



DEPARTMENT OF COMPUTER SCIENCE AND
ARTIFICIAL INTELLIGENCE
UNIVERSITY OF SEVILLE

Promoting and Inhibiting Contexts
in
Membrane Computing

Dragoş Sburlan

e-Mail: *dsburlan@univ-ovidius.ro*

Supervisors:

Dr. Mario Pérez Jiménez

Dr. Gheorghe Păun

Seville, November 2005

Promoting and Inhibiting Contexts
in
Membrane Computing

Dragoş Sburlan

Doctoral Thesis

Acknowledgments

The present work was possible due to a doctoral fellowship from the Spanish Ministry of Foreign Affairs – The Spanish Agency for International Cooperation (AECI) and to a doctoral grant from Spanish Ministry of Education.

First I would like to thank Mario Pérez Jiménez for his contagious optimism, continuous help in any problem and altruism. I owe him very much. I also gratefully acknowledge all the members of the Research Group in Natural Computing for creating such a friendly and stimulating scientific atmosphere at University of Seville.

I would also like to thank to Gheorghe Păun, my adviser during my doctoral stage in Spain. I am very indebted for his constant support, advice that went beyond the science, and enthusiasm for my research.

My thanks are addressed also to Grzegorz Rozenberg for the many constructive scientific discussions and guidance during my research stage in Leiden, The Netherlands.

A special thank I address to my faculty professors Christian Calude and Dan Luca Șerbanăți from whom I have learned the first steps in the theory of computation.

I would also like to thank my family for their love and encouragements and especially to my father who always stand on me, backing me in any matter. Last, but not least, I would like to thank my wife Cristina and my son Andrei, for their patience, support and understanding, as well as for other reasons beyond the scope of this dissertation.

My most intense collaborations have been with Artiom Alhazov, Ioan Ardelean, Matteo Cavaliere, Agustín Riscos-Núñez, Rodica Ceterchi, Radu Gramatovici, Mihai Ionescu, and Peter Leupold whose clarity, persistence, ability to create new models, and desire to write new publications, have taught me a lot.

*Dragoș Sburlan
Seville, Spain
November 2005*

Citations to previously published work

Large portions of Chapters 3, 4, 5, 6, and 7 have appeared in the following papers:

[3] Alhazov A., Sburlan D., Static Sorting Algorithms for P Systems, *Applications of Membrane Computing* (Ciobanu G., Pérez-Jiménez M., Păun G. Eds.), Springer-Verlag, Berlin, 2005.

[4] Alhazov A., Sburlan D., Ultimately Confluent Rewriting Systems. Parallel Multiset-Rewriting with Permitting or Forbidding Contexts, *Lecture Notes in Computer Science*, 3365 (2005), 178–189.

[6] Ardelean I., Cavaliere M., Sburlan D., Computing Using Signals: From Cells to P Systems, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9, 9 (2005), 631–639.

[21] Cavaliere M., Sburlan D., Time-Independent P Systems, *Lecture Notes in Computer Science*, 3365 (2005), 239–258.

[22] Cavaliere M., Sburlan D., Time and Synchronization in Membrane Systems, *Fundamenta Informaticae*, 64, 1-4 (2005), 65–77.

[23] Cavaliere M., Riscos-Núñez A., Rozenberg G., Sburlan D., String Driven Computation in Membrane Systems, manuscript.

[24] Cavaliere M., Riscos-Núñez A., Rozenberg G., Sburlan D., P Systems with External Control, manuscript.

[26] Ceterchi R., Sburlan D., Simulating Boolean Gates with P Systems, *Lecture Notes in Computer Science*, 2933 (2003), 104–122.

[47] Ionescu M., Sburlan D., On P Systems with Promoters/Inhibitors, *Journal of Universal Computer Science*, 10, 5 (2004), 581–599.

[82] Sburlan D., A Static Sorting Algorithm for P Systems with Mobile Catalysts, *An. St. Univ. "Ovidius" Constantza*, XI, 1 (2003), 195–205.

[83] Sburlan D., New Results on P Systems with Multiset Promoted/Inhibited Rules, *Bull. PAMM*, 2164 (2004), 45–54.

[84] Sburlan D., From Cells to Digital Signal Processing, *Proceedings of International Workshop on Mathematical Modelling of Environmental and Life Science Problems*, Romanian Academy of Sciences, Romania, 2004, 269–278.

- [85] Sburlan D., Clock-free P Systems, *Pre-Proceedings of the Fifth International Workshop on Membrane Computing (WMC5)*, Milan, Italy, 2004, 372–383.
- [86] Sburlan D., Modeling The Dynamical Parallelism of Bio-Systems. *M-Rate Systems, Pre-Proceedings of the Sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 517–529.
- [87] Sburlan D., Non-cooperative P systems with Priorities Characterize PsETOL, *Pre-Proceedings of the Sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 530–539.
- [88] Sburlan D., Further Results On P Systems with Promoters/Inhibitors, accepted *International Journal of Foundations of Computer Science*.

Contents

1	Introduction	1
1.1	A Glimpse to Natural Computing	4
1.2	Molecular Computing Challenges	5
2	Models of Computation Preliminaries	17
2.1	Languages, Grammars and Turing Machines	17
2.2	Regulated Rewriting	24
2.3	Register Machines	29
2.4	Lindenmayer Systems	30
2.5	Deterministic and Confluent Rewriting	31
2.6	Summary	33
3	P Systems – A Simple Computational Model	35
3.1	P Systems with Promoters/Inhibitors	35
3.2	P Systems with Non-Cooperative Promoted/Inhibited Rules	40
3.3	Catalytic P Systems with Promoted/Inhibited Rules	52
3.3.1	Computational Universality – The Generative Case	52
3.3.2	Computational Universality – The Accepting Case	59
3.4	Ultimately Confluent Universality	64
3.5	Open Problems and Forthcoming Research	72
4	P Systems Generalizations	75
4.1	Timed P Systems	75
4.1.1	Time Free P Systems	76
4.1.2	Clock-Free P Systems	81
4.2	Modeling the Dynamical Parallelism of Bio-Systems	90
4.2.1	On the Dynamical Parallelism of L Systems	91
4.2.2	On the Dynamical Parallelism of P Systems	98

4.3	Open Problems and Forthcoming Research	101
5	An Algorithmic Overview	103
5.1	Sorting with P Systems	103
5.1.1	Sorting Preliminaries	104
5.1.2	Static Sorting with P Systems with Promoters/Inhibitors . . .	107
5.1.3	Open Problems and Forthcoming Research	110
5.2	Simulating Boolean Circuits with P Systems	111
5.2.1	Motivations	111
5.2.2	Boolean Circuits Preliminaries	112
5.2.3	Simulating Boolean Gates	113
5.2.4	Simulating Boolean Circuits	117
5.2.5	Open Problems and Forthcoming Research	121
6	Two Natural Extensions	123
6.1	P Systems with External Promoters/Inhibitors	123
6.2	String Driven Computation	133
6.3	Open Problems and Forthcoming Research	142
7	Promoters & Inhibitors in the P System Framework	143
7.1	P Systems with Strong Priorities	143
7.2	Open Problems and Forthcoming Research	150
8	Conclusions	151

Chapter 1

Introduction

The beginning of the modern science that we call “Computer Science” can be traced back to a long age ago. The first roots have been probably in the Asian lands, 5000 BC, where businessmen involved in commerce with goods needed a way to tally accounts and bills. In a sense this is the moment when the abacus was born in Babylonia. It was the first true predecessor to the adding machines and computers which would follow and it works on the principle of place-value notation. In fact, the abacus is really a memory aid for the user making mental calculations, as opposed to the true mechanical calculating machines which were still to come. Also, from those ancient times we have one of the first algorithms: the Sieve of Eratosthenes which was used to determine the prime numbers.

For over a thousand years since the Chinese (re)invented the abacus, not much progress was made to automate counting and mathematics despite the remarkable advances in exact sciences done by the Greeks. All this time most of the tables of integrals, logarithms, and trigonometric values were worked out by hand.

