, and having α' in the right-hand side.

It is important to recall that, according to the semantics of the model, the sum of probabilities of all the rules belonging to the same block is always equal to 1, in particular, rules with probability equal to 1 form individual blocks. In addition, rules with overlapping (but different) left-hand sides are classified into different blocks.

Definition 4.6. A PDP system is said to feature object competition if there exists at least a pair of overlapping left-hand sides $u[v]_i^\alpha, u'[v']_i^\alpha$, where $u, v, u', v' \in M(\Gamma)$, $u \neq u' \vee v \neq v'$, and $u \cap u' \neq \emptyset \vee v \cap v' \neq \emptyset$, $1 \leq i \leq q$, $\alpha \in EC$.

Remark 4.1. It is worth noting that all rules $r \in B_{i,\alpha,\alpha',u,v}$ can be consistently applied, in the sense that each membrane i with charge α goes to the **same** charge α' by any rule of $B_{i,\alpha,\alpha',u,v}$.

Definition 4.7. Two blocks $B_{i_1,\alpha_1,\alpha'_1,u_1,v_1}$ and $B_{i_2,\alpha_2,\alpha'_2,u_2,v_2}$ are mutually consistent with each other, if and only if $(i_1 = i_2 \wedge \alpha_1 = \alpha_2) \Rightarrow (\alpha'_1 = \alpha'_2)$.

Definition 4.8. A set of blocks $\mathcal{B} = \{B^1, B^2, \dots, B^s\}$ is self consistent (or mutually consistent) if and only if \mathcal{B} is a pairwise mutually consistent family.

Remark 4.2. *In such a context, a set of blocks \mathcal{B} has a relationship from $H \times EC$ into EC , associated with it, as follows: $((i, \alpha), \alpha')$ belongs to the relationship if and only if there exist two strings $u, v \in \Gamma^*$ such that $B_{i, \alpha, \alpha', u, v} \in \mathcal{B}$. Then, a set of blocks is mutually consistent if and only if the associated relationship is functional.*

4.4 A PDP–based model of Logic Networks

This section provides a formal description of the PDP model proposed in this work. It also provides the necessary guidelines to interpret the results of the evolution of modelled gene networks properly. This way, experts can infer the state of the modelled gene network after the predefined cycles have been simulated.

4.4.1 Logic Network Dynamic P Systems

In this subsection, Logic Network Dynamic P systems (*LNDP systems*) are presented. These P systems aim to model Logic Networks by capturing the behaviour of the improved LAPP method [229]. An LNDP system is described within an expansion of PDP systems [220, 75].

Definition 4.9. *A LNDP system Π_{LN} of degree (q, m) with $q, m \geq 1$, taking T time units, $T \geq 1$, is a tuple*

$$\Pi_{LN} = (G, \Gamma, \Sigma, T, \mathcal{R}_{\mathcal{E}}, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{ij} : 1 \leq i \leq q, 1 \leq j \leq m\}, \{\mathcal{M}_j : 1 \leq j \leq m\})$$

where:

- $(G, \Gamma, \Sigma, T, \mathcal{R}_{\mathcal{E}}, \mu, \mathcal{R}, \{f_{r,j} : r \in \mathcal{R}, 1 \leq j \leq m\}, \{\mathcal{M}_{ij} : 1 \leq i \leq q, 1 \leq j \leq m\})$ is a PDP system.
- $\Sigma = \emptyset$
- $\{f_{r,j} = 1 : r \in \mathcal{R}, 1 \leq j \leq m\}$.
- For each $j ; 1 \leq j \leq m$, \mathcal{M}_j is a multiset over Γ , describing the objects initially placed in the environment e_j .

In LNDP systems we will omit the alphabet Σ . The improved LAPP method is a deterministic algorithm, so probabilities associated with communication rules are not necessary. Thus, they are not used in LNDP systems.

In addition, LNDP systems do not feature object competition. That is, the multisets in the left-hand sides of any two rules associated with any membrane in a system are disjoint and each rule of the cell-like structure in each environment is associated with a probability function that is always equal to 1.

4.4.2 The model

Here the model for the family of Logic Network Dynamic P systems is presented. This model covers any possible P system in this family, so the multisets, rules, etc. depend on the P system which represent each specific instance of a Logic Network. The definition of the general model requires the use of parameters in its constructs, which are described in Table 4.1.

Let $LN = (\Gamma, f_1, f_2)$ be a Logic Network. Let ng , nu , nb be the number of genes, unary and binary interactions of LN , respectively. Let $n = ng + nu + nb$. This model consists on an LNDP system of degree $(1, n)$,

$$\Pi_{LN'} = (G, \Gamma, T, \mathcal{R}_E, \mu, \mathcal{R}, \{\mathcal{M}_{ij} : 0 \leq i \leq q-1, 1 \leq j \leq m\}, \{\mathcal{M}_j : 1 \leq j \leq m\})$$

where:

- G is a directed graph containing a node (environment) for each gene, unary or binary interaction.
- In alphabet Γ , gene states, interaction types, contribution weights and targets are represented as outlined below.

$$\begin{aligned} \Gamma = & \{a_i, b_i, c_i : 0 \leq i \leq 1\} \cup \{go, d_0\} \cup \{unop_j, binop_j : 1 \leq j \leq 4\} \cup \\ & \{auxDest_{i,g_j,1,k} : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb + nu\} \cup \\ & \{dest_{i,g_j,1,t_k,1+ng} : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb\} \cup \\ & \{dest_{i,g_j,1,unt_{k-nb,1+ng+nb}} : 0 \leq i \leq 1, 1 \leq j \leq ng, nb+1 \leq k \leq nb+nu\} \cup \\ & \{e_{t_{k,4}*i+(1-i)*(1-t_{k,4}),t_k,1+ng} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{e_{t_{k,6}*i+(1-i)*(1-t_{k,6}),t_k,1+ng} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{e_{unt_{k-nb,4}*i+(1-i)*(1-unt_{k-nb,4}),unt_{k-nb,1+ng+nb}} : \\ & \quad 0 \leq i \leq 1, nb+1 \leq k \leq nb+nu\} \cup \\ & \{e_{F_{t_{k,8}*i+(1-i)*(1-t_{k,8}),t_k,1+ng}} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{e_{F_{i,(unt_{k,1+ng+nb})}} : 0 \leq i \leq 1, 1 \leq k \leq nu\} \cup \\ & \{clock_j : 0 \leq j \leq cc+3\} \end{aligned}$$

- Object go triggers a new cycle in the evolution of the gene states. Objects $clock_i$ synchronize critical steps in the cycle, such as the sum of the different contributions to each gene.
- Objects a_i , ($i \in \{0, 1\}$) represent gene states: (a_0 : *inactive*; a_1 : *active*). Objects b_i represent weights of (self-influence) interactions.

- Objects $unop_j, 1 \leq j \leq 4$ participate in unary interactions, representing *strong promotion*, *strong inhibition*, *weak promotion* and *weak inhibition*, respectively. Objects $binop_j, 1 \leq j \leq 3$ participate in the binary, representing *or*, *and* and *xor*.
- Objects $dest_{i,j,k}, auxDest_{i,j,k}, e_{i,k}, c_i$ and $eF_{i,k}$ are auxiliary objects involved in interactions.
- The environment alphabet is $\Sigma = \Gamma \setminus \{d_0\}$.
- Each evolution step between cycles in real networks involves 15 computational steps, so $T = 15 \cdot Cycles$, where *Cycles* is the total number of cycles to simulate.
- $\mu = []_1$ is the membrane structure.
- The initial multisets are:
 - $\mathcal{M}_{g_{k,1}} = \{ a_1^{g_{k,3}}, a_0^{1-g_{k,3}, go}, 1 \leq k \leq ng \}$. That is, each gene environment (labelled by $g_{k,1}$), contains its gene state (a_1 :active or a_0 :inactive), depending on input $g_{k,3} \in \{0, 1\}$ and go , which triggers a new cycle.
 - $\mathcal{M}_{ng+t_{i,1}} = \{ binop_{t_{i,2}}, 1 \leq i \leq nb \}$. That is, each binary interaction environment (labelled by $ng + t_{i,1}$), contains an object $binop_{t_{i,2}}$ representing the interaction (or, and, xor).
 - $\mathcal{M}_{ng+nb+unt_{i,1}} = \{ unop_{unt_{i,2}}, 1 \leq i \leq nu \}$. That is, each unary interaction environment (labelled by $ng + nb + unt_{i,1}$, contains an object $unop_{unt_{i,2}}$ representing the interaction (strong or weak promotion or inhibition).
- The rules of \mathcal{R} and $\mathcal{R}_{\mathcal{E}}$ to apply are shown below. They are put together to follow the sequential order of execution. Environment rules start with *re* and skeleton rules start with *rs*.
 - Cycle start, and contribution of each gene over its state:

$$go a_i[]_1 \rightarrow c_i b_i^{max*i} b_0^{threshold} clock_0[]_1, 0 \leq i \leq 1$$
 - For each source gene environment:
 - * Auxiliary objects *auxDest* for all possible interactions from the source gene are created:

$$(c_i)_{g_{j,1}} \rightarrow (auxDest_{i,g_{j,1},1})_{g_{j,1}}, \dots, (auxDest_{i,g_{j,1},nb+nu})_{g_{j,1}} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq j \leq ng \end{cases}$$

- * Destination objects are created for each possible binary interaction, including information about the target environment $t_{k,1} + ng$:

$$(auxDest_{i,g_j,1,k})_{g_j,1} \rightarrow (dest_{i,g_j,1,t_{k,1}+ng})_{g_j,1} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq j \leq ng \\ 1 \leq k \leq nb \end{cases}$$

- * The same is done for each possible unary interaction, where $untk - nb, 1 + ng + nb$ represents the target environment:

$$(auxDest_{i,g_j,1,k})_{g_j,1} \rightarrow (dest_{i,g_j,1,untk-nb,1+ng+nb})_{g_j,1} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq j \leq ng \\ nb+1 \leq k \leq nb+nu \end{cases}$$

- For each actual interaction, in gene environments, objects $e_{i,k}$ (value i and target k) are created for the contribution of each source gene involved in a different interaction, from their source values $t_{k,4}$ and $t_{k,6}$ (binary interactions) and $untk-nb,4$ (unary interactions):

$$(dest_{i,t_{k,3},t_{k,1}+ng})_{t_{k,3}} \rightarrow (e_{t_{k,4}*i+(1-i)*(1-t_{k,4}),t_{k,1}+ng})_{t_{k,3}} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

$$(dest_{i,t_{k,5},t_{k,1}+ng})_{t_{k,5}} \rightarrow (e_{t_{k,6}*i+(1-i)*(1-t_{k,6}),t_{k,1}+ng})_{t_{k,5}} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

$$(dest_{i,untk-nb,3,untk-nb,1+ng+nb})_{untk-nb,3} \rightarrow (e_{untk-nb,4*i+(1-i)*(1-untk-nb,4),untk-nb,1+ng+nb})_{untk-nb,3} \quad \begin{cases} 0 \leq i \leq 1 \\ nb+1 \leq k \leq nb+nu \end{cases}$$

- Sending the values to interaction environments:

$$(e_{i,t_{k,1}+ng})_{t_{k,3}} \rightarrow (a_i)_{t_{k,1}+ng} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

$$(e_{i,t_{k,1}+ng})_{t_{k,5}} \rightarrow (a_i)_{t_{k,1}+ng} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

$$(e_{i,untk-nb,1+ng+nb})_{untk-nb,3} \rightarrow (a_i)_{untk-nb,1+ng+nb} \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

- Computing the result of interactions (1/2).

- * **or** interactions:

$$binop_1 a_0^2[]_1 \rightarrow binop_1 c_0[]_1$$

$$binop_1 a_1^2[]_1 \rightarrow binop_1 c_1[]_1$$

$$binop_1 a_1 a_0[]_1 \rightarrow binop_1 c_1[]_1$$

- * **and** interactions:

$$binop_2 a_1^2[]_1 \rightarrow binop_2 c_1[]_1$$

$$binop_2 a_0^2[]_1 \rightarrow binop_2 c_0[]_1$$

$$binop_2 a_1 a_0[]_1 \rightarrow binop_2 c_0[]_1$$

- * **xor** interactions:

$$\begin{aligned} binop_3 a_1^2[]_1 &\rightarrow binop_3 c_0[]_1 \\ binop_3 a_0^2[]_1 &\rightarrow binop_3 c_0[]_1 \\ binop_3 a_1 a_0[]_1 &\rightarrow binop_3 c_1[]_1 \end{aligned}$$

* interactions of types **strong promotion**, **strong inhibition**, **weak promotion** and **weak inhibition**, respectively:

$$\begin{aligned} unop_1 a_i[]_1 &\rightarrow unop_1 c_i[]_1 : 0 \leq i \leq 1 \\ unop_2 a_i[]_1 &\rightarrow unop_2 c_{i-1}[]_1 : 0 \leq i \leq 1 \\ unop_3 a_i[]_1 &\rightarrow unop_3 c_i^i[]_1 : 0 \leq i \leq 1 \\ unop_4 a_i[]_1 &\rightarrow unop_4 c_{1-i}^i[]_1 : 0 \leq i \leq 1 \end{aligned}$$

– Evaluating the result of the interactions (2/2).

For each interaction, objects of type eF are generated and sent to the target gene environment, depending on the previous result c_i and the contribution type (+ or -).

$$\begin{aligned} (c_i)_{t_{k,1+ng}} &\rightarrow \\ (eF_{tk,8*i+(1-i)*(1-t_{k,8}),t_{k,1+ng}})_{t_{k,7}} &\begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases} \end{aligned}$$

$$\begin{aligned} (c_i)_{unt_{k,1+ng+nb}} &\rightarrow \\ (eF_{i,(unt_{k,1+ng+nb})})_{unt_{k,5}} &\begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nu \end{cases} \end{aligned}$$

– The contribution of each interaction is calculated out of objects eF . These rules generate objects b_i whose multiplicity depends on the interaction weight.

$$eF_{i,(t_{k,1+ng})}[]_1 \rightarrow b_i^{t_{k,9}}[]_1 \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nb \end{cases}$$

$$eF_{i,(unt_{k,1+ng+nb})}[]_1 \rightarrow b_i^{unt_{k,6}}[]_1 \begin{cases} 0 \leq i \leq 1 \\ 1 \leq k \leq nu \end{cases}$$

– Once each gene has received contributions from all its interactions, the global influence over the gene is calculated. The next rule removes each pair of objects (b_1, b_0) , cancelling out their contributions.

$$b_1 b_0[]_1 \rightarrow []_1$$

– *clock* objects control the cycle flow, ensuring that all contributions caused by the interactions and auto-influences have reached their target genes.

$$clock_{i-1}[]_1 \rightarrow clock_i[]_1 : 1 \leq i \leq cc + 3$$

– If objects b_0 are present, then the gene state will be *inactive*. object d_0 is created inside the membrane 1, and in a subsequent step will imply a new polarization change. Otherwise, any objects b_1 are removed, turning the state of the gene into *active*. The remaining

objects (not used destination objects, for example) are removed from the configuration.

$$b_0[]_1^- \rightarrow [d_0]_1^-$$

$$b_1[]_1^- \rightarrow []_1^-$$

$$dest_{i,j,t,k,1+ng}[]_1^- \rightarrow []_1^- \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq j \leq ng \\ 1 \leq k \leq nb \end{cases}$$

$$dest_{i,j,unt_{k-nb,1+ng+nb}}[]_1^- \rightarrow []_1^- \quad \begin{cases} 0 \leq i \leq 1 \\ 1 \leq j \leq ng \\ nb + 1 \leq k \leq nb + nu \end{cases}$$

$$[d_0]_1^- \rightarrow []_1^+$$

- Once the last step of the cycle is reached, the state of the gene is set to *active* (1) or *inactive* (0) depending of the polarization of the membrane labelled by 1. Although electrical charges are not a part of gene regulation, its use is required to set the state of the skin membrane of each environment, ensuring that all remaining objects d_0 are removed. In addition, the corresponding *go* objects are generated, the clock is removed and the polarization of the membrane is reset to 0.

$$clock_{cc+3}[]_1^+ \rightarrow go\ a_0[]_1^0$$

$$clock_{cc+3}[]_1^- \rightarrow go\ a_1[]_1^0$$

4.4.3 LN state interpretation

After the P system has taken a predefined number of computation steps, the output information, encoded in the multiplicity of objects a_1 and a_0 , is analysed. Environments with an object a_1 represent active genes (a_0 represent inactive genes). Due to the nature of the system, gene environments (that is, environments representing genes) cannot contain objects a_1 and a_0 simultaneously. If no object a_1 or a_0 is present within the environment gene, then the state of its represented gene cannot be evaluated yet. That is, it will take some additional computation steps for the system to reach an evaluable state.

In Chapter 6, Section 6.1, a case study on the evolution of a gene network extracted from real-life data is presented as an example of application of this framework.

Parameter	Description
General parameters for the system	
ng	Number of genes in the network
nb	Number of binary interactions
nu	Number of unary interactions
$threshold$	Maximum strength for an interaction
cc	Clock control
Gene configuration parameters	
$g_{i,1}$	Gene number (id)
$g_{i,3}$	Initial state of the gene
Binary interactions parameters	
$t_{i,1}$	Binary interaction number (id)
$t_{i,2}$	Interaction type (or: 1, and: 2, xor: 3)
$t_{i,3}$	1 st source gene number (id)
$t_{i,4}$	1 st source gene contribution (positive: 1, negative: 0)
$t_{i,5}$	2 nd source gene number (id)
$t_{i,6}$	2 nd source gene contribution (positive: 1, negative: 0)
$t_{i,7}$	Destination gene number (id)
$t_{i,8}$	Influence over destination gene (positive: 1, negative: 0)
$t_{i,9}$	Strength of the destination
Unary interactions parameters	
$unt_{i,1}$	Unary interaction number (id)
$unt_{i,2}$	Interaction type (strong promotion: 1, inhibition: 2; weak ones: 3, 4)
$unt_{i,3}$	Source gene number (id)
$unt_{i,4}$	Source gene contribution (positive, negative)
$unt_{i,5}$	Destination gene number (id)
$unt_{i,6}$	Influence over destination gene (positive, negative)

Table 4.1: Parameters for LNDP systems

Chapter 5

Probabilistic Guarded P Systems

Probabilistic Guarded P systems (*PGP systems*, for short) are a brand new model for the simulation of real-life phenomena, specifically for ecological processes. PGP systems are a computational probabilistic framework which takes inspiration from different Membrane Computing paradigms, mainly from Tissue-Like P systems [133, 162], PDP systems [51] and Kernel P systems [77, 79, 112, 78]. This framework aims for simplicity, considering these aspects:

Model designers: In PGP systems, model designers do not need to worry about context consistency. That is to say, they do not need to take into account that all rules simultaneously applied in a cell define the same polarization in the right-hand side. This is because the framework centralizes all context changes in (at most) a single rule per cycle, rather than distributing them across all rules. Therefore, there exist two types of rules: *context-changing* rules and *non context-changing* rules. Due to the nature of the model, only one of such rules can be applied at the same time on each cell, so context inconsistency is not possible. Moreover, the fact that the context is explicitly expressed in each cell and that cells do not contain internal cell structures simplifies transitions between contexts without loss of computational and modelling power.

Simulator developers: The fact that the framework implicitly takes care of context consistency simplifies the development of simulators for these models, as it is a non-functional requirement which does not need to be supported by simulators. In addition, the lack of internal structure in cells simplifies the simulation of object transmission; the model can be

regarded as a set of memory regions with no hierarchical arrangement, thus enabling direct region fetching.

Probabilistic Guarded P Systems can be regarded as an evolution of Population Dynamic P systems. In this context, PGP systems propose a modelling framework for ecology in which inconsistency (that is to say, undefined context of membranes) is handled by the framework itself, instead of delegating to simulation algorithms. In addition, by replacing alien concepts to biology (such as electrical polarizations and internal compartment hierarchies) by state variables known as *flags* and defined by designers models are more natural to experts, thus simplifying communication between expert and designer.

This chapter is structured as follows. Section 5.1 provides a formalization for PGP systems. Section 5.2 defines sequential algorithms for the simulation of PGP systems without object competition. Section 5.2 proposes a parallel algorithm to simulate PGP systems without object competition, proposing some ideas to deal with this feature. Finally, Section 5.4 presents a Graphical User Interface (GUI) for the analysis and simulation of PGP systems.

5.1 Formal description of PGP systems

What follows is a formalization of PGP systems, followed by a description and some remarks over its semantics.

5.1.1 Formalization of PGP systems

Definition 5.1. *A Probabilistic Guarded P system (PGP system, for short) of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Phi, \mathcal{R}, \mathcal{G}_{\mathcal{R}}, p_{\mathcal{R}}, (f_1, \mathcal{M}_1), \dots, (f_q, \mathcal{M}_q))$ where:*

- Γ and Φ are finite alphabets such that $\Gamma \cap \Phi = \emptyset$. Elements in Γ are called **objects** and elements in Φ are called **flags**.
- \mathcal{R} is a finite set of rules of the following types:
 - $\{f\} [u]_i \rightarrow [v]_j$ with $u, v \in M(\Gamma)$, $f \in \Phi$ and $1 \leq i, j \leq q$. If $i = j$, then $\{f\} [u]_i \xrightarrow{p} [v]_j$ can be denoted as $\{f\} [u \xrightarrow{p} v]_i$.
 - $\{f\} [u, f]_i \rightarrow [v, g]_i$ with $u, v \in M(\Gamma)$, $f, g \in \Phi$ and $1 \leq i, j \leq q$. Such a rule can be also denoted as $\{f\} [u, f \rightarrow v, g]_i$. Moreover, for each $f \in \Phi, u \in M(\Gamma), 1 \leq i \leq q$, there exists only one rule of type $\{f\} [u, f]_i \rightarrow [v, g]_i$.

- $\mathcal{G}_{\mathcal{R}}$ is the directed graph associated with \mathcal{R} as follows: $V = \{1, \dots, q\}$ and $(i, j) \in E$ if and only if there exists a rule of the type $\{f\} [u]_i \rightarrow [v]_j$, or $i = j$ and there exists a rule of the type $\{f\} [u, f]_i \rightarrow [v, g]_i$.
- $p_{\mathcal{R}}$ is a map from \mathcal{R} into $[0, 1]$ such that:
 - If $r \equiv \{f\} [u, f]_i \rightarrow [u, g]_i$, then $p_{\mathcal{R}}(r) = 1$.
 - For each $f \in \Phi, u \in M(\Gamma), 1 \leq i \leq q$, if r_1, \dots, r_t are rules of the type $\{f\} [u]_i \rightarrow [v]_j$, then $\sum_{k=1}^t p_{\mathcal{R}}(r_k) = 1$.
- For each i ($1 \leq i \leq q$), we have $f_i \in \Phi$ and $\mathcal{M}_i \in M(\Gamma)$.

Remark 5.1. A Probabilistic Guarded P system can be viewed as a set of q cells labelled by $1, \dots, q$ such that: (a) $\mathcal{M}_1, \dots, \mathcal{M}_q$ are finite multisets over Γ representing the objects initially placed in the q cells of the system; (b) f_1, \dots, f_q are flags that initially mark the q cells; (c) $\mathcal{G}_{\mathcal{R}}$ is a directed graph whose arcs specify connections among cells; (d) \mathcal{R} is the set of rules that allow the evolution of the system and each rule $r \in \mathcal{R}$ is associated with a real number $p_{\mathcal{R}}(r)$ in $[0, 1]$ meaning the probability of that rule to be applied in the case that it is applicable.

Remark 5.2. In PGP systems, two types of symbols are used: objects (elements in Γ) and flags (elements in Φ). It can be considered that objects are **in** cells and flags are **on** (the borderline of) cells.

5.1.2 Semantics of PGP systems

Definition 5.2. A configuration at any instant $t \geq 0$ of a PGP system Π is a tuple $\mathcal{C}_t = (x_1, u_1, \dots, x_q, u_q)$ where, for each i , $1 \leq i \leq q$, $x_i \in \Phi$ and $u_i \in M(\Gamma)$. That is to say, a configuration of Π at any instant $t \geq 0$ is described by all multisets of objects over Γ associated with all the cells present in the system and the flags marking these cells. $(f_1, \mathcal{M}_1, \dots, f_q, \mathcal{M}_q)$ is said to be the initial configuration of Π . At any instant, each cell has one and only one flag, in a similar manner to polarizations in cell-like P systems.

Definition 5.3. A rule r of the type $\{f\} [u]_i \rightarrow [v]_j$ is applicable to a configuration $\mathcal{C}_t = (x_1, u_1, \dots, x_q, u_q)$ if and only if $x_i = f$ and $u \subseteq u_i$, for all $1 \leq i \leq q$.

When applying r to \mathcal{C}_t , objects in u are removed from cell i and objects in v are produced in cell j . Flag f is not changed; it plays the role of a catalyst assisting the evolution of objects in u .

$$\begin{array}{c}
\boxed{a^4 \ b^4, \ e}_0 \quad \boxed{a^3 \ c^2, \ d}_1 \quad \boxed{b^2, \ f}_2 \\
\bullet R_0: \\
\left. \begin{array}{l} \{e\}[a^2]_0 \xrightarrow{0.2} [c^2]_1 \\ \{e\}[a^2]_0 \xrightarrow{0.7} [b]_0 \end{array} \right| \{e\}[a, e]_0 \rightarrow [c, f]_0 \quad \{f\}[b^3, f]_0 \rightarrow [e]_0 \\
\bullet R_1: \\
\left. \begin{array}{l} \{f\}[a^2]_1 \xrightarrow{0.4} [b]_1 \\ \{f\}[a^2]_1 \xrightarrow{0.6} [c^2]_2 \end{array} \right| \left. \begin{array}{l} \{d\}[c]_1 \xrightarrow{0.2} [c]_2 \\ \{d\}[c]_1 \xrightarrow{0.7} [a^2]_0 \end{array} \right| \{f\}[c^2]_1 \rightarrow [b]_0 \\
\bullet R_2: \\
\{f\}[b^2, f]_2 \rightarrow [a^2, d]_2 \quad \left. \begin{array}{l} \{d\}[a]_2 \xrightarrow{0.5} [b^2]_0 \\ \{d\}[a]_2 \xrightarrow{0.5} [c]_1 \end{array} \right|
\end{array}$$

Figure 5.1: An example of PGP system. Flags are highlighted in red and probabilities equal to 1 are omitted.

Definition 5.4. A rule r of the type $\{f\}[u, f]_i \rightarrow [v, g]_i$ is applicable to a configuration $C_t = (x_1, u_1, \dots, x_q, u_q)$ if and only if $x_i = f$ and $u \subseteq u_i$, for all $1 \leq i \leq q$.

When applying r to C_t , in cell i objects in u are replaced by those in v and f is replaced by g . In this case, Flag f is consumed, so r can be applied only once in instant t in cell i .

Remark 5.3. After applying a rule r of the type $\{f\}[u, f]_i \rightarrow [v, g]_i$, other rules r' of the type $\{f\}[u]_i \rightarrow [v]_j$ can still be applied (the flag remains in vigor). However, f has been consumed, so no more rules of the type $\{f\}[u, f]_i \rightarrow [v, g]_i$ can be applied.

Definition 5.5. A configuration is a halting configuration if no rule is applicable to it.

Definition 5.6. We say that configuration C_1 yields configuration C_2 in a transition step if we can pass from C_1 to C_2 by applying rules from \mathcal{R} in a non-deterministic, maximally parallel manner, according to their associated probabilities denoted by map $p_{\mathcal{R}}$. That is to say, a maximal multiset of rules from \mathcal{R} is applied, no further rule can be added.

Definition 5.7. A computation of a PGP system Π is a sequence of configurations such that: (a) the first term of the sequence is the initial configuration of Π , (b) each remaining term in the sequence is obtained from the previous

one by applying the rules of the system following Definition 5.6, (c) if the sequence is finite (called halting computation) then the last term of the system is a halting configuration.

5.1.3 Comparison between PGP systems and other frameworks in Membrane Computing

Probabilistic Guarded P systems (*PGP systems*) display similarities with other frameworks in Membrane Computing. As a sample, in *P systems with proteins on membranes* are a type of cell-like systems in which membranes might have attached a set of proteins which regulate the application of rules, whilst in PGP systems each cell has only one flag. Therefore, some rules are applicable if and only if the corresponding protein is present. More information about this kind of P systems can be found in [185, 186].

When comparing PGP systems and *Population Dynamics P systems* [51], it is important to remark the semantic similarity between flags and polarizations, as they both define at some point the context of each compartment. Nevertheless, as described at the beginning of this chapter, upon the application of a rule $r \equiv \{f\} [u, f]_i \rightarrow [v, g]_i$ flag f is consumed, thus ensuring that r can be applied at most once to any configuration. This property keeps PGP transitions from yielding inconsistent flags; at any instant, only one rule at most can change the flag in each membrane, so scenarios in which inconsistent flags produced by multiple rules are impossible. Moreover, in PDP systems the number of polarizations is limited to three (+, - and 0), whereas in their PGP counterpart depends on the system itself. Finally, each compartment in PDP systems contains a hierarchical structure of membranes, which is absent in PGP systems. Figure ?? summarizes this comparison.

	PGP systems	P systems with proteins	PDP systems
<i>Structure</i>	Tissue-like (given by a directed graph)	Cell-like (given by a rooted tree)	Tissue-like (given by a directed graph of environments containing a rooted tree each)
<i>Rule</i>	Each left-hand side contains one flag and a multiset of objects	Each left-hand side contains one protein and one object	Each left-hand side contains one polarization and a multiset of objects
<i>Affected compartments</i>	The application of a rule might affect, at most, two cells in the system	The application of a rule affects one and only one cell in the system	The application of a rule might affect, at most, two cells in the system
<i>Number of applications</i>	Each rule of type $r \equiv \{f\} [u, f]_i \rightarrow [v, g]_i$ can be applied, at most, only once to any configuration	Every rule is possible to be applied multiple times to any configuration	Every rule is possible to be applied multiple times to any configuration
<i>Number of flags</i>	For each configuration, there exists only one flag per cell	For each configuration, there might exist multiple proteins per cell	For each configuration, there exists only one polarization per cell

Figure 5.2: Comparison of PGP systems and P systems with proteins

5.1.4 Some definitions on the model

As it is the case in Logic Network Dynamic P systems, in PGP systems some definitions are introduced prior to describing simulation algorithms. These concepts are analogous to those described in Chapter 4, but obviously adapted to the syntax of PGP systems.

Remark 5.4. *For the sake of simplicity, henceforth the following notation will be used. For every cell i , $1 \leq i \leq q$, and t , $0 \leq t \leq T$, the flag and multiset of cell i in step t are denoted as $x_{i,t} \in \Phi$ and $\mathcal{M}_{i,t} \in M(\Gamma)$, respectively. Similarly, $|u|_y$, where $u \in M(\Gamma)$, $y \in \Gamma$ denote the number of objects y in multiset u .*

Definition 5.8. *For each $r \in \mathcal{R}_{\mathcal{E}}$ such as r is of the form $r = \{f\}[u]_i \rightarrow [v]_j$, i and j denote the left hand-side label and right-hand side label respectively. Similarly, if r is of the form $r = \{f\}[u, f]_i \rightarrow [v, g]_i$, i denotes both the left hand-side and the right-hand side label.*

Definition 5.9. *For each $u \in M(\Gamma)$, $f \in \Phi$ and $1 \leq i \leq q$, $B_{i,f,u} = \{r_{i,j,1}, \dots, r_{i,j,h_{i,j}}\}$ denotes the block of communication rules having $\{f\}[u]_i$ as left-hand side. Similarly, $B_{i,f,u,f} = \{r_{i,k,1}\}$ denotes the block of context-changing rules having $\{f\}[u, f]_i$ as left-hand side.*

Remark 5.5. *For each i , $1 \leq i \leq q$, we consider a total order in the set of all blocks associated with cell i : $\{B_{i,1}, \dots, B_{i,o_i}\}$, where o_i denotes the number of different blocks composed of rules associated with cell i . In addition, we consider a total order in $B_{i,j} = \{r_{i,j,1}, \dots, r_{i,j,h_{i,j}}\}$, where $h_{i,j}$ ($1 \leq i \leq q$, $1 \leq j \leq o_i$) denotes the number of rules in block $B_{i,j}$.*

It is important to recall that, as it is the case in PDP systems, the sum of probabilities of all the rules belonging to the same block is always equal to 1 – in particular, rules with probability equal to 1 form individual blocks. Consequently, blocks of context-changing rules are composed solely of a rule. In addition, rules with overlapping (but different) left-hand sides are classified into different blocks.

Definition 5.10. *A PGP system is said to feature object competition if there exists at least a pair of overlapping left-hand sides (possibly of different type) $\{f\}[u]_i$, $\{f\}[v]_i$ or $\{f\}[u, f]_i$, $\{f\}[v, f]_i$, where $u, v \in M(\Gamma)$, $u \neq v$ and $u \cap v \neq \emptyset$, $1 \leq i \leq q$, $f \in \Phi$.*

Remark 5.6. *It is worth noting that all rules in the model can be consistently applied. This is because there can only exist one flag $f \in \Theta$ at every membrane at the same time, and, consequently, at most one context-changing rule $r \equiv \{f\} [u, f]_i \rightarrow [v, g]_i$ can consume f and replace it (where possibly $f = g$).*

Definition 5.11. *Given a block $B_{i,f,u}$ or $B_{i,f,u,f}$, where $u \in M(\Gamma)$, $f \in \Phi$, $1 \leq i \leq q$ and a configuration $C_t = \{x_1, \mathcal{M}_1, \dots, x_q, \mathcal{M}_q\}$, $0 \leq t \leq T$, the maximum number of applications of such a block in C_t is the maximum applications of any of its rule in C_t .*

5.2 Simulation of PGP systems

When simulating PGP systems, there exist two cases, according to the model: if there exists object competition or not. In this work, only algorithms for the second case are introduced, but some ideas are given to handle object competition.

5.2.1 Temporary data structures

In addition to the elements of PGP systems, some data structures are used as temporary buffers in simulators, which are:

AB (Applicable Blocks): an array of characters of dimension $q \times N_{BM}$, where N_{BM} is the maximum number of blocks for all membranes. On every instant t , each element $AP_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq N_{BM}$, stores *true* if $x_i = f$ and *false* if $f_i \neq f$, where $B_{i,j} = B_{i,f,u} \vee B_{i,f,u,f}$.

NBA (Number of Block Applications): an array of integer numbers of dimension $q \times N_{BM}$ in which each element $NBA_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq N_{BM}$, stores the number of applications of block $B_{i,j}$.

NRA (Number of Rule Applications): an array of integer numbers of dimension $q \times N_{BM} \times N_{RM}$, where N_{RM} is the maximum number of rules for all blocks in all membranes. Each element $NRA_{i,j,k}$, $1 \leq i \leq q$, $1 \leq j \leq N_{BM}$, $1 \leq k \leq N_R$, stores the number of applications of rule $r_{i,j,k}$, identified by its cell, block and local identifier inside its block.

5.2.2 Simulation algorithm

The algorithm for simulation of PGP systems receives three parameters: the P system Π to simulate, the number of steps T and an integer number R_{iter}

which indicates for how many cycles block applications are assigned among their rules. That is, the algorithm distributes the applications of each block among its rules for R_{iter} cycles, and after that, block applications are maximally assigned among rules in a single cycle. Algorithm 5.2.4 performs this function. When simulating PGP systems without object competition, it is not necessary to randomly assign objects among blocks; as they do not compete for objects, then the number of times that each block is applied is always equal to its maximum number of applications. As it is the case of DCBA for PDP systems [138], the simulation algorithm heavily relies on the concept of block, being rule applications secondary. However, DCBA handles object competition among blocks, penalizing more those blocks which require a larger number of copies of the same object, which is inspired in the amount of energy required to join individuals from the same species, whereas object competition is not supported on the proposed algorithm. Algorithm 5.2.1 describes a simulation algorithm for PGP systems without object competition.

Algorithm 5.2.1 Algorithm for simulation of PGP systems

Input:

- T : an integer number $T \geq 1$ representing the iterations of the simulation.
- R_{Iter} an integer number $R_{Iter} \geq 1$ representing non-maximal rule iterations (i.e., iterations in which the applications selected for each rule do not necessarily need to be maximal).
- $\Pi = (\Gamma, \Phi, \mathcal{R}, \mathcal{G}_{\mathcal{R}}, p_{\mathcal{R}}, (f_1, \mathcal{M}_1), \dots, (f_q, \mathcal{M}_q))$: a PGP system of degree $q \geq 1$.

```

1: for  $t \leftarrow 1$  to  $T$  do
2:   Check block flags (see Algorithm 5.2.2)
3:   Distribute objects among blocks (see Algorithm 5.2.3)
4:   Distribute applications among rules (see Algorithm 5.2.4)
5:   Generate objects (see Algorithm 5.2.5)
6: end for

```

On each simulation step t , $1 \leq t \leq T$ and membrane i , $1 \leq i \leq q$, the following stages are applied: *Flag checking*, *Object distribution*, *Rule application distribution* and *Object generation*.

5.2.2.1 Flag checking

The first stage consists on checking which rules are guarded by flags which comply with $f_i \in \Phi$. If this is true, then a marker $AB_{i,j}$ for block j in membrane i is set to *true*, setting it to *false* otherwise. This marker is checked later in

Algorithm 5.2.3 to check block applicability. Algorithm 5.2.2 describes this procedure.

Algorithm 5.2.2 Flag checking

```

for  $i \leftarrow 1$  to  $q$  do
  for  $j \leftarrow 1$  to  $o_i$  do
    if  $B_{i,j} = B_{i,f,u} \vee B_{i,j} = B_{i,f,u,f}$  then  $\triangleright$  If the rule is guarded by flag  $f$ 
      if  $x_{i,t-1} = f$  then
         $AB_{i,j} \leftarrow true$ 
      else
         $AB_{i,j} \leftarrow false$ 
      end if
    end if
     $NBA_{i,j} \leftarrow 0$ 
    for  $k \leftarrow 1$  to  $h_{i,j}$  do  $\triangleright$  Initially, all rule applications are 0
       $NRA_{i,j,k} \leftarrow 0$ 
    end for
  end for
end for

```

5.2.2.2 Object distribution

In this stage, objects are distributed among blocks. As the system to simulate does not feature object competition, the number of applications of each block is its maximum. Then, objects are consumed accordingly. Algorithm 5.2.3 describes this procedure.

Algorithm 5.2.3 Object distribution

```

for  $i \leftarrow 1$  to  $q$  do
  for  $j \leftarrow 1$  to  $o_i$  do
    if  $AB_{i,j} = true$  then
       $NBA_{i,j} \leftarrow \min(\frac{\mathcal{M}_{i,t-1}(y)}{|u|_y}) \forall y \in \Gamma$ , where
       $B_{i,j} = B_{i,f,u} \vee B_{i,j} = B_{i,f,u,f}$  and  $|u|_y > 0$  ▷ The number
      of block applications is the minimum of all quotients between available
      and consumed objects
       $\mathcal{M}_{i,t} = \mathcal{M}_{i,t-1} - NBA_{i,j} \cdot u$ 
    else
       $NBA_{i,j} \leftarrow 0$ 
    end if
    if  $NBA_{i,j} > 0 \vee B_{i,j} = B_{i,f,u,f}$  then
       $x_{i,t} \leftarrow f$  ▷ Update the membrane flag
    end if
  end for
end for

```

5.2.2.3 Rule application distribution

Next, objects are distributed among rules according to a binomial distribution with rule probabilities and maximum number of block applications as parameters. This algorithm is composed of two stages *maximal* and *non-maximal* repartition. In the maximal repartition stage, a rule in the block is randomly selected according to a uniform distribution, so each rule has the same probability to be chosen. Then, its number of applications is calculated according to an *ad-hoc* procedure based on a binomially distributed variable $B(n, p)$, where n is the remaining number of block applications to be assigned among its rules and p is the corresponding rule probability. This process is repeated a number R_{iter} of iterations for each block $B_{i,j}$, $1 \leq i \leq q$, $1 \leq j \leq o_i$. Algorithm 5.2.4 describes this procedure. If, after this process, there are still applications to assign among rules, a rule per applicable block is chosen at random and as many applications as possible are assigned to it in the maximal repartition stage. An alternative approach would be to implement a multinomial distribution of applications for the rules inside each block, such as the way that it is implemented on the DCBA algorithm [138]. A method to implement a multinomial distribution would be the conditional distribution method, which emulates a multinomial distribution based on a sequence of binomial distri-

butions [59]. This would require to normalize rule probabilities for each rule application distribution iteration. This approach has also been tested on the simulation algorithm, but was discarded because it tended to distribute too few applications in the non-maximal repartition stage, thus leaving too many applications for the rule selected in the maximal repartition one.

Algorithm 5.2.4 Rule application distribution

```

for  $i \leftarrow 1$  to  $q$  do
  for  $j \leftarrow 1$  to  $o_i$  do
    for  $k \leftarrow 1$  to  $h_{i,j}$  do
       $NRA_{i,j,k} \leftarrow 0$ 
    end for
  end for
end for
for  $l \leftarrow 1$  to  $R_{Iter}$  do
  for  $i \leftarrow 1$  to  $q$  do
    for  $j \leftarrow 1$  to  $o_i$  do
       $k \leftarrow$  a uniform random integer number in  $\{1, \dots, h_{i,j}\}$   $\triangleright$  Select a
      random rule  $r_{i,j,k}$  in Block  $B_{i,j}$ 
       $lnrap \leftarrow B(NBA_{i,j}, p_{\mathcal{R}}(r_{i,j,k}))$   $\triangleright B$  is the binomial distribution
       $NRA_{i,j,k} \leftarrow NRA_{i,j,k} + lnrap$   $\triangleright$  Update rule applications
       $NBA_{i,j} \leftarrow NBA_{i,j} - lnrap$ 
    end for
  end for
end for
for  $i \leftarrow 1$  to  $q$  do
  for  $j \leftarrow 1$  to  $o_i$  do
     $k \leftarrow$  a uniform random integer number in  $\{1, \dots, h_{i,j}\}$ 
     $NRA_{i,j,k} \leftarrow NRA_{i,j,k} + NBA_{i,j}$ 
     $NBA_{i,j} \leftarrow 0$ 
  end for
end for

```

5.2.2.4 Object generation

Lastly, rules are applied as indicated in their right-hand side. Each rule generates objects according to its previously assigned number of applications. Algorithm 5.2.5 describes this procedure.

Algorithm 5.2.5 Object generation

```

for  $i \leftarrow 1$  to  $q$  do
  for  $j \leftarrow 1$  to  $o_i$  do
    for  $k \leftarrow 1$  to  $h_{i,j}$  do
       $\mathcal{M}_{i,t} \leftarrow \mathcal{M}_{i,t} + NRA_{i,j,k} \cdot v$ , where  $RHS(r_{i,j,k}) = [v]_j \vee$ 
       $RHS(r_{i,j,k}) = [v, f]_i$ 
    end for
  end for
end for

```

5.2.3 Simulation algorithm with object competition

The algorithm proposed in this chapter works only for models without object competition. This is because the models studied in the case studied did not have object competition, so this feature was not required. However, it might be interesting to develop new algorithms supporting it. They would be identical to their counterpart without object competition, solely differing in the protocol by which objects are distributed among blocks. As an example, it would be possible to adapt the way in which objects are distributed in the DCBA algorithm [139].