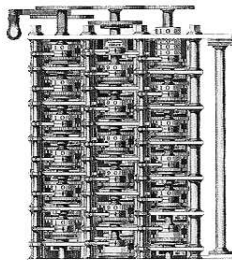


Figure 1.1: Babbage’s machine

Their accuracy required a huge amount of labor and usually the correctness of the answer to a certain problem relied on the size of the redundant personnel working on the same problem. It was in 1642 when Blaise Pascal, the well known mathematician, thinker, and scientist, built the first mechanical adding machine – the *Pascaline*.

Almost two hundred years later, Charles Babbage begins to design and build the Difference Engine being convinced that it is possible to design a calculating machine that can do long computation by regularly repeating some simple operations. He shifts his

focus on the analytical engine – a real parallel decimal computer which would operate on words of 50 decimals and was able to store 1000 such numbers. The machine would include a number of built-in operations such as conditional control, which allowed the instructions for the machine to be executed in a specific order rather than in numerical order. The instructions for the machine were to be stored on punched cards. Even more, Babbage in a visionary endeavor imagined a way to print the output of the computation.

By the year 1936, Alan Turing’s remarkable paper “On Computable Numbers” presents the concept of the Turing machine - it will revolutionize the way computers are designed. Inspired by these ideas in 1943, *Colossus*, a British vacuum tube computer, becomes operational at Bletchley Park through the combined efforts of Alan Turing, Tommy Flowers, and M.H.A. Newman. This will be known later on as the first all-electronic calculating device.

John von Neumann, the brilliant mathematician from Princeton introduces in 1945 the concept of a stored program in a draft report on the *EDVAC* (Electronic Discrete Variable Automatic Computer) design. His 1946 paper, written with Arthur W. Burks and Hermann H. Goldstine, was titled “Preliminary Discussion of the Logical Design of an Electronic Computing Instrument,” and the principles revealed in it were to have a profound effect on the subsequent development of computer science.

Since then, any discussion about computer architectures, about how computers and computer systems are organized, designed, and implemented, inevitably makes reference to the “*von Neumann architecture*” as a basis for comparison.

All the rest that came afterward was a permanent struggle of the mankind for speed and miniaturization of the machines. Processors designs, computer architectures, algorithms, formal models and so on, were all together assembled in a new science apart from engineering or mathematics: *computer science*.

However, after seven thousand years the mankind made a new turn-point in the field of computing – not only that we are looking in the nature as for a source of inspiration in our quest for finding better computing machineries, but even more, we are intending to use the

“nature toolbox” for our computing goals or to design nano-scale programmable

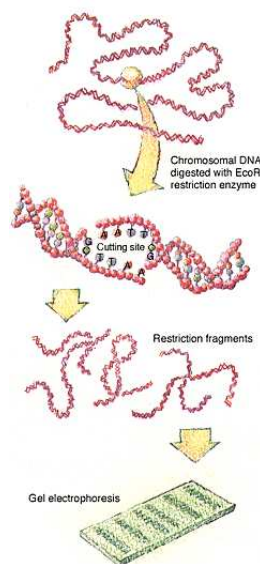


Figure 1.2: Designing a DNA-computer

devices for curing purposes, that can heal diseases, that can lengthen our life time, that can increase our abilities and so on. Besides all of this, we are trying by these means to understand the deep nature of things like how the life emerges out of presumably simple mechanisms or like how and especially why quantum effects occurs, or even to develop a new kind of intelligence.

It was John von Neumann who in 1947 introduced the notion of cellular automata in his attempt to develop an abstract model of self-reproduction in biology - a topic which had emerged at that time from investigations in cybernetics. Cellular automata were designed also to answer the basic question: *is it possible to construct robots that can construct identical robots*, i.e., robots with the same complexity? The model proposed by von Neumann gave a positive answer to such question.

Later on, in 1982, the Nobel prize-winning physicist Richard Feynman came up with the idea of a 'quantum computer'. Constructing such a machine will not be a simple task because of the delicate nature of quantum mechanical systems. However, the benefits will be remarkable; since the laws of quantum mechanics assume unintuitive principles such as superposition and entanglement, a quantum computer would be able to overpass some rules that restrict our classical computers. For example, using the superposition one can use a quantum bit of information (the *qubit*) in several computations at the same time. Using entanglement one can process information over long distances without the classical requirement of wires. Therefore, an incredible computing speed can be achieved.

In the nineties Leonard Adleman demonstrated that DNA can be used as a computing medium. Here the methodology was completely different: the goal was to develop techniques and methods for managing organic and biomolecular reactions such that molecules could be used in information processing. One particular molecule that has proved most promising for molecular computation is Deoxyribonucleic acid (DNA). DNA is a long polymer made of 4 different nucleotides (conventionally named A, T, C and G). The order or sequence of these nucleotides within DNA provides the information for making protein, the main components of the molecular scale machinery used by living organisms to carry out life sustaining functions.

All these examples represent great changes in computer science and information technology and come from mimicking the techniques and methods by which biological organisms process information or, "zooming" even more, from using nano-scale level natural phenomena. They constitute the starting point of one main branch in computer science, the new computing paradigm represented by what we now call the *Natural Computing*.

1.1 A Glimpse to Natural Computing

Natural Computing is a widespread notion referring to computing performed in nature or by nature and computing inspired by nature. In the swift developing field of computer science, natural computing has a significant role as the catalyst for the synergy of human designed computing with the computing going on in nature. This interaction leads to a profound and broader understanding of the nature of computation. Besides all these, valuable insights into natural sciences can be acquired and henceforth progress archived.

From this point of view, characteristic for man-designed computing inspired by nature is the search for abstraction of concepts, principles and mechanisms underlying natural systems. Nowadays, computer science undergoes an important metamorphosis by trying to unify and combine the computing carried on in computer science with the computing observed in nature all around us.

Thus, evolutionary algorithms use the concepts of mutation, recombination and natural selection from biology in order to deal with optimization problems; neural networks employ ideas from the highly interconnected neural structures in the animal nervous systems and are used for instance in pattern recognition or data classification; molecular computing is based on paradigms from molecular biology; membrane computing is inspired by the cell functioning and provides a framework for studying it; finally, quantum computing, based on quantum physics, exploits quantum parallelism and is already used for data encryption and secured transmissions.

Apart from sharing the natural inspiration, various fields in natural computing have also significant methodological differences. Thus, e.g., evolutionary algorithms and algorithms based on neural networks are presently implemented on conventional computers. On the other hand, molecular computing also aims at alternatives for silicon hardware by implementing algorithms in biological hardware (bioware), e.g., using DNA molecules and enzymes. Membrane computing points to even more ambitious goals by trying to model phenomena occurring in cells and to use them as computational devices. Also quantum computing aims at nontraditional hardware that would allow quantum effects to take place and remarkable computing speeds to be reached.

When speaking about biological aspects of the nature one can remark that the evolution has created a large amount of systems in which the actions of simple, locally-interacting components emerge into coordinated global information processing. In this respect, insect colonies, cellular assemblies, the immune system, are just

known examples of systems in which emergent computations occur. One common characteristic is that even if the global information-processing capabilities are not explicitly represented in the system's elementary components or in their interconnections, the whole complex is able to perform computational tasks.

The highly parallel cellular machinery designed by nature exhibits impressive problem solving competence, while operating within a dynamic environment that influences its behavior. This is why the central question one can ask is whether we can mimic nature's achievements, creating human made machines that exhibit features such as those revealed by their natural counterparts. Obviously, this ambitious goal is yet far, however, the general intent is to take small steps toward it.

Nevertheless, the search for inspiration in the nature just for mimic its behavior is not the only goal. To pursue the construction of molecular scale computers is one main trend of miniaturization. This tendency was present in microelectronics for decades and was evoked by the famous physicist Feynman¹, recognized latter by the known computer scientist Moore in what is known as the Moore's law.