5.3 Parallel simulation of PGP systems

In this section, a parallel algorithm for simulation of PGP systems is described. This algorithm has been implemented on CUDA/C++, so as to take advantage of the parallel architecture of GPU cards. PGP systems are probabilistic models, therefore, repeated simulation of the same system helps understand its dynamics with better accuracy than single simulation, as outliers are filtered out and statistical metrics over output values, such as mean and typical deviation, are progressively approximated due to the law of large numbers, which states that, when a random experiment is repeated *ad infinitum*, the percentage difference between the expected and actual values tends to zero [96].

5.3.1 Simulator data structures

The arrays employed to simulate PGP systems on parallel architectures are:

C (Cardinalities): an array of integer numbers of dimension $q \times N_\Gamma \times T$, where $q \geq 1$ is the degree of the system, N_Γ is the size of Γ and $T \geq 0$

is the number of steps simulated. Each element $C_{i,j,t}$, $1 \leq i \leq q, 1 \leq j \leq N_\Gamma, 1 \leq t \leq T$, represents the number of objects of type j in cell i in configuration t .

F (Flags): an array of integer numbers of dimension $q \times t$. Each element $F_{i,t}$, ($1 \leq i \leq q, 1 \leq t \leq T$), represents the flag in cell i in configuration t .

BC (Block Cardinalities): an array of integer numbers of dimension $N_B \times N_\Gamma$. Each element $BC_{k,j}$, $1 \leq k \leq N_B, 1 \leq j \leq N_\Gamma$, represents the number of objects of type j consumed by block k .

BL (Block Labels): an array of integer numbers of dimension N_B . Each element BL_k , $1 \leq k \leq N_B$, represents the label of block k .

BF (Block Flags): an array of integer numbers of dimension N_B , where $N_B \geq 1$ is the total number of blocks in the system. Each element BF_k , $1 \leq k \leq N_B$, represents the flag of block k .

BT (Block Type): an array of characters of dimension N_B . Each element BT_k , $1 \leq k \leq N_B$, can have value *true* or *false*. If $BT_k = \textit{true}$, then BT_k is of the form $B_{i,f,u,f}$. Otherwise, BT_k is of the form $B_{i,f,u}$.

BS (Block Size): an array of integer numbers of dimension N_B . Each element BS_k , $1 \leq k \leq N_B$, represents the number of rules in B_k .

BR (Block Rules): an array of integer numbers of dimension $N_B \times M_B$, where $M_B = \max\{BS_k : 1 \leq k \leq N_B\}$. Each element $BR_{k,l}$, $1 \leq k \leq N_B, 1 \leq l \leq M_B$, identifies a rule $r_{k,l}$ in block B_k .

RL (Rule right-hand side Label): an array of integer numbers of dimension N_R , where $N_R \geq 1$ is the total number of rules in the system. Each element RL_l , $1 \leq l \leq N_R$, represents the right-hand side label of rule r_l .

RC (Rule right-hand side Cardinalities): an array of integer numbers of dimension $N_R \times N_\Gamma$. Each element $RC_{l,j}$, $1 \leq l \leq N_R, 1 \leq j \leq N_\Gamma$, represents the number of objects of type j consumed by rule r_l .

RPROB (Rule Probability): an array of real numbers of dimension N_R . Each element $RPROB_l$, $1 \leq l \leq N_R$, represents the probability of rule r_l .

NCB (Non-Consuming Blocks): an array of characters of dimension N_B in which $NCB_k = true$, $1 \leq k \leq N_B$, if $B_{[\emptyset, f]_i}$, $f \in \Phi$, $1 \leq i \leq q$, and $NCB_k = false$ otherwise.

In addition to the structures used to represent the system, other arrays are used as well to store temporary data necessary for simulations. These are:

AB (Applicable Blocks): an array of characters of dimension N_B in which each element AP_k , $1 \leq k \leq N_B$, stores *true* if $f_i = f$ and *false* if $f_i \neq f$, where $B_k = B_{i, f, u} \vee B_{i, f, u, f}$.

NBA (Number of Block Applications): an array of integer numbers of dimension N_B in which each element NBA_k , $1 \leq k \leq N_B$, stores the number of applications of block B_k .

NRA (Number of Rule Applications): an array of integer numbers of dimension N_R in which each element NRA_l , $1 \leq l \leq N_R$, stores the number of applications of rule r_l .

5.3.2 Simulation algorithm

Algorithm 5.3.1 describes how threads are distributed to take advantage of parallel architectures when simulating PGP systems without object competition. The meaning of each stage is the same as in its sequential counterpart Algorithm 5.2.1.

Algorithm 5.3.1 has been defined so it can be implemented on any parallel platform. In particular, an implementation on CUDA (namely PGPCUDA) has been provided as a result of this thesis. In this implementation, each parallel call in Algorithm 5.3.1 is implemented as a kernel. As it is explained in Chapter 2, in CUDA threads are distributed in thread blocks. The way in which threads are arranged in blocks in each kernel is sometimes chosen because of convenience to identify data or, in the case of a base block dimension of 256, it was the one which gave best performance results. Table 5.3 explains this thread distribution.

Algorithm 5.3.1 Parallel algorithm

Input:

- T : an integer number $T \geq 1$ representing the iterations of the simulation.
- B_{Iter} : an integer number $B_{Iter} \geq 1$ representing non-maximal block iterations.
- R_{Iter} : an integer number $R_{Iter} \geq 1$ representing non-maximal rule iterations.
- Data structures from Subsection 5.3.1.

```

1: for  $t \leftarrow 1$  to  $T$  do
2:   Check block flags on  $N_B$  threads (see Algorithm 5.3.2)
3:   Clear rule applications on  $N_R$  threads (see Algorithm 5.3.3)
4:   Calculate block applications on  $N_B \times N_\Gamma$  threads (see Algorithm 5.3.4)
5:   Clear non-processed block applications on  $N_B$  threads (see Algorithm 5.3.5)
6:   Consume block objects on  $q \times N_\Gamma$  threads (see Algorithm 5.3.6)
7:   for  $n \leftarrow 1$  to  $R_{Iter}$  do
8:     Distribute applications among rules on  $N_B$  threads (see Algorithm 5.3.7)
9:   end for
10:  Distribute applications among rules maximally on  $N_B$  threads (see Algorithm 5.3.9)
11:  Generate objects on  $N_R \times N_\Gamma$  threads (see Algorithm 5.3.10)
12: end for

```

Algorithm number	Block structure
5.3.2	$256 \times S \times N_B$
5.3.3	$256 \times S \times N_R$
5.3.4	$256 \times S \times N_\Gamma \times N_B$
5.3.5	$256 \times S \times N_B$
5.3.6	$256 \times S \times N_B \times q$
5.3.7	$256 \times S \times N_B$
5.3.9	$256 \times S \times N_B$
5.3.10	$256 \times S \times N_\Gamma \times N_R$

Figure 5.3: Thread distribution among blocks for the proposed GPU implementation of Algorithm 5.3.1. S , N_Γ , q , N_B and N_R denote the number of simulations and the alphabet size, degree, total number of blocks and total number of rules in the system, respectively. Block size was set to 256 because it offered the best performance results.

Algorithm 5.3.2 Parallel flag checking

Input:

- F , BL , BF , AB , and NBA : see Subsections 5.2.1 and 5.3.1.
- l : an integer number, $1 \leq l \leq N_B$, identifying the thread in which the algorithm is applied

if $F_{BL_l, t-1} = BF_l$ **then**

$AB_l \leftarrow true$

else

$AB_l \leftarrow false$

end if

$NBA_l \leftarrow \infty$

5.4 Software environment

A simulator for PGP systems without object competition has been incorporated on P-Lingua. In addition, as it is the case in ENPSs discussed in Chapter 3, a C++ simulator for PGP systems (namely PGPC++) has also been implemented, so as to measure the performance gain of PGPCUDA by using a low-level programming language. The libraries used for random number generation during simulations are COLT [3] in the P-Lingua simulator, standard `std::rand` [15] and CURAND [10] for PGPC++ and PGPCUDA, respectively. In the case of PGPCUDA, CURAND is used as an auxiliary library to generate

Algorithm 5.3.3 Parallel rule applications clearing

Input:

- NRA : Data structures from Subsection 5.2.1.
- l : an integer number, $1 \leq l \leq N_R$, identifying the thread in which the algorithm is applied.

$$NRA_l \leftarrow 0$$

Algorithm 5.3.4 Calculate block applications

Input:

- C , BC , BL , AB , NBA and NCB : see Subsections 5.2.1 and 5.3.1
- l, j : two integer numbers, $1 \leq l \leq N_B, 1 \leq j \leq N_\Gamma$, identifying the thread in which the algorithm is applied.

```

if  $AB_l = true$  then
  if  $NCB_l = true$  then                                 $\triangleright$  If Block  $l$  is of the form  $B_{i,f,\emptyset,f}$ 
     $NBA_l \leftarrow 1$ 
  else
    if  $BC_{l,j} > 0$  then
       $lmaxnbap \leftarrow \frac{C_{BL_{l,j},t-1}}{BC_{l,j}}$ 
      if  $NBA_l > lmaxnbap$  then
         $NBA_l \leftarrow lmaxnbap$ 
      end if
    end if
  end if
end if

```

Algorithm 5.3.5 Clear non-processed block applications without object-competition

Input:

- NBA : see Subsection 5.2.1.
- l : an integer number, $1 \leq l \leq N_B$, identifying the thread in which the algorithm is applied.

if $NBA_l = \infty$ **then**
 $NBA_l \leftarrow 0$
end if

Algorithm 5.3.6 Consume block objects

Input:

- C , BC , BL and NBA : see Subsections 5.2.1 and 5.3.1.
- l, j : two integer number, $1 \leq l \leq N_B, 1 \leq j \leq N_\Gamma$, identifying the thread in which the algorithm is applied.

if $NBA_l > 0 \vee BC_{l,j} > 0$ **then**
 $C_{BL_{l,j,t}} \leftarrow C_{BL_{l,j,t-1}} - NBA_l \cdot BC_{l,j}$
end if

Algorithm 5.3.7 Distribute applications among rules

Input:

- BS , BR , $PPROB$, NRA , NBA , SQ and SQB : see Subsections 5.2.1 and 5.3.1.
- l : an integer number, $1 \leq l \leq N_B$, identifying the thread in which the algorithm is applied.

if $NBA_l > 0$ **then**
 $j \leftarrow$ a uniform random integer number in $\{1, \dots, BS_l\}$
 $lrrand \leftarrow B(NBA_l, PPROB_{BR_j})$, where B is the binomial distribution
 $NRA_{BR_j} \leftarrow NRA_{BR_j} + lrrand$ (see Algorithm 5.3.8)
 $NBA_l \leftarrow NBA_l - lrrand$
end if

Algorithm 5.3.8 Calculate binomial distribution

Input:

- j : an integer number $j > 0$ expressing the times to repeat the experiment.
- p : a real number $p \in [0, 1]$ expressing the probability of success for the experiment.

```

 $s \leftarrow 0$ 
for  $i \leftarrow 1$  to  $j$  do
   $k \leftarrow$  a uniform random real number in  $[0, 1]$ 
  if  $k \leq p$  then
     $s \leftarrow s + 1$ 
  end if
end for return  $s$ 

```

Algorithm 5.3.9 Distribute applications among rules maximally

Input:

- BS , BR , NRA and NBA : see Subsections 5.2.1 and 5.3.1
- l : an integer number, $1 \leq l \leq N_B$, identifying the thread in which the algorithm is applied

```

if  $NBA_l > 0$  then
   $j \leftarrow$  a uniform random integer number in  $\{1, \dots, BS_l\}$  ▷ Choose a
  random rule in Block  $B_l$ 
   $NRA_{BR_j} \leftarrow NRA_{BR_j} + NBA_l$ 
   $NBA_j \leftarrow 0$ 
end if

```

Algorithm 5.3.10 Generate objects

Input:

- C , RC , RL and NRA : see Subsections 5.2.1 and 5.3.1
- l, j : two integer number, $1 \leq j \leq N_\Gamma$, $1 \leq l \leq N_R$, identifying the thread in which the algorithm is applied

```

if  $NRA_l > 0$  then
   $C_{RL_l, j, t} \leftarrow C_{RL_l, j, t} + RC_{RL_l, j} \cdot NRA_l$ 
end if

```

random numbers in Algorithm 5.3.8, while in PGPC++ the facilities provided by `std::rand` are directly used. These libraries provide a wide range of functionality to generate and handle random numbers, and are publicly available under open source licenses.

5.4.1 P–Lingua extension

In order to define PGP systems, P–Lingua has been extended to support PGP rules. Specifically, given $f, g \in \Phi$, $u, v \in M(\Gamma)$, $1 \leq i, j \leq q$, $p = p_{\mathcal{R}}(r)$, rules are represented as follows:

$$\begin{aligned} \{f\}[u]_i \xrightarrow{p} [v]_j, &\equiv \text{@guard } f \text{ ?}[u]'i \text{ --> } [v]'j \text{ :: } p; \\ \{f\}[u, f]_i \rightarrow [v, g]_i &\equiv \text{@guard } f \text{ ?}[u, f]'i \text{ --> } [v, g]'i \text{ :: } 1.0; \end{aligned}$$

In both cases, if $p = 1.0$, then $\text{:: } p$ can be omitted. If $i = j$, then $\{f\}[u]_i \xrightarrow{p} [v]_j$ can be written as $\text{@guard } f \text{ ?}[u \text{ --> } v]'i \text{ :: } p$; . Likewise, $\{f\}[u, f]_i \rightarrow [v, g]_i$ can always be written as $\text{@guard } f \text{ ?}[u, f \text{ --> } v, g]'i$; . Moreover, some additional constructs have been included to ease parametrization of P systems. The idea is to enable completely parametric designs, so as experiments can be tuned by simply adjusting parameters, leaving modifications of P–Lingua files for cases in which changes in semantics are in order.

$\&\{multiset\}:\{iterators\}$ In this sentence, *multiset* is an ordinary multiset, whose indexes might depend on the iterators defined in *iterators*. *iterators* is a standard list of iterators in P–Lingua separated by commas. The types of objects generated in the multiset part might depend on the values of the variables defined in *iterators*.

It is worth noting that this sentence has some limitations. For instance, variables defined in these iterators cannot be used again in the same P–Lingua specification. In addition, those variables used in *multiset* which are defined in *iterators* can only be used as such, that is, they cannot be used as subindexes or arithmetical expressions. The reasons for these constraints correspond to technical implementation details which will not be discussed here.

$\text{@mu}(label)*=cell_structure$; In this sentence, *label* is a cell label defined at some point in the P–Lingua specification. *cell_structure* is a standard P–Lingua, tissue–like membrane structure, such as the ones which can be defined after the *@mu* sentence. This sentence adds the skin of *membrane_structure* as a child cell of *label*. As cells in tissue–like structures

have no parent, $label = 0$ for all tissue-like models. In cell-like models, the behaviour is the same, with the exception that *cell.structure* is a cell-like structure, *label* can be any label in the system and the symbol $\ast=$ is replaced by $\ +=$.

`@property(label)=set`; This sentence allows designers to define specific properties for objects. *set* is a set of symbols, which can be extended by external, standard iterators or internal ones as defined at the first point of this list. In the case of PGP systems, `@property(flag)=set` defines flags $f \in \Phi$.

In addition, two new formats have been integrated into P-Lingua. These formats (XML-based and binary) encode P systems representing labels and objects as numbers instead of strings, so they are easily parsed and simulated by third-part simulators such as PGPC++ and PGPCUDA. For the sake of implementation simplicity, these simulators output their results by identifying objects and flags as numbers instead as character strings, but the software GUI described in Subsection 5.4.2 enables automatic translation of these results into others in which labels and objects are represented as strings, as usual.

5.4.2 A graphical environment for PGP systems

A new GUI named *MeCoGUI* has been developed for the simulation of PGP systems. MeCoSim [172] could have been used instead. However, in the environment in which the simulators were developed there exist some pros and cons on this approach versus an ad-hoc simulator.

MeCoSim is an integrated development environment (IDE). That is to say, it provides all functionality required for the simulation and computational analysis of P systems. To define the desired input and output screens, it is necessary to configure a spreadsheet by using an *ad-hoc* programming language. However, it would entail teaching this language to prospective users, which are proficient in R programming language [14, 76] instead. In this sense, a more natural approach for them is to develop a GUI in which users can define input parameters and results analysis on R.

To do so, the developed GUI takes as input a P system file on P-Lingua format and a CSV file encoding its parameters, and outputs a CSV file which contains simulation results. This way, users can define inputs and analyse outputs on the programming language of their choice. CSV is a widespread, simple and free format with plenty of libraries for different languages. This flexibility comes at the cost concerning that the developed GUI is not an IDE, as input

parameters and simulation analysis cannot be directly input and viewed on the GUI. Rather, it is necessary to develop applications to generate and process these CSV files which depend on the domain of use. In some simulators (such as PGPC++ and PGPCUDA), the output CSV files represent labels and objects as integers, but this application includes a button to translate output files from PGPC++ and PGPCUDA into string-representative file formats. Figure 5.4 displays the main screen of this application.

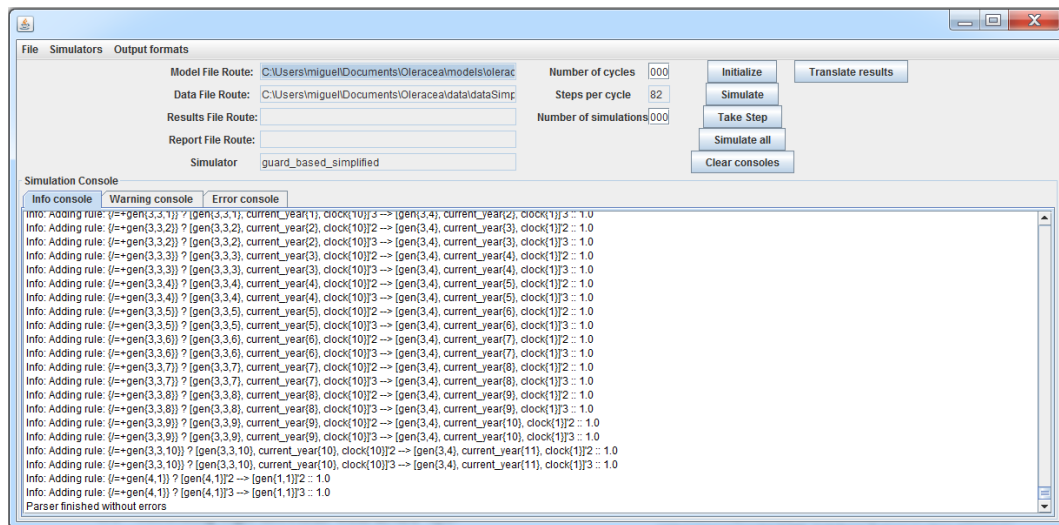


Figure 5.4: Main screen of MeCoGUI

MeCoGUI can also translate P systems into machine-readable formats, such as those read by PGPC++ and PGPCUDA. Finally, it is important to remark that these applications play the role of domain-specific spreadsheets on MeCoSim, so MeCoGUI can simulate any type of P system supported by P-Lingua. This is because only external applications for input data and simulation processing depend on the domain, not MeCoGUI itself, which is general for any type of P system. Figure Figure 5.5 graphically describes the workflow for P-Lingua and for PGPC++ and PGPCUDA.

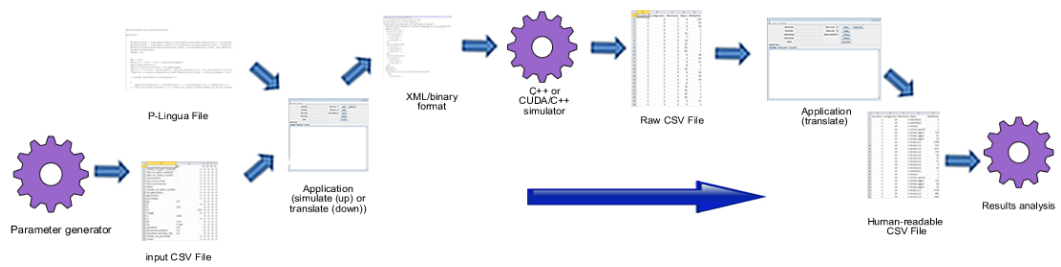


Figure 5.5: Workflow for P-Lingua simulator (upper branch) and PGPC++ and PGPCUDA (lower branch) for MeCoGUI

Part III

Results

Chapter 6

Case studies

This chapter discusses two case studies on the application of the models previously described. The phenomena studied are:

- An application of LNDP systems on genomic data about the flowering process of *Arabidopsis thaliana*. In this application, we will instantiate a model of LNDP systems for a specific gene regulatory network from *Arabidopsis thaliana*. The results will be contrasted with those from the improved Logic Analysis of Phylogenetic Profiles method. This will be introduced in Section 6.1.
- An application of PGP systems on experimental data about conservation trends of the white cabbage butterfly (*Pieris napi oleracea*). In this application, the parameters are directly given by experts in the species, who have validated the model as well. This will be introduced in Section 6.2.

6.1 Modelling logic networks with LNDP systems: *Arabidopsis thaliana*, a case study

The first case study models the behaviour of a Logic Network on the flowering process of *Arabidopsis thaliana*, indicating the parameters and outputting data obtained from the simulation of such a model. It is important to recall that the aim of LNDP systems is to reproduce the behaviour of the improved LAPP method [229], rather than be validated on field data.

6.1.1 A Logic Network on *Arabidopsis thaliana* flowering processes

Arabidopsis is a long-day botanic genre whose genetic and protein interaction networks are widely studied due to its genetic resemblance with rice and its implications on transgenic crops [207, 120]. Zhang and Zuo stated that these conditions can promote reproductive growth and induce early flowering [237]. However, short-day conditions can promote vegetative growth and induce late flowering or even no-flowering. To understand the intrinsic mechanisms of *Arabidopsis* flowering in different lighting conditions, the relationships of related genes need to be compared.

In the past ten years, much work has been reported in the field about *A. thaliana* flowering. Imaizumi *et al.* found that FKF1 is a blue light receptor which regulates flowering [110]. Later, they also showed that FKF1 together with Flavin-Binding and Kelch Repeat degrade Cycling Dof Factor1 (CDF1) to eventually control CO [109]. In the same year, Abe *et al.* found that Flowering Locus T (FT) together with FD activate Apetala1 (AP1) to initiate floral development and promote floral transition at the shoot apex [18]. Previous work deals only with one or few genes related to flowering. However, the networks considered in this work focus on the relationships among a large number of genes systematically. Bowers *et al.* proposed the Logic Analysis of Phylogenetic Profiles (LAPP) [25]. This method helps researchers to know biological functions of some genes or proteins on the basis of phylogenetic profiles, and has been developed both on theory and application [26, 238, 228]. For example, Wang *et al.* developed the improved LAPP method, and reversely constructed a logic network of sixteen genes in shoot for *Arabidopsis* under salt stimuli [228].

6.1.2 A case study on *Arabidopsis thaliana*

This model is experimentally verified on a logic network which regulates flowering processes associated with *Arabidopsis thaliana* on a long-day scenario. This relatively large network integrates gene interaction samples from NCBI/EBI database [9]. This logic network has been constructed according to the procedure described by Bowers *et al.* [25]. The total number of genes in the network is 29, whereas the total number of interactions is 99 (23 unary and 76 binary). Therefore, only a few different types of all possible interactions collected in Subsection 4.2.1 are present in this network. Only unary strong promotion and inhibition binary *AND*-like and *OR*-like interactions

are present. The vast majority of these interactions are *AND*-like with both inputs and result in non-negated form (that is, $G'_j = G_j$, $G'_k = G_k$ and $G'_l = G_l$).

Gene network data is provided in Appendix A. Specifically, gene initial states are reflected in Figure A.1 from this appendix. Unary and binary interactions are reflected in Figure A.2 and in Figures A.3, and A.4, respectively.

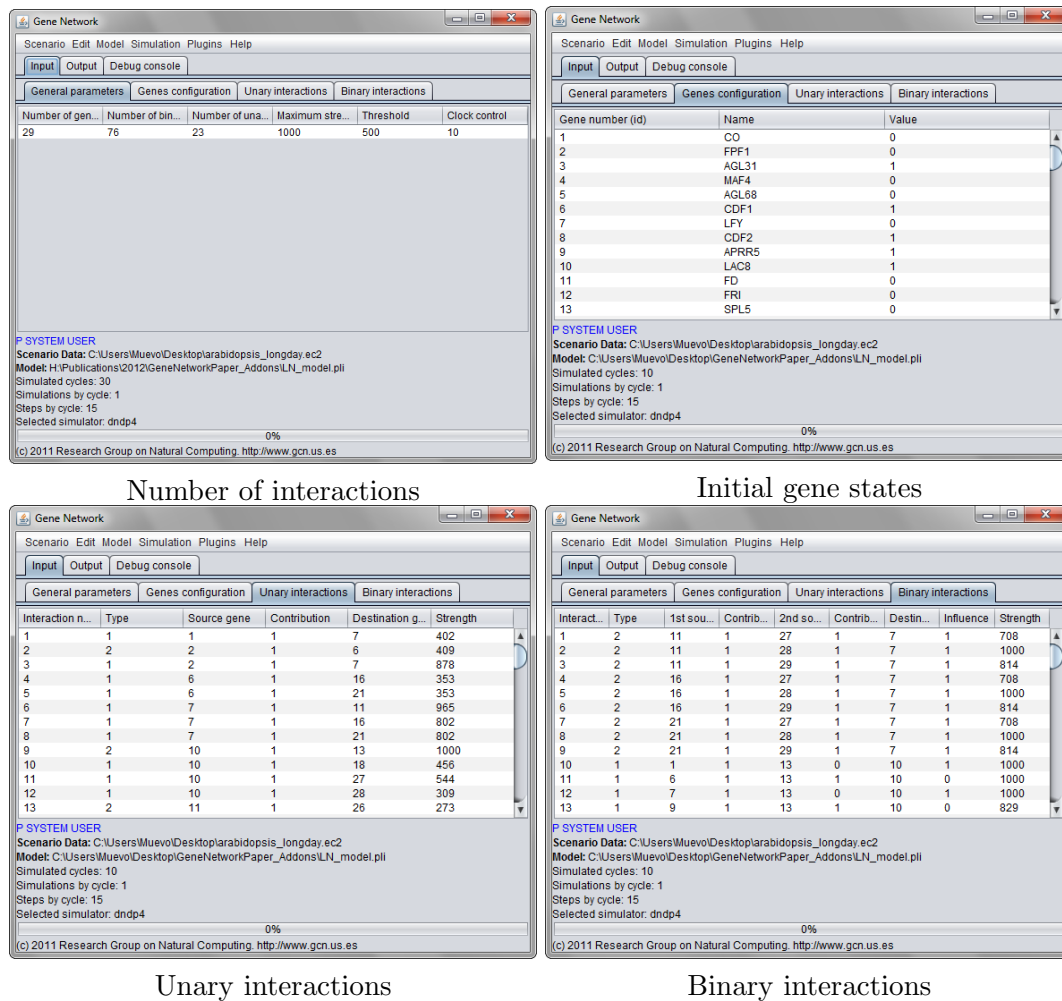


Figure 6.1: Input Data on MeCoSim interface

To verify the behaviour of the model on this scenario, the improved LAPP method (as presented in Wang et al. [229]) has been run for 30 steps on this data. Similarly, the LNDP model has been simulated for 30 cycles. Each cycle in an LNDP system consists on 15 computation steps, so the total number

Cycle	Gene	Value (0: disabled; 1: enabled)
30	1	0
30	2	0
30	3	1
30	4	0
30	5	0
30	6	1
30	7	0
30	8	1
30	9	1
30	10	1

P SYSTEM USER
Scenario Data: C:\Users\Muevo\Desktop\larabidopsis_longday.ec2
Model: H:\Publications\2012\GeneNetworkPaper_Addons\LN_model.pli
 Simulated cycles: 30
 Simulations by cycle: 1
 Steps by cycle: 15
 Selected simulator: dndp4

0%

(c) 2011 Research Group on Natural Computing. <http://www.gcn.us.es>

Final gene states

Figure 6.2: Simulation Results from MeCoSim interface

of steps simulated in the model is $30 \times 15 = 450$. LNBP model simulations were carried out by using MeCoSim [172], thus easing parameter input and outcome analysis. Figures 6.1 and 6.2 display the MeCoSim input tables used and simulation results obtained in this case study. Due to the table structure featured by MeCoSim to define parameters, the network data was easily fed into the application, as well as straightforwardly copied and pasted into commercial spreadsheets. The results match the ones obtained from the execution of the improved LAPP method on the same input data, verifying that, on this gene network and scenario, the P system model behaviour is analogous to that from the improved LAPP method.

6.2 A PGP model on the ecosystem of *Pieris napi oleracea*

This section introduces a computational model based on PGP systems on the behaviour and conservation trends of *Pieris napi oleracea* (*P. n. oleracea*, for short). This species, commonly known as mustard white butterfly, is native from eastern North America. The aim of the model is to predict population growth trends and evolutionary responses of two genotypes and three genotypes of this species, with the application of updating its conservation status in accordance with the model simulation results.

6.2.1 Ecology of the species

As described by Keeler and Chew [118], *P. n. oleracea* is generally bivoltine (i.e., has two generations per year), and the first adults emerging in early May [159]. Females laying period extends from 3 to 15 days after emergence, and oviposit on native crucifers such as toothwort *Cardamine diphylla* [42], but also on bolting plants of the invasive species garlic mustard (*Alliaria petiolata*). Eggs hatch in 5–7 days and larvae develop through five instars on the food plant selected by the mother, although they are highly likely to survive to adulthood upon reaching the 3rd instar and can migrate among plants when its carrying capacity reaches critical levels [118], upon food source depletion or when they encounter offspring from different females [57]. *P. n. oleracea* are subjected to parasitism from parasitoid wasp *Cotesia glomerata* L., whose population is low during April and May due to over-winter mortality [21], but abundant during the second *P. n. oleracea* generation [21, 20]. Pupae develop into adults within 7 days or enter pupal diapause (an hibernation-like state), and emerge as adults the following spring. Diapause is highly labile and phenotypically variable [231]. The second generation hatches in July and develops similarly to the first, but most pupae from the second generation enter diapause. A partial third generation is sometimes seen in September, when there are sufficient host plants and favourable environmental conditions [42, 159].

6.2.2 Case study

Since 1880's, this species was thought to be seriously endangered, mainly due to top-down [147] (introduction of *Cotesia glomerata* L., a parasitoid wasp) and bottom-up [95] (invasion of *A. petiolata*) processes. Informally speaking, bottom-up refers to effects caused by processes or factors working at trophic

levels below that of the focal species; whereas top-down refers to factors or processes working at trophic levels above the focal species' one [183, 106, 107]. The nature of both perturbations in the ecosystem is different; *C. glomerata* was intentionally introduced in 1880 to control the spread of the white cabbage butterfly (*Pieris rapae*) [19], an exotic invasive species which entered the US in the 1860's via Canada, whereas the introduction of *A. petiolata* in 1868 from Europe to Washington D.C. and in mid 1900s to Massachusetts is thought to be merely accidental [156].

The effects of both species on population levels of *P. n. oleracea* are also dissimilar. *C. glomerata* individuals attacks larvae of *P. n. oleracea* by ovipositing into their bodies. Upon hatching, *C. glomerata* larvae eat their way out through the infected *P. n. oleracea* larva before transforming into adult individuals. The image of Ridley Scott's Alien [206] bursting out of the infected host serves as an illustrative example of this process. On the contrary, the effect of *A. petiolata* on the species is more complex. On the one hand, the area invaded by *A. petiolata* overlaps *C. diphylla*'s, thus ravaging the natural habitat and food source of *P. n. oleracea* [42]. On the other hand, *A. petiolata* lures *P. n. oleracea* adults into laying eggs on it, due to some shared components of the glucosinolate profile of *C. diphylla* [104, 197]. However, upon hatching, larvae do not thrive due to deterrent agents *isovitexin-6'- β -glucopyranoside* and *alliarinoside* [98, 198].

The resulting evolutionary pressure on *P. n. oleracea* might give place to two complementary strategies: a possible scenario is that *P. n. oleracea* adults tend to avoid ovipositing on *A. petiolata*, whilst in a different path larvae develop tolerance to deterrents and/or possible toxins [117]. Keeler and Chew [117] surprisingly found a positive correlation between oviposition preference for *A. petiolata* and larval offspring that were able to develop on this novel host plant. The proposed model intends to shed light on evolutionary trends of *P. n. oleracea* and their strategies to cope with environmental disturbances in their habitat. Data has been obtained first-hand by experts in the species, as well as from stochastic models based on difference equations [118].

6.2.3 PGP systems as Individual Based Models

Ecological modelling is a mature discipline which studies how to reflect the properties of an ecosystem into a formal model capable of being simulated (or implemented, when possible) by computers. Traditional approaches in the discipline include differential equations, especially well-studied Ordinary Differential Equation (*ODEs*) models known as *Lotka-Volterra* systems [123]. To

account for the inherent randomness of ecosystems, new models in which events are ruled by probabilities were introduced, which are known as stochastic models [114]. One of such approaches are *Individual Based Models* or IBMs [115]. These kind of models directly reflect the behaviour and interactions of individuals in a system, rather than higher level abstractions in which subtle though important details are omitted.

Although some prototypical models were introduced in the 1950's decade [125], the first IBM models date back to mid-1970's [142] and some authors such as Grimm recognize Kaiser *et al.* as pioneers on IBM modelling, as they introduced novel formalisms which break away from classical approaches [115]. From then on, this fine-grained approach has been embraced by ecologists and applied to a eclectic assortment of ecosystems. Grimm [86] argues that the fascination of ecologists with IBMs stems from the fact that they allow them to forget about high-level, abstract representations of reality and cram up all knowledge they have from the system under study in the model. As an example, he describes models of small mammals by Halle and Halle [97] in which local, asynchronous interactions between individuals prove to be key aspects on the species survival which tend to be overlooked by classical approaches. Nevertheless, he warns that, even though at first sight IBMs might appear intuitive and natural, the expert modeller must learn which aspects lay aside from the model and which ones consider. As a matter of fact, he distinguishes between pragmatic models (those which provide a level of detail hard to achieve with classical frameworks) and paradigmatic models (those which share inherent characteristics, such as discreteness, closer to biological reality than classical approaches). Finally, he concludes that IBM modellers must be aware of keeping general perspective on the model, and maintaining in the iteratively designed model evolution patterns which can be validated by contrasting historical data.

Usually, IBMs provide solutions where classical models are no longer valid. Judson [114] and Grimm *et al.* [87] argue that, in contrast to other scientific disciplines (say nuclear physics or mathematics), biology lacks well-known natural laws, and in IBMs rules of thumb and intuition trained by long years of study fill the gap. Judson also acknowledges the role of stochastic methods in capturing ecological indeterminism, especially when rare events are paramount to understand the system's dynamics [114]. However, this proximity to the ecologists' manner of understanding the system under study comes at a cost; Lomnicki [127] remarks that IBMs introduce computational limitations not present in classical approaches: IBMs handle a considerably larger number of input parameters than their traditional counterparts, which takes its toll as a

greater computing power required. On the other hand, he acknowledges that fine-grained, metapopulation-related spatial patterns such as local extinction and emigration are easier to capture, and mentions the case of an IBM on flour beetles capable of reflecting a chaotic, ecologically-significant behaviour at a level almost impossible to attain with classical, analytical models without resorting to confuse and intricate equations [55].

PGP systems can be considered as a particular case in Individual Based Models. This classification is important because it makes them susceptible for design and analysis procedures explained by Grimm [86], consequently regarding PGP systems to be subjected to a well-known, consolidated methodology.

6.2.4 A PGP-based model on the ecosystem of *Pieris napi oleracea*

Here the model for the dynamics of *Pieris napi oleacea* is presented. This model captures the behaviour of the species in several stages, from egg to butterfly. Only female individuals are considered. In the model, two plants (*Cardamine dihylla* and *Alliaria petiolata*) are considered. These plants are numbered as 1 and 2, respectively. Moreover, each butterfly is of one of the following genotypes: heterozygous (type 1 or Rr), homozygous dominant (type 2 or RR) and homozygous recessive (type 3 or rr). Larvae undergo five instars prior to adulthood; however, mortality in instar 4 is negligible, i.e., larvae entering the 4th instar almost always enter the 5th, so it can be omitted [117]. Therefore, in the model only 4 instars are considered, being instar 5 represented as if it was instar 4. Mortality of larvae varies throughout the years [43], so the current year needs to be taken into account in the model. Although *Pieris napi oleracea* populations are usually bivoltine, due to ecological availability of a second non-native *Brassicaceae*, the cuckoo flower (*Cardamine pratensis*), some individuals comprise a third generation per year [117]. Therefore, 3 generations per year are considered in the proposed model.

The dynamics of the model are composed of four stages which are repeated cyclically. Stage 1 models butterfly emergence from pupae and potential immigration or emigration from the local population. Stage 2 models parasitism. Stage 3 models larval migration between host plants (due to depletion of an individual plant). Finally, Stage 4 models larva migration and transformation into pupa, as well as pupa diapause. This model consists of a PGP system of degree $plants + 1$, where $plants$ denote the number of types of plant in the system. Thus, the system is composed of one environment for each type of

plant plus an auxiliary environment for butterfly redistribution. Some of the parameters are taken from [118] and [117], whereas others have been directly provided by experts. Migrating butterflies parameters (F_y) have been adjusted experimentally for the model, so as to emulate carrying capacity and clumping factor constraints. Finally, the parameters synchronizing the model (nm , nls and nc) have been adjusted so that the modelled ecosystem phenomena count with enough simulation cycles to take place. All parameters used on the model are described in Table ???. The model consists on a PGP system defined as follows:

$$\Pi = (\Gamma, \Phi, \mathcal{R}, \mathcal{G}_{\mathcal{R}}, p_{\mathcal{R}}, (f_1, \mathcal{M}_1), \dots, (f_q, \mathcal{M}_q))$$

where:

- G_R is a directed graph containing a node for each plant, plus one for butterfly distribution. The current model only considers *Cardamine diphylla* and *Alliaria petiolata*, so the degree of the system is 3.
- In the working alphabet Γ , *P. n. oleracea* individuals are represented as outlined below. The meaning of nge , nin , ng , cl , nls and ny is explained in Table ???.

$$\Gamma = \{but_g, butl_g, butla_{i,g}, egg_g, pupa_g, pupad_i, 1 \leq g \leq nge\} \cup \\ \{larva_{in,i,g}, 1 \leq in \leq nin, 1 \leq g \leq nge, 1 \leq i \leq ng\} \cup \\ \{cl_j, 1 \leq j \leq cl\} \cup \{cy_k, 1 \leq k \leq ny\} \cup \{dist, rst\}$$

- Objects $but_g, butl_g, butla_{i,g}, egg_g$ and $pupa_g, 1 \leq g \leq nge, 1 \leq i \leq ng$, represent butterflies, eggs and pupae of Phenotype g respectively. Objects $butl_g$ and $butla_{i,g}$ represent ovipositing females and those which have already oviposited, respectively, and Objects $pupad_g$ represent pupae which have entered diapause.
- Objects $larva_{in,i,g} : 1 \leq in \leq nin, 1 \leq g \leq nge, 1 \leq i \leq ng$, represent larvae in instar in , phenotype g and generation i .
- Objects $cl_j, 1 \leq j \leq nls$, mark the time necessary for each development stage to be completed. It is noteworthy that the development of exemplars from larva to pupa takes considerably more time than the other stages; so this stage is carried out by more transition steps than the others.
- Objects $cy_k, 1 \leq k \leq ny$, are objects indicating the current year simulated.
- Object $dist$ triggers the redistribution of butterflies among plants. This distribution pattern varies among generations.

- Object rst restarts the clock for Stage 2.
- Alphabet Φ is composed of flags representing the current stage as outlined below.

$$\Gamma = \{gen_{i,j}, 1 \leq i \leq ng + 1, 1 \leq j \leq ns + 1, j \neq 2\} \cup \\ \{gen_{i,2,k}, 1 \leq i \leq ng + 1, 1 \leq k \leq years\} \cup \\ \{dist_i, iss_i, wait_i, 1 \leq i \leq ng\}$$

- Flags $gen_{i,j}, 1 \leq i \leq ng + 1, 1 \leq j \leq ns + 1, j \notin \{2, 3\}$, represent the current generation and stage in the model, eggs and pupae of phenotype g , respectively. Objects $gen_{i,2,y}$ and $gen_{i,3,y}, 1 \leq i \leq ng + 1, 1 \leq y \leq ny$, are represented separately, as Stages 2 and 3 depend on the current year of simulation.
- Flags $dist_i, iss_i$ and $wait_i, 1 \leq i \leq ng$, denote the time in which butterflies are redistributed from cell 1 to the rest of the cells. Similarly, objects $iss_i, 1 \leq i \leq ng$, indicate the time for butterflies to migrate to cells $j, 1 < j \leq q$, for redistribution. Finally, Objects $wait_i, 1 \leq i \leq ng$, indicate cells $j, 1 < j \leq q$, to issue a signal $dist$ to cell 1 so that it prepares for butterfly redistribution.
- Steps per stage have been adjusted so each one has a reasonable time to be simulated. Therefore, Stages 2 and 3 are given 3 steps each, whereas stage 1 takes considerably longer and therefore is given 10 steps. Including offset steps for transition between stages, the simulation of each year takes 70 steps.
- $\mathcal{G}_{\mathcal{R}} = []_1, []_2, \dots, []_{np+1}$ is the cell structure.
- The initial multisets are:
 - $\mathcal{M}_1 = \{but_g^{N_g}\}, 1 \leq g \leq nge$, that is, cell 1 contains an initial number of butterflies.
 - $\mathcal{M}_k = \{cl_1, cy_1\}, 2 \leq k \leq np + 1$, that is, the rest of the cells contain the initial year and clock cycle.
- The initial flags are:
 - $f_1 = dist_1$
 - $f_k = gen_{1,1}, 2 \leq k \leq np + 1$

- The rules of \mathcal{R} to apply are the following. First, those directly related with processes regarding *P. n. oleracea* life-cycle are described, followed by those which perform synchronization aspects. For the sake of simplicity, those probabilities which are equal to 1 are omitted:

$$\{dist_i\} \quad [but_g]_1 \xrightarrow{Prop^{i,k-1}} [but_g]_k \quad \begin{cases} 1 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \end{cases}$$

- First, butterflies are distributed among plants.

$$\{dist_i\} [dist^{np}, dist_i \rightarrow dist_{i+1}]_1, 1 \leq i \leq ng$$

- Then, the distribution flag is updated.

$$\begin{array}{l} \{gen_{i,1}\} \quad [but_3 \xrightarrow{R} but_1]_k \\ \{gen_{i,1}\} \quad [but_3 \xrightarrow{1-R} but_3]_k \end{array} \quad \begin{cases} 1 \leq i \leq ng \\ 2 \leq k \leq np + 1 \end{cases}$$

- Some homozygous butterflies with phenotype *rr* might spontaneously become heterozygous with probability *R*.