This approach will question indubitably the traditional von Neumann architecture based on single logic processing unit, single addressed memory, a control unit and a user interface. To design such machines will obviously imply many difficulties that have to be overpassed; the experience showed that there is a trade-off between the programmability, functionality, efficiency and the complexity of a certain system. From this point of view, biological systems are weaker in programmability and operability but through superior adaptability are able to efficiently solve complex problems.

1.2 Molecular Computing Challenges

Life provides examples of difficult biological information devices that might give models for technological concepts. But which elements of biological systems provide valuable models? Searching into the nature for models is a different method from trying to include biological materials into computers. This last aim may in fact not even work also because biological systems are likely to be too slow for most information processing applications. This is why the current emphasis is how to use biology to discover new computing models, and henceforth to improve the information processing.

¹Feynman (1959): "There is plenty of room at the bottom" – talk presented at American Physical Society meeting, at California Institute of Technology

Molecular computing is one fast growing topic of natural computing; it is concerned with the use of bio-molecules for the purpose of actual computations and it attempts to understand the computational nature of molecular processes occurring in living cells. Nowadays, in the molecular computing field there exist two significant branches: cellular and membrane computing. Since they start from the same source of inspiration – the cell, both of them share many common characteristics. Moreover, they perform three common remarkable features, namely, massive parallelism, locality of cellular interactions, and simplicity of basic components.

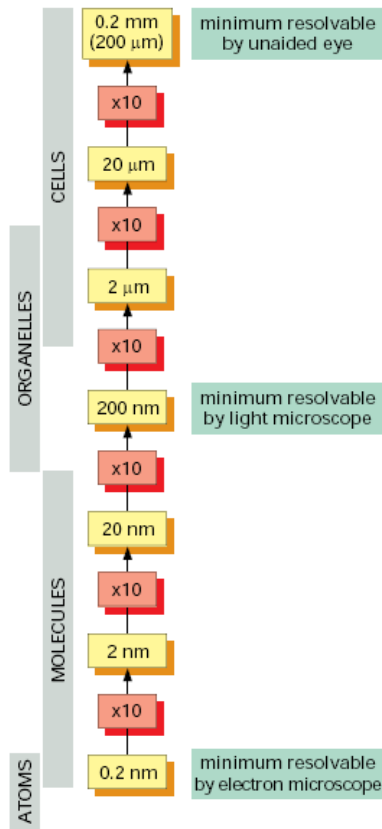


Figure 1.3: The cellular scale

The gain of studying these topics and of projecting such machines is impressive: terabytes of data storage can become usual due to miniaturization of the components (see Figure 1.3), we will handle systems that emit almost no heat, that run for days off a single battery charge, that compute faster due to massive parallelism, and possibly that perform a new type of higher order 'wet' intelligence that goes well beyond the usual notions of AI. Henceforth, we expect to happen faster (massively parallel), smaller (nano-scale), and cost efficient (energy-saving) information processing. However, one of molecular computing's biggest challenges is calculating with elements that are flawed, unreliable and decentralized.

To this extent, cellular and membrane computing stand for a proper and natural starting point for an incursion into parallel cellular machines. However, in this work, the focus will be restricted to the field of membrane systems and

especially on their computational capabilities under various constraints.

Membrane Systems

One of the newest computability model related to the area of molecular computing is based upon the notion of the membranes structure of living cells. The model was introduced in [64], and it is inspired from the fact that all the cellular-level processes involving different chemical reactions which have precise goals, can

be viewed as computing processes. In this respect, any non-trivial biological system is a construct in which different components execute “computations” in a parallel/distributed fashion, sharing the results if needed, in order to accomplish a common goal. Each component present in such a system is well delimited by others through various types of membranes that keep them separated and act as semi-permeable barriers, ensuring that certain substances always stay into the compartment while other substances stay out of it (see Figure 1.2). The exchange of substances between compartments is done only if there is an explicit request for communication and is governed by the general biological principle of causality: if there is a need for a certain substance, then some compartments responsible for producing that substance should and will provide it.

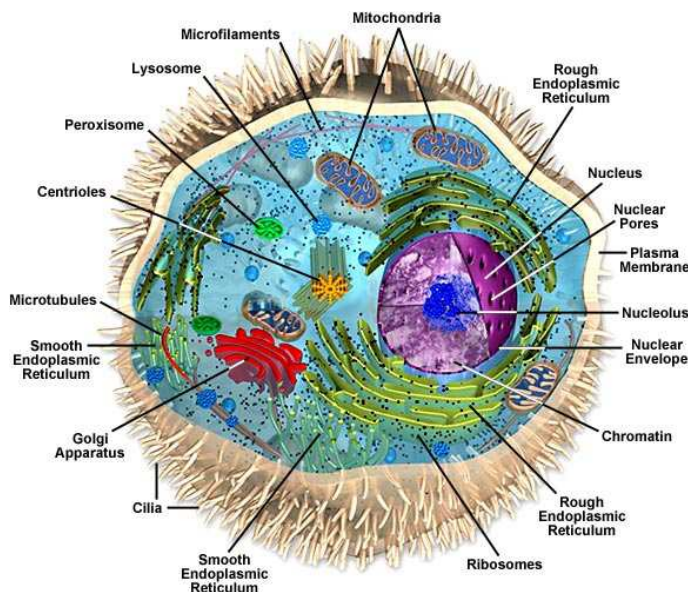


Figure 1.2: The Animal Cell - is a typical eukaryotic cell. It ranges in size between 1 and 100 micrometers and is surrounded by a plasma membrane, which forms a selective barrier allowing nutrients to enter and waste products to leave. The cytoplasm contains a number of specialized organelles, each of which is surrounded by a membrane. (Michael W. Davidson, <http://micro.magnet.fsu.edu>)

The reactions take place at the level of compartments not only by creating, transforming, eliminating substances, but also by dividing, producing or even destroying compartments. Moreover, there are systems of cells responsible for different tasks and they also communicate usually according to a “universal clock”. Therefore the whole assembly can be seen as a huge parallel device able to “compute the life” (see Figure 1.4 for a metaphoric view of the cell).

Meaningful for the present work is one fundamental topic of the cell biology – the study of how biological signals are managed by cells. Signals can arise from inside the cell or from the environment and the correct answer to certain signals is

essential for a bacteria for instance to survive in a certain environment. Starting from these biological motivations we will consider a model of membrane systems where the rules can act independently or can be controlled by signals which can be created, deleted or moved across the regions.

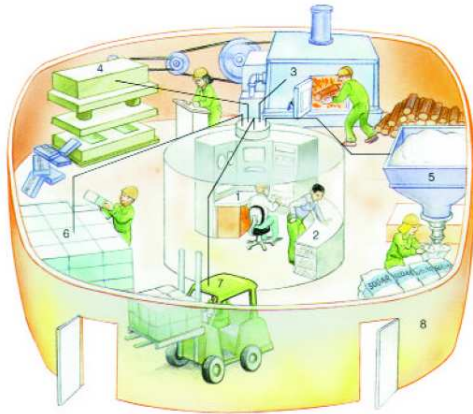


Figure 1.4: The cell resembles a big factory which contains conveyor systems, information storing centers, special compartments where chemical processes are made, energy generating power stations, and packaging centers. (Harun Yahya, *The Secrets of DNA*, 2004)

tion of surface features; they will determine their functions. If two proteins have surface regions that are complementary (in shape and charge), they may stick to each other forming a complex. The bond between the individual proteins is strong and is due to many small atomic forces.