$$\begin{array}{l} \{gen_{i,2}\} \quad [larva_{in,i,g} \xrightarrow{(1-\omega)} [larva_{in,i,g}]_k \\ \{gen_{i,2}\} \quad [larva_{in,i,g}]_k \xrightarrow{\frac{\omega \cdot Se}{nR}} [larva_{in+1,i,g}]_j \\ \{gen_{i,2}\} \quad [larva_{in,i,g} \xrightarrow{\omega \cdot (1-Se)}]_k \end{array} \quad \begin{cases} 1 \leq in \leq nin \\ 1 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k, j \leq np + 1 \\ k \neq j \end{cases}$$

- Larvae migrate among systems with probability ω . Some might die with probability *Se*.

$$\begin{array}{l} \{gen_{i,2,y}\} \quad [but_g \xrightarrow{(1-Fy) \cdot D} butl_g]_k \\ \{gen_{i,2,y}\} \quad [but_g \xrightarrow{1-(1-Fy) \cdot D}]_k \end{array} \quad \begin{cases} 1 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \\ 1 \leq y \leq ny \end{cases}$$

- Butterflies might leave the system with probability F_y . Also, butterflies might lay eggs with probability *D*.

$$\begin{array}{l}
\{gen_{i,2,y}\} \quad [butl_1 \xrightarrow{(1-F_1) \cdot D} butla_{i,1}, egg_1^{\frac{Ef_i}{2}}, egg_2^{\frac{Ef_i \cdot p_i}{2}}, egg_3^{\frac{Ef_i \cdot (1-p_i)}{2}}]_k \\
\{gen_{i,2,y}\} \quad [but_2 \xrightarrow{(1-F_2) \cdot D} butla_{i,2}, egg_1^{Ef_i \cdot (1-p_i)}, egg_2^{Ef_i \cdot p_i}]_k \\
\{gen_{i,2,y}\} \quad [but_3 \xrightarrow{(1-F_3) \cdot D} butla_{i,3}, egg_1^{Ef_i \cdot p_i}, egg_3^{Ef_i \cdot (1-p_i)}]_k
\end{array} \quad \left\{ \begin{array}{l} 1 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \\ 1 \leq y \leq ny \end{array} \right.$$

- Butterflies oviposit and die according to their phenotype. For each one which has oviposit, a marking object *butla* is left.

$$\begin{array}{l}
\{gen_{1,2,y}\} \quad [egg_g \xrightarrow{H_1} larva_{1,1,g}]_k \\
\{gen_{1,2,y}\} \quad [egg_g \xrightarrow{1-H_1}]_k \\
\{gen_{i,2,y}\} \quad [egg_g \xrightarrow{H_2 \cdot Hat_{k-1,g}} larva_{1,2,g}]_k \\
\{gen_{i,2,y}\} \quad [egg_g \xrightarrow{1-H_2 \cdot Hat_{k-1,g}}]_k
\end{array} \quad \left\{ \begin{array}{l} 2 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \\ 1 \leq y \leq ny \end{array} \right.$$

- Eggs hatch with a probability according to their generation.

$$\begin{array}{l}
\{gen_{i,3,y}\} \quad [larva_{in,i,g} \xrightarrow{P_{y,in,i} \cdot Det_{k-1,g}} larva_{in+1,i,g}]_k \\
\{gen_{i,3,y}\} \quad [larva_{in,i,g} \xrightarrow{1-P_{y,in,i} \cdot Det_{k-1,g}}]_k
\end{array} \quad \left\{ \begin{array}{l} 1 \leq in \leq npin - 1 \\ 2 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \\ 1 \leq y \leq years \end{array} \right.$$

- Larvae are subjected to parasitism in the second stage. Those which survive reach the next instar.

$$\begin{array}{l}
\{gen_{i,4}\} \quad [larva_{nin,i,g} \xrightarrow{U} pupa_g]_k \\
\{gen_{i,4}\} \quad [larva_{nin,i,g} \xrightarrow{1-U}]_k
\end{array} \quad \left\{ \begin{array}{l} 1 \leq in \leq nin - 1 \\ 2 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \end{array} \right.$$

- Larvae which have survived to reach instar *nin* develop into pupae with probability *U*. Those which have not die instead.

$$\begin{array}{l}
\{gen_{i,4}\} [pupa_g \xrightarrow{(1-O_i) \cdot M_i} but_g]_k \\
\{gen_{i,4}\} [pupa_g \xrightarrow{(1-O_i)}]_k \\
\{gen_{i,4}\} [pupa_g \xrightarrow{O_i} pupad_g]_k
\end{array} \quad \left\{ \begin{array}{l} 2 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \end{array} \right.$$

- Pupae emerge as butterflies, die or enter diapause.

$$\begin{array}{l} \{gen_{ng+1,1}\}[but_g \rightarrow]_k \\ \{gen_{ng+1,1}\}[egg_g \rightarrow]_k \\ \{gen_{ng+1,1}\}[pupa_g \rightarrow]_k \\ \{gen_{ng+1,1}\}[but_g \rightarrow]_k \\ \{gen_{ng+1,1}\}[butl_g \rightarrow]_k \\ \{gen_{ng+1,1}\}[butla_{i,g} \rightarrow]_k \end{array} \quad \left\{ \begin{array}{l} 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \\ 1 \leq i \leq ng \end{array} \right.$$

- Only diapausing pupae can survive overwinter. Therefore, butterflies, pupae (not overwintering) and eggs die.

$$\begin{array}{l} \{gen_{ng+1,1}\}[pupad_g \xrightarrow{Sw} but_g]_k \\ \{gen_{ng+1,1}\}[pupad_g \xrightarrow{1-Sw}]_k \end{array} \quad \left\{ \begin{array}{l} 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \end{array} \right.$$

- Pupae which overwinter become butterflies with probability Sw .

$$\{iss_i\} [but_g]_k \rightarrow [but_g]_1 \quad \left\{ \begin{array}{l} 1 \leq i \leq ng \\ 1 \leq g \leq nge \\ 2 \leq k \leq np + 1 \end{array} \right.$$

- On each generation transition, butterflies are redistributed.

$$\begin{array}{l} \{iss_i\}[iss_i \rightarrow dist, wait_i]_k \\ \{wait_i\}[dist]_k \rightarrow [dist]_1 \\ \{wait_i\}[cl_c \rightarrow cl_{c+1}]_k \\ \{wait_i\}[cl_{nc}, wait_i \rightarrow cl_1, gen_{i+1,1}]_k \\ \{gen_{i,j}\}[cl_c \rightarrow cl_{c+1}]_k \\ \{gen_{i,1}\}[cl_l \rightarrow cl_{l+1}]_k \\ \{gen_{i,2,y}\}[cl_h \rightarrow cl_{h+1}]_k \\ \{gen_{i,3,y}\}[cl_c \rightarrow cl_{c+1}]_k \\ \{gen_{i,j}\}[cl_{nc}, gen_{i,j} \rightarrow cl_1, gen_{i,j+1}]_k \\ \{gen_{i,1}\}[cl_{nm}, gen_{i,1} \rightarrow gen_{i,2}]_k \\ \{gen_{i,2}\}[cl_{nls}, gen_{i,1} \rightarrow gen_{i,3}]_k \\ \{gen_{i,3}\}[cy_y, gen_{i,2} \rightarrow cl_1, rst, gen_{i,3,y}]_k \\ \{gen_{i,2,y}\}[rst \rightarrow cl_1]_k \\ \{gen_{i,3,y}\}[rst \rightarrow cl_1]_k \\ \{gen_{ip,2,y}\}[cl_{nc}, gen_{ip,2,y} \rightarrow cl_1, gen_{ip,4}]_k \\ \{gen_{ng,3,y}\}[cy_y, cl_{nc} \rightarrow cy_y, cl_1, gen_{ng,3}]_k \\ \{gen_{ip,2,y}\}[cl_{nls}, gen_{ip,2,y} \rightarrow cl_1, gen_{ip,2}]_k \\ \{gen_{ng,3,y}\}[cy_y, cl_{nls} \rightarrow cy_{y+1}, cl_1, gen_{ng,3}]_k \\ \{gen_{ng+1,1}\}[gen_{ng+1,1} \rightarrow gen_{1,1}]_k \end{array} \quad \left\{ \begin{array}{l} 1 \leq l < nm \\ 1 \leq c < nc \\ 1 \leq h < nls \\ 3 < j \leq ns \\ 1 \leq ip < ng \\ 1 \leq y < ny \\ 1 \leq i \leq ng \\ 2 \leq k \leq np + 1 \end{array} \right.$$

- Additionally, some rules deal with synchronization. These rules synchronize butterfly redistribution and update the current clock, generation and year.

Parameter	Description
General parameters	
ng	Number of generations considered
np	Number of plants considered
nge	Number of genotypes considered
nin	Number of instars considered
$npin$	Number of instars subjected to parasitism
ny	Number of years simulated
ns	Number of stages
nm	Number of cycles for Stage 1
nls	Number of cycles for Stage 2
nc	Number of cycles for Stages 3 and 4
Initial state	
N_g	Initial butterflies of phenotype g
Butterfly behaviour parameters	
$Prop_{i,k}$	Proportion of butterflies in plant k in generation i
R	Proportion homozygous butterflies becoming heterozygous
F_y	Proportion of migrating butterflies in year y
D	Proportion of ovipositing butterflies
p_i	Proportion of recessive homozygous butterflies in generation i
Ef_i	Number of eggs of genotype i per female
ω	Proportion of larvae migrating
H_i	Hatching success for generation i
$Hat_{k,g}$	Hatching success in plant k and genotype g
Se	Proportion of larvae dying during migration
$P_{y,in,i}$	Larvae mortality due to parasitism in year y , instar in and generation i
$Det_{k,g}$	Larvae adaptation of genotype g to plant k , instar in and generation i
U	Probability for larvae of becoming pupae upon reaching the last instar
O_i	Proportion of pupae entering diapause in generation i
M_i	Proportion of pupae which emerge successfully in generation i
Sw	Proportion of pupae surviving overwinter

Table 6.1: Parameters for *Pieris napi oleracea* model

6.2.5 Results obtained

The model was simulated using PGPC++ 50 times for 10 years. As it can be seen in Figure 6.4, the standard deviation stabilized, which was the criteria used for experts to decide that no more simulations were necessary. The results were analysed with a script coded in R language, measuring the average and

standard deviation for the number of butterflies which lay eggs each year.

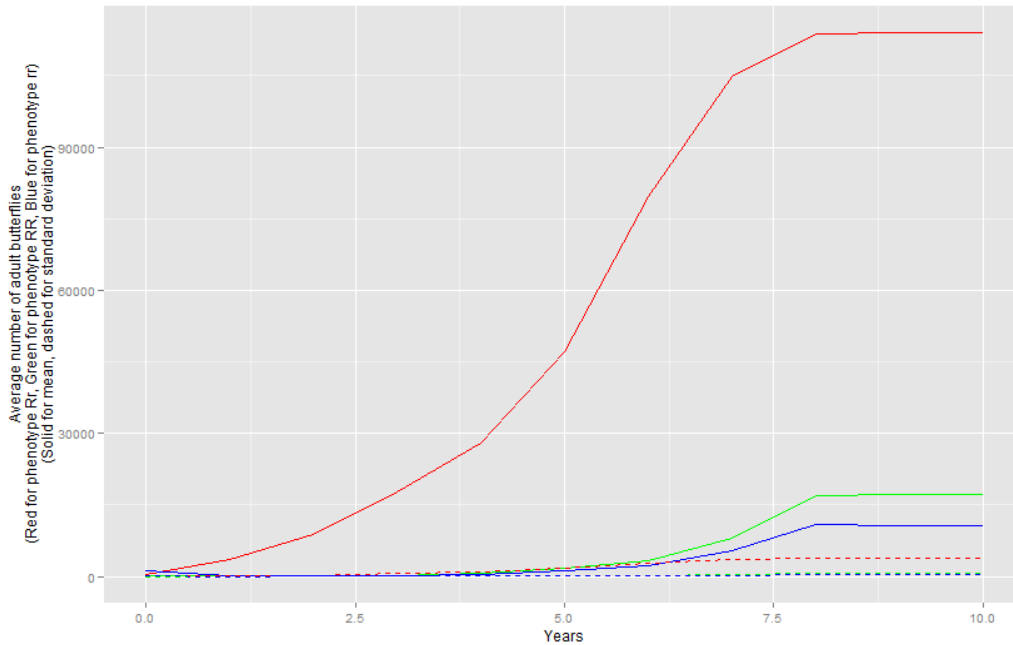


Figure 6.3: Values predicted by the simulator. Solid and dashed lines represent average population levels and typical deviations (in individuals) among simulations, respectively. Red lines display values for phenotype Rr , green lines those of RR and blue lines those of rr .

Figure 6.4 displays some qualitative desirable properties for the system at hand, which were used by the experts who provided the parameters to positively validate the model. For instance, butterflies of phenotype Rr are considered to perform substantially better than their RR and rr counterparts. In addition, butterflies of phenotype RR perform better than those of rr . Therefore, although the initial number of individuals of phenotype rr is rather large in comparison with those of genotypes Rr and RR , these butterflies quickly grow out of rr exemplars, which are worse suited to adapt to *A. petiolata* plants [43]. The large difference between population levels between genotypes RR and Rr can be explained because homozygous butterflies can mutate to become heterozygous, but the reversal process is rare enough to be discarded from the model [117]. Moreover, *C. glomerata* species is in turn parasitoid by another parasitoid wasp (*C. rubecula*), which competitively excludes *C. glomerata* and thus lightens the predatory pressure from *C. oleracea* individuals and explains its overall population growth over time [99]. Finally, population levels are smoothed as they approach 100000 individuals, which is a key figure in which carrying capacity effects take place [118], i.e., the ecosystem cannot

support any more individuals due to scarcity of resources and, consequently, fetters their proliferation.

6.2.6 Performance Analysis

A performance analysis has been conducted with the studied model to measure the acceleration gained by PGPCUDA in comparison with both PGPC++ and P-Lingua. In all scenarios, the model was simulated for 10 years varying the number of simulations per instance. Although 50 simulations is a figure big enough to obtain knowledge about the system, the idea in this analysis is to assess the performance gain of the simulator as the number of simulations grows. In fact, the *P. n. oleracea* model was merely chosen as a real-case scenario, with no intent of being an archetypal benchmark for PGPCUDA.

The charts clearly display that, contrarily to what it would be expected, PGPC++ outperforms PGPCUDA from the very beginning. PGPCUDA simulation runtime is quickly saturated, and no runs beyond 150 simulations could be executed due to errors on the target machine (simulation reports were not properly written on disc). In this sense, in Chapter 7, Section 7.3 some guidelines to improve the performance of PGPCUDA are outlined as future work. Regarding P-Lingua, Simulation times in Java are absent beyond 20 simulations; MeCoGUI crashed on runs above this number, probably due to the resource requirements offset imposed by the Java Virtual Machine (JVM) [6]. Specifically, the application was unresponsive and, eventually, turned black and stopped working. The P-Lingua simulator is an adequate tool to simulate PGP systems for a small number of simulations, with the idea to test the evolution of the P system designed. However, this simulator proved to be an unsuitable alternative to massively simulate PGP systems, so their simulation times are omitted from the charts. Figure 6.5 displays the simulation times and acceleration factors obtained without considering memory transference code. It can be seen that both simulation time lines grow in parallel; both display a straight line with only a small peak for 100 simulations, which can be attributed to noise. The acceleration remains modest in all simulations, but it also can be seen that it increases throughout time. A performance analysis by using NVIDIA Nsight Profiler [10] revealed only a 3% occupation on the target device, so resource exhaustion does not seem to be the problem. However, it is worth recalling that the device of choice was not a proper High Performance Computing server (at the time of the experiment there was no computer of such characteristics compatible with the tools used for development). There-

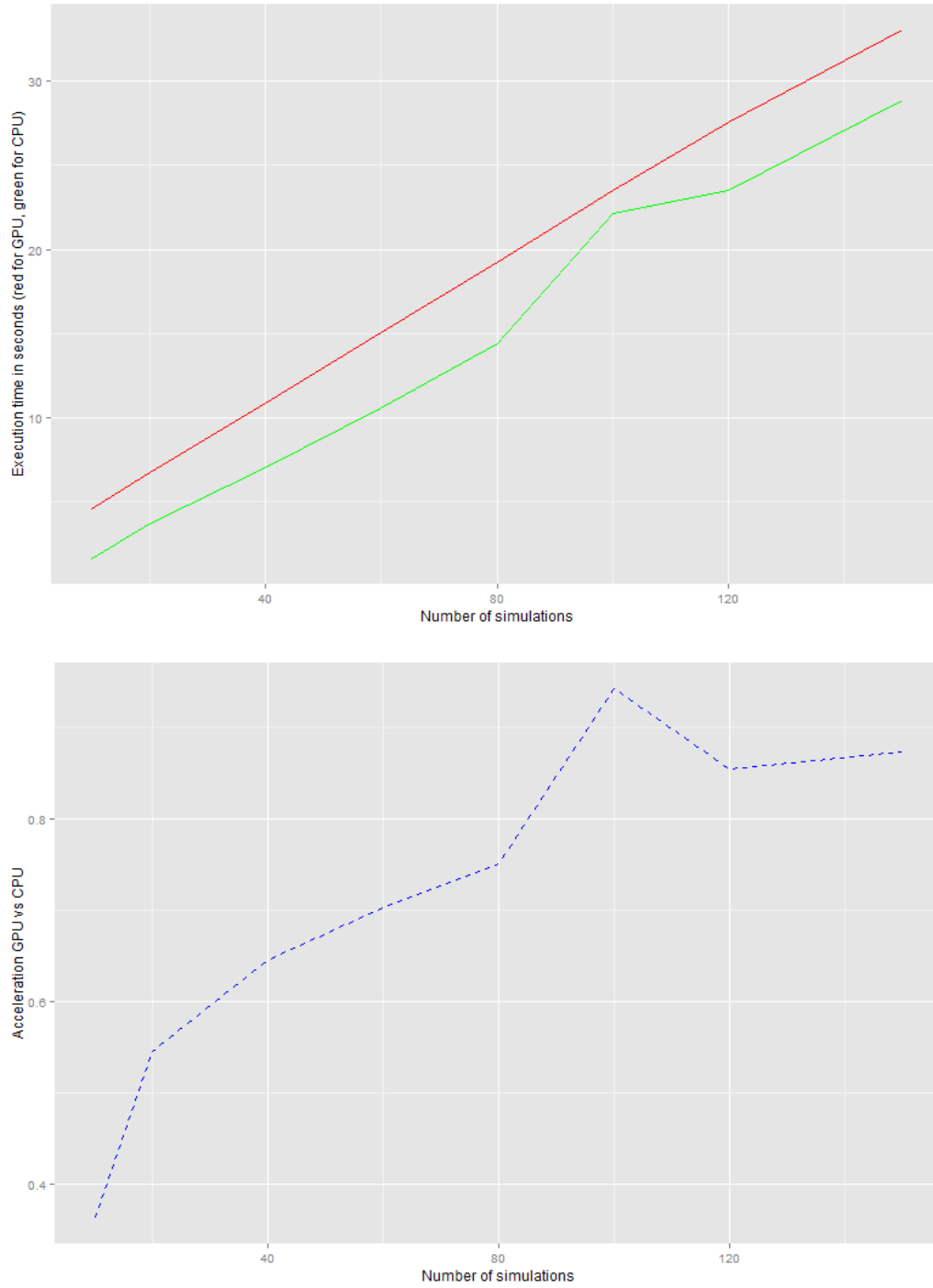


Figure 6.4: Simulation times (left) and acceleration factors (right) for PGPCUDA and PGPC++

fore, an adaptation of the current code for Linux systems, such as the High Performance Computing server at the Research Group on Natural Computing, running on Linux with three NVIDIA Tesla C1060 and one NVIDIA GT 550i cards [10], would enable a more exhaustive performance analysis.

Chapter 7

Conclusions

This chapter summarizes the work presented in this document, recapitulating the achievements which compose this thesis. Firstly, we provide a summary of the whole document, chapter by chapter, highlighting the main contributions and withdrawing general conclusions from the main results. Finally, we propose some future research lines based on this work.

7.1 Summary by chapter

Membrane Computing is a novel discipline which studies the properties of P systems, which are theoretical devices inspired by the structure and functioning of the living cell [190]. Shortly after its introduction, Membrane Computing has been used as a modelling framework for biochemical phenomena, with a plethora of literature examples [41, 199, 45, 102]. Recently, the usage of Membrane Computing as a modelling framework has been extended to the field of ecology [51, 52, 48], thus suggesting its applicability as an approach to model other real-life processes out of biochemistry.

Some of these phenomena have a massively parallel structure. Consequently, it would be appropriate for the modelling framework of choice to be parallel as well. What is more, the systems simulating the models should be also parallel, because the computational power required to simulate models with a large parallel structure would make sequential approaches highly inefficient. Because Membrane Computing provides a parallel modelling framework, it is suitable to model real-life phenomena, which sometimes are large enough to exceed the boundaries of what can be sequentially simulated efficiently with the current technology. This gives place to a need for parallel software applications simulating Membrane Computing models for real-life phenomena. Different

computer architectures have been used for the parallel simulation of P systems, but *Graphic Processing Units (GPUs)* have allegedly among the largest shares of simulated frameworks. There exist simulators based on GPU technology both for P systems solving NP-complete problems [38, 37, 31, 29, 112, 61] and for the simulation of ecological models in Population Dynamics with Membrane Computing [138, 139]. Their promising results reveals GPU technology as a more than suitable platform for the simulation of Membrane Computing models at a large scale.

The main objective of this thesis is the development of Membrane Computing models and simulators for real-life phenomena. Specifically, three phenomena are addressed: Gene Regulatory Networks, Ecosystems and (to a lesser extent) Bio-Inspired Robotics. This variety of applications proves the versatility of Membrane Computing as a modelling framework. Sequential and parallel simulators have also been developed to handle models in these frameworks. To do so, the P-Lingua language [71] has been extended, adding new features and, in the case of ecosystems, new models from scratch. The sequential simulators have been developed on C++ and Java, being the latter incorporated into the P-Lingua API. The parallel simulators have been implemented by using CUDA [10], a programming language for Parallel Computing on GPUs which has already been successfully applied to simulate P Systems [38, 37, 29, 31, 61].