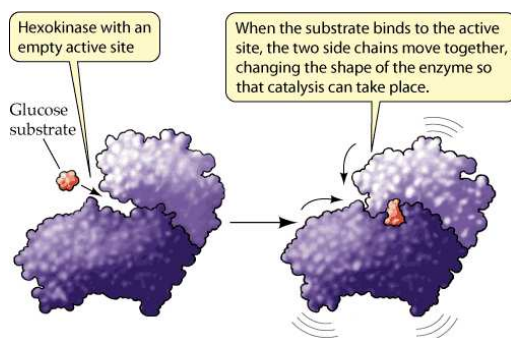


Figure 1.5: Enzyme activation many chemical reactions in the cell are catalyzed by the presence of an associated enzyme (in other words, the enzyme permits the chemical rule to happen; more

In a cell, apart from some small molecules like water or some metabolites, there are four large types of macromolecules: *nucleic acids*, *proteins*, *lipids*, and *carbohydrates*. Each class is formed by a small number of units that can be combined in order to produce biochemical structures of great complexity. Here we will focus mainly on proteins. There are about 20 kinds of amino acids that combine themselves in order to form proteins. Each protein folds in a well defined three-dimensional shape (sometimes from multiple strings of monoacids) that can be mechanically flexible. However, as we will see later on, what is important from reactions pathways viewpoint is the protein's collec-

During a complexation event, the protein may be bent or opened, hence revealing new interaction surfaces. By this phenomenon, many proteins act as enzymes: they bring together compounds, including other proteins, and greatly facilitate chemical reactions between them without being themselves affected. In biology it is known that

information on this topic can be found in [68]). On the other hand, in bacteria, the enzymes (proteins) can be activated/inactivated during the cellular process (in other words, an inactivated enzyme is not able to catalyze the corresponding reaction).

In bacteria, the *enzyme activation/inactivation*, as well as several other important processes, is controlled by covalent modification of proteins (so called the post-translational *covalent modifications of proteins*) – see [96] for details.

The covalent modification involves the attachment of objects (chemical substances) at different positions along the string represented by the protein (i.e., the enzyme). The attached object could belong to different type of substances such as phosphoryl or methyl. One basic fact is that the attachment of the object (either phosphoryl, methyl or other chemicals) occurs at specific places along the protein; during the cellular processes the substance (for example, phosphoryl) travels along the cell and it is attached to the enzyme where such enzyme must be activated. Each such site along the protein acts as a Boolean switch; over a dozen of them can be present on a single protein.

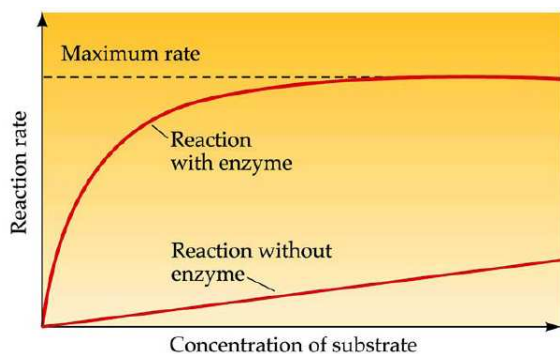


Figure 1.6: Reactions with enzymes are up to 10 billion times faster than those without enzymes. Enzymes typically react with between 1 and 10,000 molecules per second. Fast enzymes catalyze up to 500,000 molecules per second.

We can imagine the substance like a *promoter* that travels along the regions of the cell and (promotes) activates the corresponding enzyme (and then the corresponding catalyzed reaction) when it is attached/unattached to/from the enzyme.

In what follows we will briefly describe how the phosphoryl movement can control the process of enzyme activation/inactivation in *Escherichia coli*.

The enzyme activation/inactivation by using a covalent bond of phosphate (as in the case of *Escherichia coli*) was first described in mammals (liver) around half a century ago. Then, Fisher and Krebs showed that an enzyme involved in metabolism was regulated by the addition (reaction called phosphorylation) or the removal (reaction called dephosphorylation) of phosphoryl, suggesting that reversible phosphorylation could control enzyme activity.

One classical example of enzyme activation/inactivation by covalent attaching of phosphoryl occurs in *Escherichia coli* for *isocitrate dehydrogenase*. Isocitrate dehydrogenase is an enzyme which in the active state takes away two atoms of hy-

drogen (thus its name “dehydrogenase”) and one molecule of carbon dioxide from a chemical called isocitrate; thus isocitrate is converted to another chemical called 2-oxoglutarate. The enzyme isocitrate dehydrogenase is *active when phosphoryl is NOT attached* on it. The enzyme is inactivated by attaching the phosphoryl and the inactivated enzyme is not able to perform the conversion of isocitrate to 2-oxoglutarate.

The bond of phosphoryl to isocitrate dehydrogenase is catalyzed by an enzyme called kinase whereas the removal of phosphoryl from phosphorylated isocitrate dehydrogenase is catalyzed by a so called phosphatase.

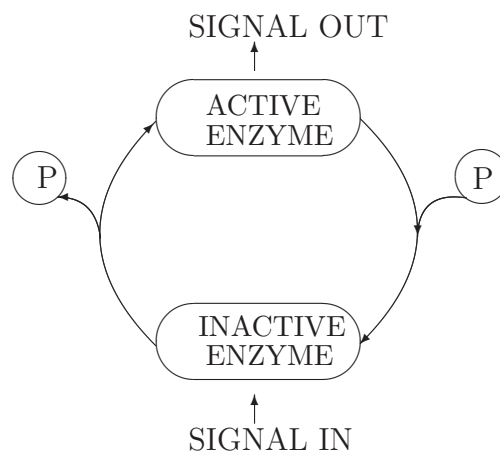


Figure 1.7: Isocitrate dehydrogenase activation in *Escherichia coli* by phosphate removal, and inactivation by phosphorylation

In particular, phosphate is attached by the kinase at a very precise position within the protein, exactly at the level of the 113th aminoacid; these reactions are reversible and they enable the cell to either activate (by dephosphorylation) or to inactivate (by phosphorylation) the enzyme isocitrate dehydrogenase. The idea of this process is described in Figure 1.7.

The phosphoryl movement around the cell can also control the so-called *two-component regulatory system* presents, for example, in *Escherichia coli*. A two-component regulatory system is a system composed by two proteins: a sensor (S) and a response regulator (RS). Such system responds to environmental signals (*stimuli*) like changes in oxygen concentration, light intensity, starvation, water activity, and so on (for more details it is possible to consult [95]).

The responses of a two-component regulatory system are essential for bacterial cells (as well as other types of cells) to survive in a given environment (for example,

in *Escherichia coli*, it seems to exist around 50 different two-component regulatory systems).

The responses of a two-component regulatory systems are based, again, on the phosphorylation/dephosphorylation of proteins described before for enzyme activation/inactivation (for details about two-component regulatory systems the reader can consult [95]).

Two-component regulatory systems and the enzyme activation/inactivation are important because they illustrate how, within the same cell (in this case, *Escherichia coli*), the attaching of the same object, phosphoryl (that moves around the cell), can modulate (activate/deactivate) different functions such as the conversion of izocitrate to 2-oxoglutare (in the case of enzyme activation/inactivation) or it can change the proportion of some transport proteins (e.g., porins) at the plasma membrane (in the case of a two-component regulatory systems).

In addition to all phenomena described above we can even more say that some reactions are possible within the cell if, for instance, the concentration of salinity (or the electrical charge, or the temperature) has a certain value. All these factors, as well as many other more, participate, evolve, and control the way reactions take place.

All these facts (but not only; actually there exists many more other examples) indicate that cells are, in many respects, information processing devices. This is way we are motivated to have an overview of these aspects by using a formal model – the membrane systems with promoters and inhibitors. Now, in order to take advantages from this massive parallel molecular device which is the cell, first, the computational capabilities of the model as well as its properties must be expressed through an underlying formal theory. Even if we will limit ourselves to some restricted model, we still can acquire valuable information regarding the functioning of the cell. Moreover, once the formal theory shows the computational capabilities of the model, algorithms that solve practical problems can be developed. And these bring us to the scope of the current work.