7.2 Thesis overview

What follows is an overview of this thesis, summarizing each chapter and highlighting its major achievements, so as to give the reader a general idea about the results obtained from the thesis.

7.2.1 Summary

This thesis is divided in three parts. The first one introduces Natural Computing in general and Membrane Computing in particular, describing some of the models in the field and some simulators currently implemented for them. The second part focuses on the achievements obtained from this thesis, describing different variants of P systems suitable for computational modelling and the phenomenon under study in each case. Finally, the third part discusses some case studies on the models and simulators presented in part II.

Chapter 1 discusses the field of Natural Computing, with an emphasis on Membrane Computing and some of the main variants proposed in the field. The

chapter ends with two complementary approaches for computational modelling in this discipline which aims to capture the inherent randomness inherent to natural phenomena in several ways: stochastic and probabilistic approaches.

Chapter 2 describes some of the simulators for Membrane Computing models already developed. This chapter is divided in four parts. The first part overviews some of the approaches developed prior to the introduction of P-Lingua. The second part describes the P-Lingua framework, and explains how it revolutionized the state of the art concerning the modelling and simulation of P systems, including some other complementary software frameworks in this line. The third part is a general overview of different parallel architectures which have been used to simulate P systems throughout the years. Finally, the fourth part focuses on the results obtained on the simulation of P systems by means of GPU technology.

Chapter 3 introduces a GPU-based simulator for *Enzymatic Numerical P Systems* (ENPSs), a variant of P systems in which continuous numerical variables are interconnected with each other by means of programs and switch on and off the application of these programs. These variables are associated with membranes which, in turn, are arranged in a cell-like structure. ENPSs are simulated by SNUPS [157], a software tool for the simulation of Numerical P systems. This application simulates these P systems encoded on XML files. The proposed GPU simulator (namely ENPSCUDA) is compatible with this format, and achieved an acceleration of about 6.5x for a dummy model with 15000 membranes and about 10x for a model approximating the exponential function (e^x) with 100000 membranes, both in comparison with a C/C++ counterpart (ENPSC++) developed to measure the acceleration obtained by ENPSCUDA.

Chapter 4 presents a model for *Gene Regulatory Networks* (GRN) within the framework of Population Dynamics P (PDP) systems, a modelling framework originally designed to model processes in ecology. This model, namely *Logic Network Dynamic P* (LNDP) systems, simulates the dynamics of a type of GRNs known as *Logic Networks* (LNs) in which several gene states can influence a third one. A formalization of both LNs and LNDP systems is provided in that chapter. The model is able to reproduce the behaviour of the improved *Logic Analysis of Phylogenetic Profiles* method (improved LAPP method) [229, 228], an algorithm for the simulation of the dynamics of GRNs which considers the possibility of several genes simultaneously influencing a third one. The improved LAPP method outputs the most likely state (i.e., the statistical expectancy) of the gene network after a number of discrete time

steps. As the improved LAPP method is honed over time, the proposed model can be adapted as well, thus becoming iteratively more accurate in tandem with the aforementioned algorithm. Finally, an extension of the P–Lingua language for the generalization of PDP systems is described, so that any LNDP system can be parametrically instantiated without modifying the P–Lingua file.

Chapter 5 introduces *Probabilistic Guarded P* (PGP) Systems, a brand new modelling framework in Membrane Computing for ecological phenomena. The model is formalized and two simulation algorithms (one for sequential and another for parallel architectures) are introduced. These algorithms are restricted for models which do not feature object competition, but some ideas for their extension so that they are able to handle models with object competition are provided as well. Like in Chapter 4, an extension of the P–Lingua language is provided to enable PGP systems in P–Lingua, as well as a Graphical User Interface (GUI) to simulate P systems.

Finally, Chapter 6 describes some case studies for the models and simulators introduced in Chapters 4 and 5. In the first case study, a logic network associated with the flowering process of *Arabidopsis thaliana* is presented. The model succeeds in capturing the behaviour of the Improved LAPP method, thus having a predictive power equivalent to such an algorithm. In the second case study, a model of the ecosystem of *Pieris napi oleracea*, a butterfly native to Northeastern U.S.A., is presented. This model is inspired by another one described in [118], which consists on a set of discrete stochastic equations. The parameters for the Membrane Computing model are either found in [118] and [117] or directly provided by experts in the ecosystem. The experimental validation of the model enabled the analysis of scenarios of special interest for the experts, concluding that the dynamics of the system reflects a steady growth in the overall number of butterflies and an especially sharp increase in the population of heterozygous butterflies with both dominant and recessive alleles for adaptation to *Alliaria petiolata*, an invasive species of plant which is overcoming native crucifer *Cardamine diphylla*. Moreover, a performance analysis of the GPU–based simulator for PGP systems described in Chapter 5 showed that the parallel simulator did not manage to outperform its sequential counterpart. Therefore, a further performance analysis is left as future work.

7.2.2 Major achievements

What follows is a listing of the most relevant contributions from this thesis, which summarizes the achievements obtained from this work.

- A GPU-based simulator for Enzymatic Numerical P Systems [73]. A performance analysis of this simulator reveals an acceleration factor of up to 10x for a model with 100000 membranes which approximates the function e^x , in comparison with a C++ counterpart [74]. This simulator is already available in [13], and has given place to the following publications:
 - 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.
 - M. García-Quismondo, A.B. Pavel, M.J. Pérez-Jiménez. Simulating large-scale ENPS models by means of GPU. In M.A. Martínez del Amor, Gh. Paun, I. Pérez Hurtado, F.J. Romero (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, Volume I, Seville, Spain, January 30–February 3, 2012, Report RGNC 01/2012, Fénix Editora, 2012, pp. 137–152.
- A Membrane Computing model for Gene Regulatory Networks which considers the possibility of several genes influencing a third one [220, 75]. This family of P systems reproduces the behaviour of the improved Logic Analysis of Phylogenetic Profiles method, and has been validated via a case study on the dynamics of a Gene Regulatory Network associated with the flowering process of *Arabidopsis thaliana* [219]. This model is already available in [12], and has given place to the following publications:
 - L. Valencia-Cabrera, M. García-Quismondo, Y. Su, M.J. Pérez-Jiménez, L. Pan, H. Yu. Modeling logic gene networks by means of probabilistic dynamic P systems. *International Journal of Unconventional Computing*, 9, 5–6 (2013), pp. 445–464.
 - L. Valencia-Cabrera, M. García-Quismondo, M.J. Pérez-Jiménez, Y. Su, H. Yu, L. Pan. Analising gene networks with PDP systems. *Arabidopsis thaliana*, a case study. In L. Valencia-Cabrera, M. García-Quismondo, L.F. Macías-Ramos, M.A. Martínez del Amor, Gh. Păun, A. Riscos-Núñez (eds.) *Proceedings of the Eleventh Brainstorming Week on Membrane Computing*, Seville, Spain, Febru-

ary 4–8, 2013, Report RGNC 01/2013, Fénix Editora, 2013, pp. 257–272.

- A Membrane Computing framework for ecological phenomena. This model, known as Probabilistic Guarded P Systems, is inspired by Population Dynamics P systems and aims to simplify the design and simulation of models, as well as a sequential and a parallel (GPU-based) simulator for these models. In addition, a model inside this framework on the ecosystem of *Pieris napi oleracea*, a butterfly native to north-eastern U.S.A., is provided. The model has been experimentally validated by contrasting its results with those obtained by experts in the field. The parallel simulator did not manage to outperform a sequential, C++ counterpart, so further performance improvements are left as future work. Both the model and the simulator are yet to be published, but the latter will be readily available in the P-Lingua [12] and PMCGPU [13] websites.

7.3 Future work

This section proposes some future work so as to continue the achievements obtained from this thesis. Due to the eclectic nature of this work, these research lines are listed regarding the addressed topic.

Enzymatic Numerical P Systems (ENPSs): The simulator discussed in Chapter 3 would be suitable for large scale models which can be applied within the field of robotics. The massively parallel environment provided by the GPUs is suitable for Enzymatic Numerical P Systems simulations. However, it would be interesting to explore the possibility of scaling-up the currently existing robot behaviours modelled with ENPSs and simulate them by means of GPU clusters. These systems might be applied to model the behaviour of massive robot swarms in which robots need to coordinate one another [94], for instance to come up with a planning depending on the environment and revise their plan when necessary [239] and complex sensor networks. Massive, coordinated robot swarms are getting closer to reality by the hour [215], and these models might help predict their behaviour under unexpected circumstances.

ENPSs can be used to model different behaviours, such as *follow the leader*, *obstacle avoidance* and *wall following* [167]. Therefore, the parallel simulator might be used to reproduce several concurrent robot be-

haviours. That is, simulating situations in which robots need to achieve more than one objective at the same time, which is the case of multi-objective robotics, i.e., robots which need to accomplish different (and possibly antagonistic) objectives at the same time [239, 217, 88]. The resulting code can be integrated on FPGA cards which can be embedded on real robots, enabling real-time multi-objective behaviour. In this concern, OpenCL [11] enables compilation not only to GPU cards, but also to other parallel devices such as FPGAs and ARM mobile processors [2]. Moreover, it is important to remark that, although ENPSs are deterministic models, they allow the definition of *input variables* whose values are set by the environment, rather than by programs. This environment usually displays a non-deterministic behaviour [213, 17], thus introducing a random element in the system and, consequently, providing an application for the simulator to repeatedly simulate the same model.

Logic Networks: Chapter 4 describes a Membrane Computing framework for the modelling and simulation of Gene Regulatory Network dynamics. As an additional complementary work, the model can be applied to other large logic networks apart from *Arabidopsis thaliana*. In this sense, a case study on gene regulatory networks from bacterium *Escherichia coli* would be interesting because it is a species extensively used in synthetic biology [85, 83, 143]. Another further enhancement for the framework would be the application of more well-grounded simulation methods than the improved Logic Analysis of Genetic Profiles algorithm, such as the Gillespie algorithm [80] or some extension.

Another proposed line of work consists on a further enhancement by applying random mutations to the genes comprising the network. That is to say, we take into account dynamics in which gene states are not deterministically dictated by network interactions, but also subjected to random modifications. This upgrade could shed light into non-deterministic cell differentiation processes, so as to compare these new dynamics with the ones displayed by the deterministic model proposed here. Finally, the explicit incorporation of proteins regulating gene interaction such as transcription factors would be a step forward towards realistic simulations of gene network dynamics in Membrane Computing.

Probabilistic Guarded P Systems: In Chapter 5, a framework for the modelling and simulation of ecosystems in Membrane Computing is proposed, as well as a sequential and a parallel simulation algorithm for models in this framework. The model succeeds in reflecting the behaviour

of the ecosystem studied in a case study concerning the behaviour of species *Pieris napi oleracea*. In this sense, an extension of the model explicitly integrating parasitoid *Cotesia glomerata* and overparasitoid (i.e. parasitoid parasiting parasitoids) *Cotesia rubecula* would continue yielding information about trends in the ecosystem [118, 117, 43]. Moreover, more case studies in the framework of Probabilistic Guarded P systems constitute a conspicuous line of work.

The software environment can be also subject of improvement. For instance, the software tool MeCoGUI can be integrated into MeCoSim [172]. This would entail a small restructuration on the manner in which the parameters are parsed and the results are displayed, because the parameters would be given by an external CSV file rather than directly input in the application. Likewise, MeCoSim would need not only to display the results of simulations in a graphical environment, but to output simulation files to be analysed by external tools.

Concerning the simulation of PGP systems, an extension of the currently implemented simulators capable of handling models with object competition would be a way forward. Although some ideas are given in that chapter about how a simulator would distribute objects among competing blocks, it is left to be implemented both in sequential and in parallel architectures. Moreover, the performance displayed by the GPU simulator requires an exhaustive profiling analysis, possibly by using NVIDIA Nsight Profiler [10], so as to identify bottlenecks and other hindrances and garner useful information towards a performance improvement of the simulator. Although the computational power of the C++ sequential simulator developed was enough for the model presented in the case study, the parallel simulator would be a useful tool for the cases in which the model structure is too large to be efficiently simulated in sequential architectures.

Some performance improvements can be implemented on PGPCUDA, such as an optimization of memory accesses, possibly by reducing the number of arrays used and taking advantage of faster memories such as shared memory. Furthermore, an adaptation of the current code for Linux systems, such as the High Performance Computing server at the Research Group on Natural Computing [4], running on Linux with three NVIDIA Tesla C1060 and one NVIDIA GT 550i cards [10], would enable a more exhaustive performance analysis.

After such an analysis, the resulting information would allow to identify

and overcome bottlenecks due to inefficient memory accesses. Moreover, it would be possible to improve the way in which random numbers are generated for binomial distribution, possibly using large steps to increase the times that a rule is selected on each rule selection iteration, and to develop more efficient manners to emulate multinomial distribution values. In addition, this analysis would pave the way for model-oriented optimizations, conjointly studying the models and the simulators to take advantage of specific features of the existing models. Finally, one of these lines of future work entail an adaptation of the simulator for more advanced architectures such as the novel Kepler cards [10] and GPU clusters, so as to accelerate simulations for massive models yet to come.

Appendices

Appendix A

Gene Network Data

Gene number	Initial state
1	0
2	0
3	1
4	0
5	0
6	1
7	0
8	1
9	1
10	1
11	0
12	0
13	0
14	1
15	1

Gene number	Initial state
16	0
17	1
18	1
19	1
20	0
21	0
22	1
23	1
24	0
25	1
26	0
27	1
28	1
29	1

Figure A.1: Initial gene states in the *Arabidosis thaliana* gene network on the longday scenario taken as case study

ID	Logic	Weight
1	$g_1 \rightarrow g_7$	0.402
2	$g_2 \rightarrow \neg g_6$	0.409
3	$g_2 \rightarrow g_7$	0.878
4	$g_6 \rightarrow g_{16}$	0.353
5	$g_6 \rightarrow g_{21}$	0.353
6	$g_7 \rightarrow g_{11}$	0.965
7	$g_7 \rightarrow g_{16}$	0.802
8	$g_7 \rightarrow g_{21}$	0.802
9	$g_{10} \rightarrow \neg g_{13}$	0.1000
10	$g_{10} \rightarrow g_{18}$	0.456
11	$g_{10} \rightarrow g_{27}$	0.544
12	$g_{10} \rightarrow g_{28}$	0.309

ID	Logic	Weight
13	$g_{11} \rightarrow \neg g_{26}$	0.273
14	$g_{12} \rightarrow g_{16}$	0.282
15	$g_{12} \rightarrow g_{21}$	0.282
16	$g_{16} \rightarrow \neg g_{29}$	0.713
17	$g_{17} \rightarrow g_{24}$	0.425
18	$g_{17} \rightarrow g_{26}$	0.389
19	$g_{19} \rightarrow g_{29}$	0.551
20	$g_{20} \rightarrow \neg g_{22}$	0.303
21	$g_{21} \rightarrow \neg g_{29}$	0.713
22	$g_{22} \rightarrow g_{26}$	0.439
23	$g_{28} \rightarrow g_{29}$	0.292

Figure A.2: Unary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study

ID	Logic	Weight
1	$g_{11} \wedge g_{27} \rightarrow g_7$	0.708
2	$g_{11} \wedge g_{28} \rightarrow g_7$	1
3	$g_{11} \wedge g_{29} \rightarrow g_7$	0.814
4	$g_{16} \wedge g_{27} \rightarrow g_7$	0.708
5	$g_{16} \wedge g_{28} \rightarrow g_7$	1
6	$g_{16} \wedge g_{29} \rightarrow g_7$	0.814
7	$g_{21} \wedge g_{27} \rightarrow g_7$	0.708
8	$g_{21} \wedge g_{28} \rightarrow g_7$	1
9	$g_{21} \wedge g_{29} \rightarrow g_7$	0.814
10	$g_1 \vee \neg g_{13} \rightarrow g_{10}$	1
11	$g_6 \wedge g_{13} \rightarrow \neg g_{10}$	1
12	$g_7 \vee \neg g_{13} \rightarrow g_{10}$	1
13	$g_9 \wedge g_{13} \rightarrow \neg g_{10}$	0.829
14	$g_{11} \vee \neg g_{13} \rightarrow g_{10}$	1
15	$g_{12} \vee \neg g_{13} \rightarrow g_{10}$	0.829
16	$\neg g_{13} \vee g_{16} \rightarrow g_{10}$	1
17	$\neg g_{13} \vee g_{18} \rightarrow g_{10}$	0.728
18	$g_{13} \wedge g_{19} \rightarrow \neg g_{10}$	0.829
19	$\neg g_{13} \vee g_{21} \rightarrow g_{10}$	1
20	$\neg g_{13} \vee g_{27} \rightarrow g_{10}$	1
21	$g_{27} \vee \neg g_{28} \rightarrow g_{10}$	0.728
22	$g_{10} \wedge g_{16} \rightarrow g_{11}$	0.741
23	$g_{10} \wedge g_{21} \rightarrow g_{11}$	0.741
24	$g_{14} \wedge g_{16} \rightarrow g_{11}$	0.741
25	$g_{14} \wedge g_{21} \rightarrow g_{11}$	0.741
26	$g_{15} \wedge g_{16} \rightarrow g_{11}$	0.741
27	$g_{15} \wedge g_{21} \rightarrow g_{11}$	0.741
28	$g_{16} \wedge g_{17} \rightarrow g_{11}$	0.741
29	$g_{16} \wedge \neg g_{20} \rightarrow g_{11}$	0.741
30	$g_{16} \wedge g_{21} \rightarrow g_{11}$	0.741

ID	Logic	Weight
31	$g_{16} \wedge g_{22} \rightarrow g_{11}$	0.741
32	$g_{16} \wedge g_{23} \rightarrow g_{11}$	0.741
33	$g_{16} \wedge \neg g_{24} \rightarrow g_{11}$	0.741
34	$g_{16} \wedge g_{25} \rightarrow g_{11}$	0.741
35	$g_{16} \wedge \neg g_{26} \rightarrow g_{11}$	0.741
36	$g_{16} \vee g_{29} \rightarrow g_{11}$	0.741
37	$g_{17} \wedge g_{21} \rightarrow g_{11}$	0.741
38	$\neg g_{20} \wedge g_{21} \rightarrow g_{11}$	0.741
39	$g_{21} \wedge g_{22} \rightarrow g_{11}$	0.741
40	$g_{21} \wedge g_{23} \rightarrow g_{11}$	0.741
41	$g_{21} \wedge \neg g_{24} \rightarrow g_{11}$	0.741
42	$g_{21} \wedge g_{25} \rightarrow g_{11}$	0.741
43	$g_{21} \wedge \neg g_{26} \rightarrow g_{11}$	0.741
44	$g_{21} \vee \neg g_{29} \rightarrow g_{11}$	0.741
45	$g_8 \wedge g_{21} \rightarrow g_{16}$	0.801
46	$g_{10} \wedge g_{21} \rightarrow g_{16}$	1
47	$g_{11} \vee g_{21} \rightarrow g_{16}$	1
48	$g_{11} \vee \neg g_{29} \rightarrow g_{16}$	1
49	$g_{14} \wedge \neg g_{19} \rightarrow g_{16}$	0.801
50	$g_{14} \wedge g_{21} \rightarrow g_{16}$	1
51	$g_{15} \wedge g_{21} \rightarrow g_{16}$	1
52	$g_{17} \wedge g_{21} \rightarrow g_{16}$	1
53	$\neg g_{19} \wedge g_{21} \rightarrow g_{16}$	0.801
54	$\neg g_{20} \wedge g_{21} \rightarrow g_{16}$	1
55	$g_{21} \wedge g_{22} \rightarrow g_{16}$	1
56	$g_{21} \wedge g_{23} \rightarrow g_{16}$	1
57	$g_{21} \wedge \neg g_{24} \rightarrow g_{16}$	1
58	$g_{21} \wedge g_{25} \rightarrow g_{16}$	1
59	$g_{21} \wedge \neg g_{26} \rightarrow g_{16}$	1
60	$g_{21} \vee \neg g_{29} \rightarrow g_{16}$	1

Figure A.3: Binary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study (1/2)

ID	Logic	Weight
61	$g_8 \wedge g_{16} \rightarrow g_{21}$	0.801
62	$g_{10} \wedge g_{16} \rightarrow g_{21}$	1
63	$g_{11} \vee g_{16} \rightarrow g_{21}$	1
64	$g_{11} \vee \neg g_{29} \rightarrow g_{21}$	1
65	$g_{14} \wedge g_{16} \rightarrow g_{21}$	1
66	$g_{14} \wedge \neg g_{19} \rightarrow g_{21}$	0.801
67	$g_{15} \wedge g_{16} \rightarrow g_{21}$	1
68	$g_{16} \wedge g_{17} \rightarrow g_{21}$	1

69	$g_{16} \wedge \neg g_{19} \rightarrow g_{21}$	0.801
70	$g_{16} \wedge \neg g_{20} \rightarrow g_{21}$	1
71	$g_{16} \wedge g_{22} \rightarrow g_{21}$	1
72	$g_{16} \wedge g_{23} \rightarrow g_{21}$	1
73	$g_{16} \wedge \neg g_{24} \rightarrow g_{21}$	1
74	$g_{16} \wedge g_{25} \rightarrow g_{21}$	1
75	$g_{16} \wedge \neg g_{26} \rightarrow g_{21}$	1
76	$g_{16} \vee \neg g_{29} \rightarrow g_{21}$	1

Figure A.4: Binary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study (2/2)