Once we have chosen the machine model, we will encounter a significant difficulty common to such parallel systems, namely, the challenging task one is faced with in designing them to perform a specific behavior or solve a particular problem when various constraints are imposed. However, the formal model of membrane systems is so flexible that holds potential both scientifically, as tools for studying phenomena of interest in biology, as well as practically, showing a range of potential future applications, ensuing the construction of new algorithms or even new computer

designs.

Apart from the current introduction that is a short historical presentation of computability theory along with a sharp foreword of the recent field of membrane computing and its motivations, the present work is structured in several chapters. They cover various topics related to P systems with promoters and/or inhibitors and their computational capabilities under several constraints, along the classical lines of computability theory as studied in computer science.

Chapter 2 represents a general survey of the formal language theory focusing mostly on computational universal machinery as well as on another bio-inspired formal topic – the Lindenmayer systems. Also, basic notations, definitions and results regarding grammars and automata are treated here. Notions like Chomsky hierarchy, Turing machines, register machines, regulated rewriting as well as L systems are introduced here and will be used during the work. Here we also study the computational capability of a particular form of random context grammar; the results obtained will be employed afterward when dealing with some open problems in the membrane computing field.

Chapter 3 introduces the framework of P systems. Here the focus will be on the model of membrane systems with promoters/inhibitors. First, we review the definition of the model as it was presented in the introductory paper *Membrane Systems with Promoters/Inhibitors* (see [16]). Several representative universality results are presented along with some relevant examples.

The results regarding P systems with non-cooperative and catalytic (using only one catalyst) promoted/inhibited rules at the level of rules introduce a new method to show computational universality in the P systems framework, namely the usage of regularly controlled grammars. In this respect, the result concerning P systems with promoters at the rule level, obtained using regular regulated rewriting, confirms the one presented in the Gheorghe Păun's book *Membrane Computing – An introduction*, [66], by showing that two membranes suffice in order to obtain universality. A universality proof for P systems with inhibitors (open problem number 5 in the same book) is given using a similar construction. Also, it is proved that if we use non-cooperative rules and promoters or inhibitors of weight two, then the systems are computationally universal. Moreover, the systems satisfy a special property we define: they are *ultimately confluent*. This means that if the systems allow at least one halting computation, then their final configurations are reachable from any reachable configuration. In addition, P systems with non-cooperative inhibited rules prove to generate the same family of sets of vectors as the family of Parikh images

of languages generated by ETOL systems (open problem number 6 in [66]). In what concerns P systems with non-cooperative promoted rules we did not find their exact generative power. However, we conjecture that such systems are equivalent with systems with non-cooperative inhibited rules. To support this supposition we bring out several arguments.

Many classical parallel software and hardware systems assume usually a very low level of fault tolerance, that is, they are not able to recover in the face of errors; certainly, in case of computer software, even a single malfunction can often bring an entire program to a failure. Modeling the cell (or using it as a computational device) by means of rewriting systems will imply indubitably thousands or even millions of rules. If this will be the case, the issue of resilience will become extremely important since faults will be likely to occur, and they will propagate with high probability. Chapter 4 proposes two extensions of the classical definition of P systems with promoters/inhibitors. Here we study the issue of fault tolerance, examining the resilience in what concerns the execution time and the parallelism of the rules.

To this aim we introduce the concept of *timed* P systems by associating a time value to each rule representing the time needed by the rule (or by each application of the rule) to be entirely executed. Then, we have found a proper subclass of P systems with non-cooperative and catalytic (with only one catalyst) promoted/inhibited rules that are able to generate/accept the same family of sets of vectors of numbers as the upper class, regardless the values of times associated to the rules. Such systems with the mentioned property will be called *time-independent* P systems.

Another feature present in the original P system definition is the maximal parallelism of the rewriting in one computational step. Here we question also this aspect and we investigate systems that compute using a variable parallelism of the applications of rules. We define such type of parallelism with the help of multifunctions that for a given configuration establish the number of times rules are applied. Regarding this matter, we propose two ways of performing the derivation, the *weak* and the *strong* modes, respectively. In the weak mode, the rules are applied according to the corresponding multifunctions (with competition on objects, but not forbidding the conditions given by the multifunctions); therefore, it might happen that certain symbols in a given configuration are not rewritten even if there are rules handling them because, for instance, the value set by the multifunction is zero. The strong mode of derivation assumes that rules are applied in a manner described by the multifunctions, but, in addition, each distinct symbol from the sentential form should be rewritten at least once (of course, if there are rules that can handle it); therefore

a rule might be applied once even if the value given by the multifunction was zero.

In this respect, we have relaxed the original P system definition by showing for the classes of systems we study that they contain proper subclasses able to generate/accept the same families of sets of vectors/numbers as the corresponding upper classes, regardless the rewriting parallelism in a given configuration. Systems having such property will be called *parallel-free* P systems. Here we have considered only P systems with promoters and inhibitors but the results can be furthermore extended to other models of P systems and in general to other parallel models of computation.

Chapter 5 represents a more practical approach and it consists of two sections: *Sorting with P Systems* and *Simulating Boolean Circuits with P Systems*. The first section defines the concepts of ranking and sorting order and proposes algorithms to solve these problems within the framework of P systems with promoters/inhibitors. Despite the disorder among the objects present in the multisets, the time complexity results for the algorithms presented are better than in the sequential case and this is due to the massive parallelism feature of the P systems. The algorithm used computes the result of the sorting problem by “carving” – informally, we decrease one by one each component up to the time when one component is exhausted, we trigger a signal when this happens, meaning that we have found the next component that must be eliminated. We re-iterate the process and we eliminate objects in the order of their initial multiplicity.

The second section is dedicated to the simulation, using P systems, of a Boolean circuit – another parallel model of computation. The reason of simulation is because for the Boolean circuits there are known many good algorithms for different problems (matrix multiplication, for instance) and, therefore, once we have an algorithmic construction of a P system that simulates the computations of a circuit, we will have, implicitly, the solution of the problem in terms of P systems. First the Boolean gates AND, OR, NOT are developed using P systems with promoters/inhibitors. They represent modules that can be assembled together (in the same fashion as we link Boolean gates in Boolean circuit) in order to obtain a Boolean circuit simulator. Also a synchronizer module is developed – its goal is to synchronize different branches of computation. The time complexity obtained for computing a Boolean function is proportional with the depth of the corresponding Boolean circuit.

In Chapter 6 we propose two new bio-mimetic models derived from the original P systems with promoters/inhibitors definition. *P systems with external control* model the passage of cell’s metabolites through membranes according to the charging

differences in two neighboring regions. Here, the rules of the system might be activated by the presence/absence of certain object(s) in an immediate neighboring region. For such models we have obtained both non-universality and universality results, depending on the use of catalytic rules.

String driven P systems embody the notion of the “stored program”; here the whole computation is driven by a string of promoters (the program) that travels through the membranes of the system and activates via its left-most symbol certain rules; the string loses its leftmost symbol at each passage through a membrane. Depending on the family of languages from where a language is chosen, and hence from which the string of promoters is taken, and on the features of the P systems we get various computational characterizations.

The last topic carried out in this thesis regards the usage and the help of P systems with promoters/inhibitors in attacking various problems within the general framework of P systems. In Chapter 7 we show how using the inhibiting mechanism we can simulate systems having defined a partial order on the set of rules. Thus, we show that P systems with non-cooperative rules and strong priorities are equivalent with P systems with non-cooperative inhibited rules and generate exactly *PsETOL* (open problem number 2 in [66]).

To conclude, we only have to say that even if I was inspired and motivated by some biological phenomena occurring at the cellular level, the present work was not intended to answer a specific biological problem. Rather than this, the results obtained here are relevant for computer science since in the abstraction process of cell functioning we left biology far behind. However, this does not mean that we cannot start from membrane computing and turn back to the biological reality. In this respect, there are at least by two main directions of research: implementation of membrane systems in biological media and models/simulators of cellular phenomena that can return meaningful and useful information to biologists.