Gene number	Initial state
1	0
2	0
3	1
4	0
5	0
6	1
7	0
8	1
9	1
10	1
11	0
12	0
13	0
14	1
15	1

16	0
17	1
18	1
19	1
20	0
21	0
22	1
23	1
24	0
25	1
26	1
27	1
28	1
29	1

Figure A.5: Final gene states in the *Arabidosis thaliana* gene network on the longday scenario taken as case study

Appendix B

PGP Model Data

Parameter	Value
ng	3
np	2
nge	3
nin	4
$npin$	2
ny	10
ns	4
nm	2
nls	10
nc	3

g	N_g
1	100
2	150
3	2970

$Prop_{i,k}$		
$i \backslash k$	1	2
1	0.5	0.5
2	0.5	0.5
3	0.5	0.5

Figure B.1: Integer simulation parameters (left) and values for N_g and $Prop_{i,k}$ for the simulated ecosystem

y	F_y
1	5/10
2	8/10
3	7/10
4	8/10
5	8/10
6	8.5/10
7	8.5/10
8	8.5/10
9	8.75/10
10	8.85/10

i	p_i
1	0.3
2	0.3
3	0.3

i	Ef_i
1	114
2	114
3	114

i	H_i
1	0.73
2	0.53
3	0.73

$Hat_{k,g}$			
$k \backslash g$	1	2	3
1	0.9	0.6	0.2
2	0.8	0.6	0.2

$Det_{k,g}$			
$k \backslash g$	1	2	3
1	1	1	1
2	1	1	1.5

Figure B.2: Values for F_y and $Hat_{k,g}$ (left) and p_i , Ef_i , H_i and $Det_{k,g}$ (right) for the simulated ecosystem

$y \backslash in$	$P_{y,in,1}$			$P_{y,in,2}$			$P_{y,in,3}$		
	1	2	3	1	2	3	1	2	3
1	0.539	0.539	0.539	0.340	0.340	0.340	0.340	0.340	0.340
2	0.534	0.534	0.534	0.335	0.335	0.335	0.335	0.335	0.335
3	0.528	0.528	0.528	0.329	0.329	0.329	0.329	0.329	0.329
4	0.521	0.521	0.521	0.322	0.322	0.322	0.322	0.322	0.322
5	0.514	0.514	0.514	0.315	0.315	0.315	0.315	0.315	0.315
6	0.506	0.506	0.506	0.307	0.307	0.307	0.307	0.307	0.307
7	0.497	0.497	0.497	0.298	0.298	0.298	0.298	0.298	0.298
8	0.487	0.487	0.487	0.288	0.288	0.288	0.288	0.288	0.288
9	0.475	0.475	0.475	0.276	0.276	0.276	0.276	0.276	0.276
10	0.462	0.462	0.462	0.262	0.262	0.262	0.262	0.262	0.262

Figure B.3: Values for $P_{y,in,i}$ for the simulated ecosystem

Parameter	Value
R	0.01
D	0.8
ω	0.3
Se	0.3
U	0.83
Sw	0.169

y	O_y
1	0.56
2	0.56
3	0.56

i	M_i
1	0.41
2	0.41
3	0.41

Figure B.4: Values for non-parametrized probabilities (left) and for O_y and M_y (right) for the simulated ecosystem

Bibliography

- [1] AMD home page. <http://www.amd.com/unleash/>. Official website.
- [2] ARM processor architecture. <http://www.arm.com/products/processors/instruction-set-architectures/index.php>. Official website.
- [3] Colt library. <http://acs.lbl.gov/software/colt/index.html>. Official website.
- [4] GPU server at the research group on natural computing (RGNC). <http://www.gcn.us.es/gpucomputing>. HPC GPU server at the RGNC.
- [5] Infobiotics web page. <http://www.infobiotics.org/>.
- [6] Java web page. <http://www.java.com>. Official website.
- [7] MeCoSim web page. <http://www.p-lingua.org/mecosim>.
- [8] MetaPlab web page. <http://mplab.sci.univr.it/>.
- [9] National Center for Biotechnology Information. <http://www.ncbi.nlm.nih.gov/>.
- [10] NVIDIA CUDA home page. <http://www.nvidia.es/cuda>.
- [11] OpenCL standard webpage. <http://www.khronos.org/opencvl>. Official website.
- [12] P-Lingua webpage. <https://www.p-lingua.org>.
- [13] PMCGPU web page. <http://sourceforge.net/projects/pmcgpu/>. A website including some software tools for the simulation of P systems on GPU architectures.

- [14] R project. <http://www.r-project.org/>. Official website.
- [15] Rand function in C++/C Standard General Utilities Library (cstdlib). <http://www.cplusplus.com/reference/cstdlib/rand/>.
- [16] The OpenMP API specification for parallel programming. <http://www.openmp.org>. Official website.
- [17] M. R. Abdessemed and A. Bilami. Evolutionary research of optimal strategies for exclusive positioned clustering in simulated environment of collective robotics. *Robotics and Autonomous Systems*, 58(10):1130 – 1137, 2010.
- [18] M. Abe, Y. Kobayashi, S. Yamamoto, Y. Daimon, A. Yamaguchi, Y. Ikeda, H. Ichinoki, M. Notaguchi, K. Goto, and T. Araki. Fd, a bzip protein mediating signals from the floral pathway integrator ft at the shoot apex. *Science*, 309(5737):1052–1056, 2005.
- [19] B. Bartlett, C. Clausen, and U. S. A. R. Service. *Introduced parasites and predators of arthropod pests and weeds: a world review*. Agriculture handbook. Agricultural Research Service, U.S. Dept. of Agriculture : for sale by the Supt. of Docs., U.S. Govt. Print. Off., 1978.
- [20] J. Benson, R. V. Driesche, A. Pasquale, and J. Elkinton. Introduced braconid parasitoids and range reduction of a native butterfly in new england. *Biological Control*, 28(2):197 – 213, 2003.
- [21] J. Benson, A. Pasquale, R. V. Driesche, and J. Elkinton. Assessment of risk posed by introduced braconid wasps to pieris virginiensis, a native woodland butterfly in new england. *Biological Control*, 26(1):83 – 93, 2003.
- [22] D. Besozzi, P. Cazzaniga, D. Pescini, and G. Mauri. Seasonal variance in P system models for metapopulations. *Progress in Natural Science*, 17(4):392–400, 2007.
- [23] J. Blakes, J. Twycross, F. J. Romero-Campero, and N. Krasnogor. The infobiotics workbench: an integrated in silico modelling platform for systems and synthetic biology. *Bioinformatics*, 2011.
- [24] E. Bolthausen and M. V. Wathrich. Bernoulli’s law of large numbers. *ASTIN Bulletin*, 43:73–79, 2013.

- [25] P. M. Bowers, S. J. Cokus, T. O. Yeates, and D. Eisenberg. Use of logic relationships to decipher protein network organization. *Science*, 5705(306):2246–2249, 2004.
- [26] P. M. Bowers, B. D. O’Connor, S. J. Cokus, E. Sprinzak, T. O. Yeates, and D. Eisenberg. Utilizing logical relationships in genomic data to decipher cellular processes. *the FEBS journal*, 272(1):5110–5118, 2005.
- [27] C. Buiu, O. Arsene, C. Cipu, and M. Patrascu. A software tool for modeling and simulation of numerical P systems. *Biosystems*, 103(3):442 – 447, 2011.
- [28] C. Buiu, C. Vasile, and O. Arsene. Development of membrane controllers for mobile robots. *Information Sciences*, 187(0):33 – 51, 2012.
- [29] F. 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.
- [30] F. Cabarle, H. 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.
- [31] 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.
- [32] H. Cao, F. Romero-Campero, S. Heeb, M. CAmara, and N. Krasnogor. Evolving cell models for systems and synthetic biology. *Systems and Synthetic Biology*, 4(1):55–84, 2010.
- [33] M. Cardona, M. Colomer, M. Pérez-Jiménez, D. Sanuy, and A. Margalida. Modeling ecosystems using P systems: The bearded vulture, a case study. In D. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 137–156. Springer Berlin Heidelberg, 2009.
- [34] T. Carletti and A. Filisetti. The stochastic evolution of a protocell: The gillespie algorithm in a dynamically varying volume. *Computational and Mathematical Methods in Medicine*, pages 1–13, 2012.

- [35] A. Castellini and V. Manca. Metaplab: A computational framework for metabolic P systems. In D. W. Corne, P. Frisco, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 157–168. Springer Berlin Heidelberg, 2009.
- [36] 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.
- [37] 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.
- [38] 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.
- [39] V. M. Cervantes-Salido, O. Jaime, C. A. Brizuela, and I. M. Martínez-Pérez. Improving the design of sequences for {DNA} computing: A multiobjective evolutionary approach. *Applied Soft Computing*, (0):–, 2013.
- [40] R. Ceterchi, M. Mutyam, G. Păun, and K. G. Subramanian. Array-rewriting p systems. *Natural Computing*, 2(3):229–249, 2003.
- [41] 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*, 17:424–431, 2007.
- [42] F. S. Chew. Coexistence and local extinction in two pierid butterflies. *The American Naturalist*, 118(5):655–672, 1981.
- [43] F. S. Chew and S. P. Courtney. Plant apparency and evolutionary escape from insect herbivory. *The American Naturalist*, 138(3):pp. 729–750, 1991.
- [44] G. Ciobanu and D. Paraschiv. P system software simulator. *Fundamenta Informaticae*, 49(1):61–66, 2002.

- [45] G. Ciobanu, G. Păun, and G. Stănescu. P transducers. *New Generation Computing*, 24(1):1–28, 2006.
- [46] 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.
- [47] E. A. Codling, M. J. Plank, and S. Benhamou. Random walk models in biology. *Journal of The Royal Society Interface*, 5(25):813–834, 2008.
- [48] M. Colomer, I. Pérez-Hurtado, M. Pérez-Jiménez, and A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic P systems over a standard virtual ecosystem. *Natural Computing*, 11(3):369–379, 2012.
- [49] M. A. Colomer, C. Fondevilla, and L. Valencia-Cabrera. A new P system to model the subalpine and alpine plant communities. In *Ninth Brainstorming Week on Membrane Computing*, pages 91–112, Seville, Spain, 2011. Fenix Editora.
- [50] M. A. Colomer, S. Lavín, I. Marco, A. Margalida, I. Pérez-Hurtado, M. J. Pérez-Jiménez, D. Sanuy, E. Serrano, and L. Valencia-Cabrera. Modeling population growth of pyrenean chamois (*rupicapra p. pyrenaica*) by using P systems. *Lecture Notes in Computer Science*, 6501:144–159, 2011.
- [51] M. A. Colomer, A. Margalida, and M. J. Pérez-Jiménez. Population Dynamics P system (PDP) models: A standardized protocol for describing and applying novel bio-inspired computing tools. *PLoS ONE*, 8(4):1–13, 2013.
- [52] 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(1):33 – 47, 2011.
- [53] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
- [54] A. Cerdón-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):349–363, 2004.

- [55] R. F. Costantino, R. A. Desharnais, J. M. Cushing, and B. Dennis. Chaotic dynamics in an insect population. *Science*, 275(5298):389–391, 1997.
- [56] 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.
- [57] S. P. Courtney and S. Courtney. The 'edge-effect' in butterfly oviposition: causality in *anthocharis cardamines* and related species. *Ecological entomology*, 7(2):131–137, 1982.
- [58] M. A. Covington. Antialiasing on the IBM PS2 VGA by treating color bits as subpixels. *Journal of Microcomputer Applications*, 12(3):253 – 257, 1989.
- [59] C. S. Davis. The computer generation of multinomial random variates. *Computational Statistics and Data Analysis*, 16(2):205–217, 1993.
- [60] L. N. de Castro. Fundamentals of natural computing: an overview. *Physics of Life Reviews*, 4:1–36, 2007.
- [61] 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.
- [62] 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.
- [63] D. Díaz-Pernil, I. Pérez-Hurtado, M. 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*, pages 187–203. Springer Berlin Heidelberg, 2009.
- [64] 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.
- [65] R. Donaldson and D. Gilbert. A model checking approach to the parameter estimation of biochemical pathways. In *Proceedings of the 6th International Conference on Computational Methods in Systems Biology, CMSB '08*, pages 269–287, Berlin, Heidelberg, 2008. Springer-Verlag.
- [66] S. R. Eddy. What is a hidden markov model? *Nature*, 22(10):1315 – 1316, 2004.
- [67] A. El-Kateeb. Hardware switch for DMA transfer to augment CPU efficiency. *Microprocessors and Microsystems*, 7(3):117 – 120, 1983.
- [68] F. Fontana, L. Bianco, and V. Manca. P systems and the modeling of biochemical oscillations. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 199–208. Springer Berlin Heidelberg, 2006.
- [69] P. Frisco. P systems with symport-antiport. *Scholarpedia*, 6(10):11704, 2011.
- [70] A. Funahashi, M. Morohashi, H. Kitano, and N. Tanimura. CellDesigner: a process diagram editor for gene-regulatory and biochemical networks. *Biosilico*, 1(5):159–162, 2003.
- [71] M. García-Quismondo, R. Gutiérrez-Escudero, M. A. Martínez-del Amor, E. F. 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, 2010.
- [72] M. García-Quismondo, M. A. Gutiérrez-Naranjo, and D. Ramírez-Martínez. How does a P system sound? In *Eighth Brainstorming Week on Membrane Computing*, pages 123–132. Fenix Editora, 2010.
- [73] M. García-Quismondo, L. F. Macias-Ramos, and M. J. Pérez-Jiménez. volume 4 of *Topics in Intelligent Engineering and Informatics*, chapter Implementing Enzymatic Numerical P Systems for AI Applications by Means of Graphic Processing Units, pages 137–159. Springer Berlin Heidelberg, 2013.

- [74] M. García-Quismondo, A. B. Pavel, and M. J. Pérez-Jiménez. Simulating large-scale ENPS models by means of GPU. In *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, volume I, pages 137–152. Fénix editora, 2012.
- [75] M. García-Quismondo, L. Valencia-Cabrera, Y. Su, M. J. Pérez-Jiménez, L. Pan, and H. Yu. Modeling logic gene networks by means of Probabilistic Dynamic P systems. In L. Pan, G. Paun, and T. Song, editors, *Asian Conference on Membrane Computing*, pages 30–60, Wuhan, China, 2012.
- [76] M. Gardener. *Beginning R: The Statistical Programming Language*. Wrox, 2012.
- [77] 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.
- [78] 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*, pages 97–124, 2013.
- [79] 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.
- [80] D. T. Gillespie. Stochastic simulation of chemical kinetics. *Annual Review of Physical Chemistry*, 58(1):35–55, 2007.
- [81] W. R. Gills. *Markov Chain Monte Carlo in Practice (Chapman & Hall/CRC Interdisciplinary Statistics)*. Chapman and Hall/CRC, 1 edition, 1995.
- [82] O. Gimenez, S. Bonner, R. King, R. Parker, S. Brooks, L. Jamieson, V. Grosbois, B. Morgan, and L. Thomas. Winbugs for population ecologists: Bayesian modeling using markov chain monte carlo methods. In D. L. Thomson, E. Cooch, and M. Conroy, editors, *Modeling Demographic Processes In Marked Populations*, volume 3 of *Environmental and Ecological Statistics*, pages 883–915. Springer US, 2009.

- [83] J. R. Gittins, D. A. Phoenix, and J. M. Pratt. Multiple mechanisms of membrane anchoring of escherichia coli penicillin-binding proteins. *FEMS Microbiology Reviews*, 13(1):1 – 12, 1994.
- [84] B. Gough. *GNU Scientific Library Reference Manual*. Network Theory Ltd., 2009.
- [85] H. M. Grewal, W. Gaastra, A.-M. Svennerholm, J. Röli, and H. Sommerfelt. Induction of colonization factor antigen i (cfa/i) and coli surface antigen 4 (cs4) of enterotoxigenic escherichia coli: relevance for vaccine production. *Vaccine*, 11(2):221–226, 1993.
- [86] V. Grimm. Ten years of individual-based modelling in ecology: what have we learned and what could we learn in the future? *Ecological Modelling*, 115(203):129–148, 1999.
- [87] V. Grimm, T. Wyszomirski, D. Aikman, and J. Uchmaski. Individual-based modelling and ecological theory: synthesis of a workshop. *Ecological Modelling*, 115(2-3):275–282, 1999.
- [88] H. Guo, Y. Meng, and Y. Jin. A cellular mechanism for multi-robot construction via evolutionary multi-objective optimization of a gene regulatory network. *Biosystems*, 98(3):193 – 203, 2009. Evolving Gene Regulatory Networks.
- [89] A. Gutiérrez, L. Fernández, F. Arroyo, and S. Alonso. Suitability of using microcontrollers in implementing new P-system communications architectures. *Artificial Life and Robotics*, 13(1):102–106, 2008.
- [90] A. Gutiérrez, L. Fernandez, 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.
- [91] M. Gutiérrez-Naranjo, M. Pérez-Jiménez, and F. Romero-Campero. A linear solution for qsat with membrane creation. In R. Freund, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3850 of *Lecture Notes in Computer Science*, pages 241–252. Springer Berlin Heidelberg, 2006.

- [92] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and A. Riscos-Núñez. Towards a programming language in cellular computing. *Electronic Notes in Theoretical Computer Science*, 123(0):93–110, 2005.
- [93] M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, and F. J. Romero-Campero. A uniform solution to SAT using membrane creation. *Theoretical Computer Science*, 371(1-2):54 – 61, 2007. `¡ce:title¿Computing and the Natural Sciences¡/ce:title¿.`
- [94] R. Haghighi and C. Cheah. Multi-group coordination control for robot swarms. *Automatica*, 48(10):2526–2534, 2012.
- [95] N. G. Hairston, F. E. Smith, and L. B. Slobodkin. Community Structure, Population Control, and Competition. *The American Naturalist*, 94(879):421–425, 1960.
- [96] A. Hald. James Bernoulli’s law of large numbers for the binomial distribution and its generalization. In *A History of Parametric Statistical Inference from Bernoulli to Fisher, 1713–1935*, Sources and Studies in the History of Mathematics and Physical Sciences, pages 11–15. Springer New York, 2007.
- [97] S. Halle and B. Halle. Modelling activity synchronisation in free-ranging microtine rodents. *Ecological Modelling*, 115(2-3):165–176, 1999.
- [98] M. Haribal, Z. Yang, A. B. Attygalle, J. A. A. Renwick, and J. Meinwald. A cyanoallyl glucoside from *alliaria petiolata*, as a feeding deterrent for larvae of *pieris napi oleracea*. *Journal of Natural Products*, 64(4):440–443, 2001.
- [99] M. V. Herlihy. *Interactions between Pieris oleracea and Pieris rapae (lepidoptera: peridae) butterflies, and the biological control agents Cotesia glomerata and Cotesia rubecula (hymenoptera: braconidae)*. PhD thesis, University of Massachusetts, 2013.
- [100] M. F. Hill, J. D. Witman, and H. Caswell. Spatio-temporal variation in markov chain models of subtidal community succession. *Ecology Letters*, 5(5):665–675, 2002.
- [101] T. Hinze, S. Hayat, T. Lenser, N. Matsumaru, and P. Dittrich. Hill kinetics meets P systems: a case study on gene regulatory networks as computing agents in silico and in vivo. In *Proceedings of the 8th*

- international conference on Membrane computing, WMC'07*, pages 320–335, Berlin, Heidelberg, 2007. Springer-Verlag.
- [102] T. Hinze, S. Hayat, T. Lenser, N. Matsumaru, and P. Dittrich. Hill kinetics meets P systems: A case study on gene regulatory networks as computing agents *in silico* and *in vivo*. In *Workshop on Membrane Computing*, pages 320–335, 2007.
- [103] D. House, M. Walker, Z. Wu, J. Wong, and M. Betke. Tracking of cell populations to understand their spatio-temporal behavior in response to physical stimuli. In *Computer Vision and Pattern Recognition Workshops, 2009. CVPR Workshops 2009. IEEE Computer Society Conference on*, pages 186–193, 2009.
- [104] X. Huang, J. Renwick, and F. Chew. Oviposition stimulants and deterrents control acceptance of *alliararia petiolata* by *pieris rapae* and *p. napi oleracea*. *CHEMOECOLOGY*, 5-6(2):79–87, 1994.
- [105] M. Hucka, A. Finney, H. M. Sauro, H. Bolouri, J. C. Doyle, H. Kitano, , the rest of the SBML Forum:, A. P. Arkin, B. J. Bornstein, D. Bray, A. Cornish-Bowden, A. A. Cuellar, S. Dronov, E. D. Gilles, M. Ginkel, V. Gor, I. I. Goryanin, W. J. Hedley, T. C. Hodgman, J.-H. Hofmeyr, P. J. Hunter, N. S. Juty, J. L. Kasberger, A. Kremling, U. Kummer, N. Le Novire, L. M. Loew, D. Lucio, P. Mendes, E. Minch, E. D. Mjolsness, Y. Nakayama, M. R. Nelson, P. F. Nielsen, T. Sakurada, J. C. Schaff, B. E. Shapiro, T. S. Shimizu, H. D. Spence, J. Stelling, K. Takahashi, M. Tomita, J. Wagner, and J. Wang. The systems biology markup language (SBML): a medium for representation and exchange of biochemical network models. *Bioinformatics*, 19(4):524–531, 2003.
- [106] M. D. Hunter and P. W. Price. Playing Chutes and Ladders Heterogeneity and the Relative Roles of Bottom-Up and Top-Down Forces in Natural Communities. *Ecology (Tempe)*, 73(3), 1992.
- [107] M. D. Hunter, G. C. Varley, and G. R. Gradwell. Estimating the relative roles of top-down and bottom-up forces on insect herbivore populations: a classic study revisited. *Proceedings of the National Academy of Sciences*, 94(17):9176–9181, 1997.
- [108] O. H. Ibarra, M. J. Pérez-Jiménez, and T. Yokomori. On spiking neural P systems. *Natural Computing*, 9(2):475–491, 2010.

- [109] T. Imaizumi, T. F. Schultz, F. G. Harmon, L. A. Ho, and S. A. Kay. Fkfl f-box protein mediates cyclic degradation of a repressor of constans in arabidopsis. *Science*, 309(5732):293–297, 2005.
- [110] T. Imaizumi, H. G. Tran, T. E. Swartz, W. R. Briggs, and S. A. Kay. FKF1 is essential for photoperiodic-specific light signaling in Arabidopsis. *Nature*, 426:301–309, 2003.
- [111] M. Ionescu, G. Păun, and T. Yokomori. Spiking neural P systems. *Fundamenta Informaticae*, 71(2,3):279–308, 2006.
- [112] 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.
- [113] 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.
- [114] O. P. Judson. The rise of the individual-based model in ecology. *Trends in Ecology and Evolution*, 9(1):9–14, 1994.
- [115] H. Kaiser. Quantitative description and simulation of stochastic behaviour in dragonflies (*aeschna cyanea*, odonata). *Acta Biotheoretica*, 25(2-3):163–210, 1976.
- [116] R. Kawai. Nonnegative compartment dynamical system modelling with stochastic differential equations. *Applied Mathematical Modelling*, 36(12):6291–6300, 2012.
- [117] M. S. Keeler and F. S. Chew. Escaping an evolutionary trap: preference and performance of a native insect on an exotic invasive host. *Oecologia*, 156(3):559–568, 2008.
- [118] 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.

- [119] D. B. Kirk and W.-m. W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.
- [120] T. Koyama, T. Ono, M. Shimizu, T. Jinbo, R. Mizuno, K. Tomita, N. Mitsukawa, T. Kawazu, T. Kimura, K. Ohmiya, and K. Sakka. Promoter of arabidopsis thaliana phosphate transporter gene drives root-specific expression of transgene in rice. *Journal of Bioscience and Bioengineering*, 99(1):38 – 42, 2005.
- [121] R. Kuo, P. Wu, and C. Wang. An intelligent sales forecasting system through integration of artificial neural networks and fuzzy neural networks with fuzzy weight elimination. *Neural Networks*, 15(7):909 – 925, 2002.
- [122] M. Kwiatkowska, G. Norman, and D. Parker. *Symbolic Systems Biology*, chapter Probabilistic Model Checking for Systems Biology, pages 31–59. Jones and Bartlett, 2010.
- [123] J. A. Langa, A. Rodríguez, and A. Suárez. On the long time behavior of non-autonomous lotka–volterra models with diffusion via the sub-supertrajectory method. *Journal of Differential Equations*, 249(2):414 – 445, 2010.
- [124] A. Leporati, A. E. Porreca, C. Zandron, and G. Mauri. Improving the universality results of enzymatic numerical P systems. In *Eleventh Brainstorming Week on Membrane Computing*, pages 177–200. Fenix Editora, 2013.
- [125] P. H. Leslie, D. Chitty, and H. Chitty. The estimation of population parameters from data obtained by means of the capture-recapture method: Ii. an example of the practical applications of the method. *Biometrika*, 40(1-2):137–169, 1953.
- [126] W. S. Levine. *The Control Handbook*. CRC Press, New York, NY, USA, 1996.
- [127] A. Lomnicki. Individual-based models and the individual-based approach to population ecology. *Ecological Modelling*, 115(2-3):191 – 198, 1999.
- [128] C. Ltd. *Handel-C Language Reference Manual*. 2005.

- [129] T. Lu, D. Volfson, L. Tsimring, and J. Hasty. Cellular growth and division in the Gillespie algorithm. *Systems Biology*, 1(1):121+, 2004.
- [130] R. Luo, L. Ye, C. Tao, and K. Wang. Simulation of E. coli gene regulation including overlapping cell cycles, growth, division, time delays and noise. *PLoS ONE*, 8(4):e62380, 2013.
- [131] M. Malița. Membrane computing in prolog. In G. P. Cristian S. Calude, Michael J. Dinneen, editor, *Pre-proceedings of the Workshop on Multiset Processing*, volume I, pages 159–175. 2000.
- [132] V. Manca, L. Bianco, and F. Fontana. Evolution and oscillation in P systems: Applications to biological phenomena. In G. Mauri, G. Păun, M. Pérez-Jiménez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 3365 of *Lecture Notes in Computer Science*, pages 63–84. Springer Berlin Heidelberg, 2005.
- [133] C. Martín-Vide, G. Paun, J. Pazos, and A. Rodríguez-Patón. Tissue P systems. *Theoretical Computer Science*, 296(2):295–326, 2003.
- [134] C. Martín-Vide, J. Pazos, G. Păun, and R.-P. A. 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.
- [135] V. Martínez, A. Gutiérrez, and L. Mingo. Circuit fpga for active rules selection in a transition P system region. In M. Kippen, N. Kasabov, and G. Coghill, editors, *Advances in Neuro-Information Processing*, volume 5507 of *Lecture Notes in Computer Science*, pages 893–900. Springer Berlin Heidelberg, 2009.
- [136] M. A. Martínez-del Amor. *Accelerating Membrane Systems Simulators using High Performance Computing with GPU*. PhD thesis, Department of Computer Science and Artificial Intelligence. University of Sevilla, 2013.
- [137] 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.
- [138] 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.
- [139] 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.
- [140] M. A. Martínez-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.
- [141] M. A. Martínez-del Amor, I. Pérez-Hurtado, M. J. Pérez-Jiménez, A. Riscos-Núñez, and F. Sancho-Caparrini. A simulation algorithm for multienvironment probabilistic P systems - a formal verification. *International Journal of Foundations of Computer Science*, 22(01):107–118, 2011.
- [142] R. May. *Stability and complexity in model ecosystems*. Number 6 in Monographs in population biology. Princeton Univ. Press, Princeton, NJ, 2. ed. edition, 1974.
- [143] O. Mol and B. Oudega. Molecular and structural aspects of fimbriae biosynthesis and assembly in escherichia coli. *FEMS Microbiology Reviews*, 19(1):25–52, 1996.
- [144] R. Molla-Vayá and R. Vive-Hernando. Fixed-point digital differential analyser with antialiasing (FDDAA). *Computers and Graphics*, 26(2):329 – 339, 2002.
- [145] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley, 1st edition, 2011.
- [146] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.
- [147] W. W. Murdoch. Community Structure, Population Control, and Competition. *The American Naturalist*, 100(912):219–226, 1966.

- [148] I. A. Nepomuceno-Chamorro. A java simulator for membrane computing. *Journal of Unconventional Computation*, 10(5):620–629, 2004.
- [149] 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.
- [150] V. Nguyen, D. Kearney, and G. Gioiosa. An implementation of Membrane Computing using reconfigurable hardware. *Computing and informatics*, 27(3):551–569, 2008.
- [151] 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.
- [152] 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.
- [153] 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.
- [154] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.
- [155] P. Nordin, W. Banzhaf, F. Informatik, F. Informatik, L. F. Systemanalyse, and L. F. Systemanalyse. Evolving turing-complete programs for a register machine with self-modifying code. In *Genetic algorithms: proceedings of the sixth international conference (ICGA95)*, pages 318–325. Morgan Kaufmann, 1995.
- [156] V. Nuzzo. Invasion pattern of herb garlic mustard (*alliaría petiolata*) in high quality forests. *Biological Invasions*, 1(2-3):169–179, 1999.

- [157] 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.
- [158] S. Omkar, R. Khandelwal, S. Yathindra, G. N. Naik, and S. Gopalakrishnan. Artificial immune system for multi-objective design optimization of composite structures. *Engineering Applications of Artificial Intelligence*, 21(8):1416 – 1429, 2008.
- [159] P. Opler and G. Krizek. *Butterflies east of the Great Plains: an illustrated natural history*. Johns Hopkins University Press, 1984.
- [160] 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.
- [161] P. S. Pacheco. *Parallel programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.
- [162] L. Pan and M. J. Pérez-Jiménez. Computational complexity of tissue-like p systems. *Journal of Complexity*, 26(3):296 – 315, 2010.
- [163] 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.
- [164] G. Paun, M. J. Pérez-Jiménez, and A. R.-N. nez. Tissue P systems with cell division. *International Journal of Computers Communications and Control*, 3:295–303, 2008.
- [165] 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.
- [166] A. B. Pavel and C. Buiu. Using enzymatic numerical P systems for modeling mobile robot controllers. *Natural Computing*, 11(3):387–393, 2012.
- [167] A. B. Pavel, C. Vasile, and I. Dumitrache. Membrane computing in robotics. In J. Kelemen, J. Romportl, and E. Zackova, editors, *Beyond Artificial Intelligence*, volume 4 of *Topics in Intelligent Engineering and Informatics*, pages 125–135. Springer Berlin Heidelberg, 2013.

- [168] A. B. Pavel, C. I. Vasile, and I. Dumitrache. Robot localization implemented with enzymatic numerical P systems. In T. Prescott, N. Lepora, A. Mura, and P. Verschure, editors, *Biomimetic and Biohybrid Systems*, volume 7375 of *Lecture Notes in Computer Science*, pages 204–215. Springer Berlin Heidelberg, 2012.
- [169] L. S. Penrose. The Elementary Statistics of Majority Voting. *Journal of the Royal Statistical Society*, 109(1):53–57, 1946.
- [170] I. Pérez-Hurtado. *Desarrollo y aplicaciones de un entorno de programación para Computación Celular: P-Lingua*. PhD thesis, Department of Computer Science and Artificial Intelligence. University of Sevilla, 2010.
- [171] I. Pérez-Hurtado, M. Pérez-Jiménez, A. Riscos-Núñez, and F. J. Romero-Campero. Membrane computing (tutorial). In C. Calude, J. Kari, I. Petre, and G. Rozenberg, editors, *Unconventional Computation*, volume 6714 of *Lecture Notes in Computer Science*, pages 38–39. Springer Berlin Heidelberg, 2011.
- [172] 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 K. Li, Z. Tang, R. Li, A. K. Nagar, and R. Thamburaj, editors, *IEEE Fifth International Conference on Bio-inspired Computing: Theories and Applications (BIC-TA 2010)*, volume I, pages 637–643, Changsha, China, 2010. IEEE, Inc.
- [173] M. Pérez-Jiménez and A. Riscos-Núñez. Solving the subset-sum problem by P systems with active membranes. *New Generation Computing*, 23(4):339–356, 2005.
- [174] M. 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*, pages 250–268. Springer Berlin Heidelberg, 2004.
- [175] 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, 2004. Fenix Editora.

- [176] M. J. Pérez-Jiménez and F. J. Romero-Campero. Solving the bin packing problem by recognizer P systems with active membranes. In *Proceedings of the Second Brainstorming Week on Membrane Computing*, 2004.
- [177] M. J. Pérez-Jiménez and F. J. Romero-Campero. P systems, a new computational modelling tool for systems biology. In C. Priami and G. Plotkin, editors, *Transactions on Computational Systems Biology VI*, volume 4220 of *Lecture Notes in Computer Science*, pages 176–197. Springer Berlin Heidelberg, 2006.
- [178] M. J. Pérez-Jiménez, A. Romero-Jiménez, and F. Sancho-Caparrini. A polynomial complexity class in P systems using membrane division. *Journal of Automata, Languages and Combinatorics*, 11:423–434, 2006.
- [179] M. J. Pérez-Jiménez and F. Sancho-Caparrini. A formalization of transition P systems. *Fundam. Inf.*, 49(1):261–272, 2002.
- [180] 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.
- [181] C. A. Petri. *Kommunikation mit Automaten*. PhD thesis, Universität Hamburg, 1962.
- [182] C. A. Petri. Fundamentals of a Theory of Asynchronous Information Flow. In *Proceedings of IFIP Congress*, pages 386–390. North Holland Publishing Company, 1963.
- [183] M. E. Power. Top-Down and Bottom-Up Forces in Food Webs: Do Plants Have Primacy. *Ecology*, 73(3):733+, 1992.
- [184] A. Profir, E. Gutuleac, and E. Boian. Simulation of continuous-time P systems using descriptive rewriting timed Petri nets. In *SYNASC*, pages 458–461, 2005.
- [185] A. Păun and B. Popa. P systems with proteins on membranes. *Fundamenta Informaticae*, 72(4):467–483, 2006.
- [186] A. Păun and B. Popa. P systems with proteins on membranes and membrane division. In *Proceedings of the 10th international conference on Developments in Language Theory, DLT'06*, pages 292–303, Berlin, Heidelberg, 2006. Springer-Verlag.

- [187] A. Păun and G. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, 20(3):295–305, 2002.
- [188] G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61(1):108 – 143, 2000.
- [189] 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.
- [190] G. Păun. *Membrane Computing: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.
- [191] 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.
- [192] G. Păun and R. Păun. Membrane computing as a framework for modeling economic processes. In *Symbolic and Numeric Algorithms for Scientific Computing, 2005. SYNASC 2005. Seventh International Symposium on*, pages 8 pp.–, 2005.
- [193] G. Păun and R. Păun. Membrane Computing and economics: Numerical P systems. *Fundam. Inf.*, 73(1,2):213–227, 2006.
- [194] Z. Qi, J. You, and H. Mao. P systems and petri nets. In C. Martin-vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 2933 of *Lecture Notes in Computer Science*, pages 286–303. Springer Berlin Heidelberg, 2004.
- [195] 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.
- [196] D. Ramírez-Martínez and M. A. Gutiérrez-Naranjo. A software tool for dealing with spiking neural P systems. In M. A. Gutiérrez-Naranjo, editor, *Fifth Brainstorming Week on Membrane Computing*, pages 299–313. Fenix Editora, 2007.
- [197] J. A. Renwick. The chemical world of crucivores: Lures, treats and traps, 2002.

- [198] J. A. Renwick, W. Zhang, M. Haribal, and A. B. Attygalle. Dual chemical barriers protect a plant against different larval stages of an insect. *Journal of Chemical Ecology*, 27(8):1575–1583, 2001.
- [199] F. J. Romero-Campero and M. J. Pérez-Jiménez. A model of the quorum sensing system in vibrio fischeri using P systems. *Artificial Life*, 14(1):95–109, 2008.
- [200] 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. P-Systems Applications to Systems Biology.
- [201] F. J. Romero-Campero, J. Twycross, M. Cámara, M. Bennett, M. Gheorghe, and N. Krasnogor. Modular assembly of cell systems biology using P systems. *International Journal of Foundations of Computer Science*, 20(03):427–442, 2009.
- [202] A. Romero-Jiménez, M. Gutiérrez-Naranjo, and M. Pérez-Jiménez. Graphical modeling of higher plants using P systems. In H. Hoogeboom, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4361 of *Lecture Notes in Computer Science*, pages 496–506. Springer Berlin Heidelberg, 2006.
- [203] A. Romero-Jiménez and M. J. Pérez-Jiménez. Computing partial recursive functions by transition P systems. 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 320–340. Springer Berlin Heidelberg, 2004.
- [204] 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.
- [205] M. Scheffer, J. M. Baveco, D. L. Deangelis, E. H. R. R. Lammens, and B. Shuter. Stunted Growth and Stepwise Die-Off in Animal Cohorts. *American Naturalist*, 145(3):376–388, 1995.
- [206] R. Scott. Alien, the 8th passenger.
- [207] C. Shao, X. Ma, X. Xu, and Y. Meng. Identification of the highly accumulated microrna*s in arabidopsis (arabidopsis thaliana) and rice (oryza sativa). *Gene*, 515(1):123 – 127, 2013.

- [208] D. Shiffman. *The Nature of Code: Simulating Natural Systems with Processing*. Theoklesia, Llc, 1st edition, 2012.
- [209] 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.
- [210] A. Spicher, O. Michel, M. Cieslak, J.-L. Giavitto, and P. Prusinkiewicz. Stochastic P systems and the simulation of biochemical processes with dynamic compartments. *Biosystems*, 91(3):458 – 472, 2008. P-Systems Applications to Systems Biology.
- [211] V. A. Sprau. *Computer Science Made Simple: Learn how hardware and software work– and how to make them work for you!* Broadway, 1st edition, 2005.
- [212] R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *J. of Global Optimization*, 11(4):341–359, 1997.
- [213] L. Su and M. Tan. A virtual centrifugal force based navigation algorithm for explorative robotic tasks in unknown environments. *Robotics and Autonomous Systems*, 51(4):261 – 274, 2005.
- [214] 68030 features Harvard architecture. *Microprocessors and Microsystems*, 10(10):567, 1986.
- [215] Y. Takemura, M. Sato, and K. Ishii. Toward realization of swarm intelligence mobile robots. *International Congress Series*, 1291(0):273 – 276, 2006. Brain-Inspired IT II: Decision and Behavioral Choice Organized by Natural and Artificial Brains. Invited and selected papers of the 2nd International Conference on Brain-inspired Information Technology.
- [216] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. Checkl: Transparent checkpointing and process migration of opencl applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 864–876, 2011.
- [217] A. T. Tolmidis and L. Petrou. Multi-objective optimization for dynamic task allocation in a multi-robot system. *Engineering Applications of Artificial Intelligence*, 26(5–6):1458 – 1468, 2013.

- [218] S. M. Trimberger. *Field-Programmable Gate Array Technology*. Springer-Verlag New York, Inc., Boston, MA, USA, 1994.
- [219] L. Valencia-Cabrera, M. García-Quismondo, M. J. Pérez-Jiménez, Y. Su, H. Yu, and L. Pan. Analysing gene networks with pdp systems. arabidopsis thaliana, a case study. *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*, pages 257–272, 2013.
- [220] L. Valencia-Cabrera, M. García-Quismondo, M. J. Pérez-Jiménez, Y. Su, H. Yu, and L. Pan. Modeling logic gene networks by means of probabilistic dynamic p systems. *International Journal of Unconventional Computing*, 9(5-6):445–464, 2013.
- [221] C. Vasile, A. Pavel, I. Dumitrache, and G. Păun. On the power of enzymatic numerical P systems. *Acta Informatica*, 49(6):395–412, 2012.
- [222] C. I. Vasile, A. B. Pavel, and I. Dumitrache. Improving the universality results of enzymatic numerical P systems. In *Tenth Brainstorming Week on Membrane Computing*, pages 207–214. Fenix Editora, 2012.
- [223] S. Verlan and J. Quiros. Fast hardware implementations of P systems. In *Proceedings of the 13th international conference on Membrane Computing, CMC’12*, pages 404–423. Springer-Verlag, 2013.
- [224] J. Villasenor and B. Hutchings. The flexibility of configurable computing. *Signal Processing Magazine, IEEE*, 15(5):67–84, 1998.
- [225] J. Wang, V. Athitsos, S. Sclaroff, and M. Betke. Detecting objects of variable shape structure with hidden state shape models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(3):477–492, 2008.
- [226] J. Wang, P. Shi, H. Peng, M. Pérez-Jiménez, and T. Wang. Weighted fuzzy spiking neural p systems. *Fuzzy Systems, IEEE Transactions on*, 21(2):209–220, 2013.
- [227] 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.

- [228] S. Wang, Y. Chen, Q. Wang, E. Li, Y. Su, and D. Meng. Analysis for stimuli to shoot genes of *Arabidopsis thaliana* based on logical relationships. *Optimization and Systems Biology*, 11:435–447, 2006.
- [229] S. Wang, Y. Chen, Q. Wang, E. Li, Y. Su, and D. Meng. Analysis for gene networks based on logic relationships. *Journal of Systems Science and Complexity*, 23:999–1011, 2010.
- [230] D. Whitley, T. Starkweather, and C. Bogart. Genetic algorithms and neural networks: optimizing connections and connectivity. *Parallel Computing*, 14(3):347 – 361, 1990.
- [231] C. Wiklund, P.-O. Wickman, and S. Nylin. A sex difference in the propensity to enter direct/diapause development: A result of selection for protandry. *Evolution*, 46(2):519–528, 1982.
- [232] D. J. Wilkinson. Stochastic modelling for quantitative description of heterogeneous biological systems. *Nature Reviews Genetics*, 10(2):122–133, 2009.
- [233] S. Wolfram. *A New Kind of Science*. Turnaround, 1st edition, 2002.
- [234] Z. Wu, M. Betke, J. Wang, V. Athitsos, and S. Sclaroff. Tracking with dynamic hidden-state shape models. In D. Forsyth, P. Torr, and A. Zisserman, editors, *Computer Vision at ECCV 2008*, volume 5302 of *Lecture Notes in Computer Science*, pages 643–656. Springer Berlin Heidelberg, 2008.
- [235] X.-S. Yang and M. Karamanoglu. Swarm intelligence and bio-inspired computation: An overview. In X.-S. Yang, Z. Cui, R. Xiao, A. H. Gandomi, and M. Karamanoglu, editors, *Swarm Intelligence and Bio-inspired Computation*, pages 3 – 23. Elsevier, Oxford, 2013.
- [236] X. Zeng, H. Adorna, M. 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.
- [237] S.-Z. Zhang and Z. Jian-Rhu. Advance in the flowering time control of *Arabidopsis*. *Progress in Biochemistry and Biophysics*, 33:301–309, 2006.

-
- [238] X. Zhang, S. Kim, T. Wang, and C. Baral. Joint learning of logic relationships for studying protein function using phylogenetic profiles and the rosetta stone method. *Trans. Sig. Proc.*, 54(6):2427–2435, 2006.
- [239] Y. Zhang, D. wei Gong, and J. hua Zhang. Robot path planning in uncertain environment using multi-objective particle swarm optimization. *Neurocomputing*, 103(0):172 – 185, 2013.
- [240] J. Zobitz, A. Desai, D. Moore, and M. Chadwick. A primer for data assimilation with ecological models using Markov Chain Monte Carlo (MCMC). *Oecologia*, 167(3):599–611, 2011.