Here we were mainly interested by parallel multiset rewriting under certain constraints and we have explored formal systems that perform their computations both in “flawless” and “flawed” mediums. In any case, most of the notions and results obtained along the thesis can be intuitively extended to strings and henceforth they can become relevant for the classical theoretical computer science.

Finally, at each chapter, several new open problems arose and they constitute topics for further work and research.

Chapter 2

Models of Computation Preliminaries

The present chapter represents a brief introduction to the theory of computing, sufficient for the needs of the subsequent chapters. Several computing models are presented in a concise manner together with their generative or accepting capabilities.

We present notions like Chomsky grammars, Turing machines, regulated rewriting, and Lindemayer systems; they will be used later on in characterizing the computational power of the membrane system models we study.

Finally, we introduce the concept of ultimately confluent rewriting systems – a “weaker” definition of confluent systems, with the computations “unavoidably leading” to the same result, but not necessarily bounded by the number of steps.

2.1 Languages, Grammars and Turing Machines

An *alphabet* is a finite nonempty set of symbols. A *string* over an alphabet V is a finite sequence of symbols juxtaposed. Formally, we define a string as $w = \prod_{i=0}^n a_i$ where $a_i \in V$, $1 \leq i \leq n$ (we denote by \prod the catenation operation). The length of a string is the number of symbols composing the string, i.e., $|w| = |\prod_{i=0}^n a_i| = n + 1$. The empty string is the string consisting of zero symbols and is denoted by λ .

For an alphabet V , denote by $V^* = \{\prod_{i=0}^m a_i \mid m \in \mathbb{N}, a_i \in V, 1 \leq i \leq m\}$ the set of all strings of symbols from V . By $V^+ = V^* \setminus \{\lambda\}$ we denote the set of nonempty strings over V . A *language* over the alphabet V is a subset of V^* . A language which does not contain the empty string is called λ -free language. The number of occurrences of a given symbol $a \in V$ in the word $w \in V^*$ is denoted

by $|w|_a$. $\Psi_V(w) = (|w|_{a_1}, |w|_{a_2}, \dots, |w|_{a_n})$ is called the *Parikh vector* associated with the string $w \in V^*$ where $V = \{a_1, a_2, \dots, a_n\}$. For a language $L \subseteq V^*$, $\Psi_V(L) = \{\Psi_V(w) \mid w \in L\}$ is the *Parikh mapping* associated with V . If FL is a family of languages, by $PsFL$ is denoted the family of *Parikh images* of languages in FL . A set of vectors is called *semilinear* if it can be represented as a union of a finite number of sets of the form: $\{v_0 + \sum_{i=1}^m \alpha_i v_i \mid \alpha_i \in \mathbb{N} \text{ for } 1 \leq i \leq m\}$ where $v_i \in \mathbb{N}^n$ for $0 \leq i \leq m$. A language is called semilinear if $\Psi_V(L)$ is a semilinear set.

A *multiset* over an arbitrary set X is a mapping $M : X \rightarrow \mathbb{N}$. We denote by $M(x)$, $x \in X$, the multiplicity of x in the multiset M . If the set $X = \{x_1, \dots, x_n\}$ is finite, then the multiset M can be explicitly given in the form $\{(x_1, M(x_1)), \dots, (x_n, M(x_n))\}$. The support of a multiset M is the set $supp(M) = \{x \in X \mid M(x) \geq 1\}$. A multiset M is empty when its support is empty. Let $M_1, M_2 : X \rightarrow \mathbb{N}$ be two multisets. We say that M_1 is included in M_2 (and we denote this by $M_1 \subseteq M_2$) if $M_1(x) \leq M_2(x)$, for all $x \in X$. The inclusion is strict if $M_1 \subseteq M_2$ and $M_1 \neq M_2$. The union (difference) of two multisets, $M_1 \cup M_2 : X \rightarrow \mathbb{N}$ (respectively, $M_1 \setminus M_2 : X \rightarrow \mathbb{N}$), is defined as $(M_1 \cup M_2)(x) = M_1(x) + M_2(x)$ (respectively, for $M_2 \subseteq M_1$, $(M_1 \setminus M_2)(x) = M_1(x) - M_2(x)$) for all $x \in X$. A multiset M of finite support, $\{(x_1, M(x_1)), \dots, (x_n, M(x_n))\}$ can also be represented by the string: $w = x_1^{M(x_1)} x_2^{M(x_2)} \dots x_n^{M(x_n)}$ and all permutations of this string precisely identify the objects in the support of M and their multiplicities. Moreover, the Parikh image of w , $\Psi_X(w)$ is exactly the vector $(M(x_1), \dots, M(x_n))$ of multiplicities. The cardinality of a multiset $w = x_1^{M(x_1)} x_2^{M(x_2)} \dots x_n^{M(x_n)}$ is $card(w) = M(x_1) + M(x_2) + \dots + M(x_n)$.

Let $l \in \mathbb{N}$ and $w = x_1^{t_1} \dots x_n^{t_n}$, $x_i \in X$, $t_i \in \mathbb{N}$, $1 \leq i \leq n$ a multiset over X . Then we can define the product $l * w = x_1^{l \cdot t_1} x_2^{l \cdot t_2} \dots x_n^{l \cdot t_n}$.

Consider a finite set of symbols $V = \{a_1, a_2, \dots, a_n\}$. An arbitrarily multiset rewriting rule is a pair (u, v) with $u \in V^+$, $v \in V^*$ multisets over the set V ; such a rule is typically written as $u \rightarrow v$. For a multiset rewriting rule $r : u \rightarrow v$, with $u, v \in V^*$ multisets over V , let $left(r) = u$ and $right(r) = v$.

Let $w \in V^*$ be a multiset of symbols over V and let $R = \{r_1, r_2, \dots, r_k\}$ be a set of multiset rewriting rules such that $r_i = u_i \rightarrow v_i$, with $u_i, v_i \in V^*$, $1 \leq i \leq k$, multisets of symbols over V . Denote by $R_w^{ap} \subseteq R$ the set of *applicable* multiset rewriting rules to w , that is, $R_w^{ap} = \{r \in R \mid left(r) \subseteq w\}$.

Denote by $R_w^{sap} = r_1^{t_1} r_2^{t_2} \dots r_k^{t_k}$, $t_i \in \mathbb{N}$, $1 \leq i \leq k$, a multiset over R of simultaneously applicable multiset rewriting rules to w . R_w^{sap} is any multiset such

that:

$$\bigcup_{1 \leq i \leq k} t_i * left(r_i) \subseteq w.$$

Denote by R_w^{SAP} the set of all multisets of simultaneously applicable rules to w , i.e., $R_w^{SAP} = \{R_w^{sap} \mid R_w^{sap} \in R^*\}$.

For $x = r_1^{i_1} r_2^{i_2} \dots r_k^{i_k} \in R_w^{SAP}$ let

$$D_x = \text{supp}\left(\bigcup_{i=1}^k left(r_i)\right).$$

The set D_x indicates all distinct symbols from w that are rewritten by an application of x .

Denote by

$$R_w^{MSAP} = \{x = r_1^{i_1} r_2^{i_2} \dots r_k^{i_k} \in R_w^{SAP} \mid \text{card}(D_x) = \max_{y \in R_w^{SAP}} (\text{card}(D_y))\}$$

the set of multisets of simultaneously applicable rules to w , called the *maximal component* of R_w^{SAP} .

Remark 2.1.1 *The maximal component of R_w^{SAP} contains all multisets of simultaneously applicable rules such that the rewriting of distinct symbols is maximal (in the sense of the processed objects).*

Let Y be a set of multisets over R ; we denote $Pr(Y) = \{r_1 r_2 \dots r_k \mid r_1^{t_1} r_2^{t_2} \dots r_k^{t_k} \in Y\}$. We will use the set of sets $\{X \subseteq R_w^{MSAP} \mid Pr(X) = Pr(R_w^{MSAP})\}$.

Let

$$R_w^{MAX} = \{x \in R_w^{SAP} \mid \text{there exists } y \in R_w^{SAP} \text{ such that } x \subseteq y \text{ implies } x = y\}$$

is the set of multisets of all maximal simultaneously applicable rules to w .

Remark 2.1.2 *The concept of maximal parallelism of rewriting (used, for instance, in the P systems framework) is expressed using the set $R_w^{MAX} \subseteq R_w^{SAP}$; for example, considering a P system Π in a given configuration and a region of Π containing a set of rules R that acts on the multiset w , an element in R_w^{MAX} gives a possible ensemble of rules that can be applied on w in a maximal parallel manner.*

In addition, one can remark that $R_w^{MSAP} \supseteq R_w^{MAX}$.

In a straightforward manner, the notions presented above regarding multisets and rules can be extended to strings and productions. Therefore, depending on

the context, we will make use of the same notations when expressing the rewriting of strings (i.e., notions like *the set of applicable rewriting productions to a string w* (R_w^{ap}), *the multiset of simultaneously applicable productions to a string w* (R_w^{sap}), *the set of all multisets of simultaneously applicable productions to a string w* ((R_w^{SAP})), and *the set of multisets of all maximal simultaneously applicable productions to a string w* (R_w^{MAX})) make sense for a string w and a set of productions R).

Example 2.1.1 Let $V = \{a, b, c\}$. Consider the multiset $w = aaabbbbccc$ and the set of multiset rewriting rules $R = \{r_1 : abc \rightarrow \alpha, r_2 : bcc \rightarrow \beta, r_3 : aac \rightarrow \gamma, r_4 : acccc \rightarrow \theta\}$. Then we have:

- $R_w^{SAP} = \{r_1, r_1^2, r_1^3, r_2, r_3, r_1r_2, r_1r_3, r_2r_3\}$;
- $R_w^{MAX} = \{r_3, r_1r_2, r_1r_3, r_2r_3\}$;
- $D_{r_1r_3} = D_{r_1} = \{a, b, c\}$; $D_{r_3} = D_{r_4} = \{a, c\}$;
- $R_w^{MSAP} = \{r_1, r_1^2, r_1^3, r_1r_2, r_1r_3, r_2r_3\}$.

Example 2.1.2 Let $V = \{a, b, c, d\}$. Consider the string $w = abc$ and the set of rewriting rules $R = \{r_1 : a \rightarrow \alpha_1, r_2 : a \rightarrow \alpha_2, r_3 : b \rightarrow \beta, r_4 : c \rightarrow \gamma, r_5 : d \rightarrow \theta\}$. Then we have:

$$\begin{aligned}
 R_w^{SAP} &= \{r_1, r_2, r_1^2, r_2^2, r_3, r_4, r_1r_2, r_1r_3, r_2r_3, r_1r_4, r_2r_4, r_3r_4\} \\
 &\cup \{r_1^2r_3, r_2^2r_3, r_1^2r_4, r_2^2r_4, r_1r_2r_3, r_1r_3r_4, r_2r_3r_4\} \\
 &\cup \{r_1^2r_3r_4, r_2^2r_3r_4, r_1r_2r_3r_4\}; \\
 R_w^{MAX} &= \{r_1^2r_3r_4, r_2^2r_3r_4, r_1r_2r_3r_4\}; \\
 R_w^{MSAP} &= \{r_1r_3r_4, r_2r_3r_4, r_1^2r_3r_4, r_2^2r_3r_4, r_1r_2r_3r_4\}.
 \end{aligned}$$

Some Basic Operations with Languages

The *catenation* of two languages L_1 and L_2 is $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$. We can define recurrently: $L^0 = \{\lambda\}$ and $L^{i+1} = LL^i$, $i \geq 0$. Then $L^* = \bigcup_{i=0}^{\infty} L^i$ is the **-Kleene closure*, and $L^+ = \bigcup_{i=0}^{\infty} L^i$ is the *+Kleene closure* of L .

A mapping $h : V \rightarrow U^*$, extended to $h : V^* \rightarrow U^*$ by $h(\lambda) = \{\lambda\}$ and $h(x_1x_2) = h(x_1)h(x_2)$ for $x_1, x_2 \in V^*$ is called a *morphism*. If $h(a) \neq \lambda$ for each $a \in V$, then h is a λ -free morphism.

A morphism $h : V^* \rightarrow U^*$ is called a *coding* if $h(a) \in U$ for each $a \in V$ and a *weak coding* if $h(a) \in U \cup \{\lambda\}$ for each $a \in V$. For a morphism $h : V^* \rightarrow U^*$,

we define a mapping $h^{-1} : U^* \longrightarrow \mathcal{P}(V^*)$ (and we call it an *inverse morphism*) by $h^{-1}(w) = \{x \in V^* \mid h(x) = w\}$, $w \in U^*$.

A family of languages is called a *trio* if it is closed under intersection with regular languages, λ -free morphisms, and inverse morphisms. A trio closed under union is called *semi-AFL*. A semi-AFL closed under concatenation and Kleene $+$ is called *AFL*. A trio/semi-AFL/AFL is said to be *full* if it is closed under arbitrary morphisms (and Kleene $*$ in the case of AFL's).

Grammars

We say that $y \in V^+$ is a *subword* of the word $w \in V^*$ iff w can be represented as $w = xyz$ with $x, z \in V^*$ (y is called *proper subword* iff $w \neq y$). A *rewriting system* is a finite set of rules in the form $u \rightarrow v$ where u and v are words over a finite alphabet V , indicating that an occurrence of u as a subword (word) can be replaced by v . A rewriting system will transform words in other words, languages in other languages.

A *Chomsky grammar* is a quadruple $G = (N, T, P, S)$, where N, T are disjoint alphabets (N is the *nonterminal alphabet*, T is the *terminal alphabet*), $S \in N$ is the *starting symbol*, and P is a finite set of rewriting rules (called *productions set*), that is, a subset of $(N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ (consequently, the number of nonterminals in the left part of a rule is at least 1). A rule $(u, v) \in P$ is written in the form $u \rightarrow v$.

According to the form of production rules, Chomsky grammars are classified as follows:

- type-0 grammars: rules $(u \rightarrow v)$ as above (no additional restriction on u or v is made);
- *context-sensitive* or type-1 grammars: for each $u \rightarrow v \in P$ we have $u = u_1 A u_2$, $v = u_1 x u_2$, $u_1, u_2 \in (N \cup T)^*$, $A \in N$ and $x \in (N \cup T)^+$ (the rule $S \rightarrow \lambda$ is allowed, providing that S does not appear in the right-hand members of rules in P);
- *context-free* grammars or type-2 grammars: each rule $u \rightarrow v \in P$ has $u \in N$, $v \in (N \cup T)^*$;
- *regular grammars* or type-3 grammars: each rule $u \rightarrow v \in P$ has $u \in N$, $v \in T^* \cup T^* N$.

For a Chomsky grammar $G = (N, T, P, S)$ and for $\alpha', \beta' \in (N \cup T)^*$ we say that α' directly derives into β' , and we write $\alpha' \Longrightarrow \beta'$ if there exist $\alpha_1, \alpha_2, \alpha, \beta \in (N \cup T)^*$

such that $\alpha' = \alpha_1\alpha\alpha_2$, $\beta' = \alpha_1\beta\alpha_2$ and $(\alpha \rightarrow \beta) \in P$. The set of *sentential forms* of a grammar G is $S(G) = \{\alpha \in (N \cup T)^* \mid S \Longrightarrow^* \alpha\}$ where by \Longrightarrow^* we have denoted the reflexive-transitive closure of \Longrightarrow . The *language* generated by G is the set $L(G) = S(G) \cap T^*$.

We denote by *RE* (recursive enumerable), *CS*, *CF*, *REG* the *families of languages* generated by arbitrary, context-sensitive, context-free, and regular grammars, respectively. More formally, for example

$$RE = \{L \mid \text{there is } G, \text{ grammar of type 0, such that } L(G) = L\}.$$

The following inclusions (also known as *Chomsky hierarchy*) hold:

$$REG \subset CF \subset CS \subset RE.$$

We have that:

- *RE*, *CF*, *REG* are *full* AFL's;
- *CS* is an AFL (not full).

Every context-free language is semilinear. Every context-free language over one-letter alphabet is regular. The length set of a context-free language is a finite union of arithmetical progressions.

Turing Machines

A *Turing machine* (TM) is a seven-tuple system $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where Q is a finite set of states, Σ is the input alphabet, Γ is the tape alphabet, $\Sigma \subset \Gamma$ and $Q \cap \Gamma = \emptyset$, $q_0 \in Q$ is the initial state, $B \in \Gamma \setminus \Sigma$ is the blank symbol, $F \subseteq Q$ is the set of final states, and δ is the transition function,

$$\delta : Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\}).$$

The Turing machine has a *tape* divided in cells that may store symbols from Γ (each cell may store exactly one symbol from Γ). The tape is bounded to the left and is unbounded to the right. The machine has a *finite control* and a *tape head* that can read/write one cell of the tape during a time instant. The *input* word is a word over Σ and is stored on the tape starting with the leftmost cell while all the other cells contain the blank symbol B . Initially, the tape head is on the leftmost cell and the finite control is in the state q_0 . The machine will perform moves according to the transition function. A move depends on the current cell, that is scanned by the

tape head, and on the current state of the finite control. A move consists of: change the state, write a symbol from Γ to the current cell, and move the head one cell to the left or one cell to the right. An input word is accepted iff after a finite number of moves the Turing machine enters a final state.

An *instantaneous description* of a Turing machine is a string $\alpha q X \beta$, where $\alpha, \beta \in \Gamma^*$, $q \in Q$, $X \in \Gamma$. The string encode the description of the Turing machine at a time as follows: q is the current state of the finite control, X is the content of the current cell of the tape, α is the content of the tape on the left side of the current cell, whereas β is the content of the tape on the right side of the current cell until the rightmost cell that is non-blank, i.e., all the cells to the right of β contain B .

The Turing machine M defines a direct transition relation between instantaneous descriptions, denoted by \vdash_M , in the following way:

$$\begin{aligned} \alpha q X \beta \vdash_M \alpha Y p \beta & \text{ iff } (p, Y, R) \in \delta(q, X), \\ \alpha q \vdash_M \alpha Y p & \text{ iff } (p, Y, R) \in \delta(q, B), \\ \alpha Z q X \beta \vdash_M \alpha p Z Y \beta & \text{ iff } (p, Y, L) \in \delta(q, X), \\ \alpha Z q \vdash_M \alpha p Z Y & \text{ iff } (p, Y, L) \in \delta(q, B), \end{aligned}$$

where $\alpha, \beta \in \Gamma^*$, $X, Y, Z \in \Gamma$, and $p, q \in Q$.

The transition relation, denoted \vdash_M^* is the reflexive and transitive closure of \vdash_M . The language accepted by M is:

$$L(M) = \{w \in \Sigma^* \mid q_0 w \vdash_M^* \alpha p \beta, \text{ for some } p \in F, \text{ and } \alpha, \beta \in \Gamma^*\}.$$

The family of recursively enumerable languages is characterized as follows:

$$RE = \{L \mid \text{there exists a Turing machine } M \text{ such that } L = L(M)\}.$$

A Turing machine defines an effective procedure in the intuitive sense. The converse assertion (*Church's thesis*), i.e., that each effective procedure can be defined by a Turing machine cannot be proved since it identifies a formal notion – the Turing machine, with an informal (intuitive) notion – the effective procedure. However, the equivalence of Turing machines with a wide variety of diverse models has proved that the Church thesis was not questioned. The thesis is accepted in the sense that the informal idea of an “*algorithm*” is identified with the existence of a Turing machine that halts for each input.

A Turing machine as presented above is a special purpose computer - the machine is restricted to perform one particular type of computation. On the other hand,

nowadays digital computers are general purpose machines that can execute any program they receive. To overcome this objection it was designed a reprogrammable Turing machine, called universal Turing machine. A universal Turing machine M_u is an automaton that, given as input the description of any Turing machine M and a string w , can simulate the computation of M on w .

Let us stress an important distinction here, that between computational completeness and universality. Given a class \mathcal{C} of computability models, we say that \mathcal{C} is *computationally complete* if the devices in \mathcal{C} can characterize the power of Turing machines (or of any other type of equivalent computing devices). This means that given a Turing machine M we can find an element C in \mathcal{C} such that C is equivalent with M . Thus, completeness refers to the capacity of covering the level of computability of Turing machines (in grammatical terms, this means to generate all recursively enumerable languages). Universality is an internal property of \mathcal{C} and it means the existence of a fixed element of \mathcal{C} which is able to simulate any given element of \mathcal{C} , in the way described above for Turing machines.

For further details regarding these topics we refer the reader to [44], [46], [60], [79], and [80].

2.2 Regulated Rewriting

In any Chomsky grammar, at some given step in a derivation one can use for rewriting any applicable rule in any desired place of the sentential form. In order to restrict this “nondeterminism” some regulated mechanisms which can control the derivation process were considered. Using such regulations we can arrive to computational universality even if we start with context-free grammars. In literature there are many types of regulations which restrict the use of rules in a Chomsky grammar. Here we will present only two of them which are relevant for the next chapters, namely regularly controlled grammars with appearance checking and λ rules and random context grammars with appearing checking and λ rules. For all types of grammars with regulated rewriting presented here we will use context-free rules as the generative device.

Regularly Controlled Grammars

Regularly Controlled Grammars as well as *Matrix Grammars* were introduced in a particular form already in 1965 and they turn out to be very useful for membrane computing.

A *regularly controlled context-free grammar with appearance checking* is a 6-tuple $G_{rC} = (N, T, P, S, R, F)$, where N, T, P and S are specified as in context-free grammar, R is a regular set over P , and F is a subset of P .

For a rule $p = A \rightarrow w \in P$ and $x, y \in (N \cup T)^*$ we define the application $x \xRightarrow{p}^{ac} y$ of the production p in appearing checking mode by $x = x_1 A x_2$ and $y = x_1 w x_2$, or $x = y$, A does not appear in x , and $p \in F$.

The language $L(G_{rC})$, generated by the regularly controlled context-free grammar with appearance checking G_{rC} , consists of all words $w \in T^*$ such that there is a derivation

$$S \xRightarrow{p_1}^{ac} w_1 \xRightarrow{p_2}^{ac} w_2 \cdots \xRightarrow{p_n}^{ac} w_n = w$$

with $p_1 p_2 \dots p_n \in R$.

We say that G is a regularly controlled grammar without appearance checking iff $F = \emptyset$.

By λrC , λrC_{ac} , rC and rC_{ac} we will denote the families of languages generated by regularly controlled grammars (without appearance checking), regularly controlled grammars with appearance checking, regularly controlled grammars without erasing rules (and without appearance checking), and regularly controlled grammars with appearance checking and without erasing rules, respectively.

The following results stand: $CF \subset rC \subseteq \lambda rC \subset \lambda rC_{ac} = RE$.

Random Context Grammars

The original formal study of random-context grammars is by Andries P. J. van der Walt in [94]. Since then, several papers considered their computational powers under several constraints as well as various properties (we indicate [33], [34], and [35] for further details). We start by recalling the definition and the main results concerning their computational power.

A random context grammar is a quadruple $G = (N, T, P, S)$, where N, T, S are defined as in the context-free grammars and P is a finite set of random context rules, that is, triplets of the form $(A \rightarrow \alpha, Q, R)$, where $A \rightarrow \alpha$ is a context-free rule with $A \in N$, $\alpha \in (N \cup T)^*$, and $Q, R \subseteq N$. For $x, y \in (N \cup T)^*$ we write $x \xrightarrow{rc} y$ iff $x = x_1 C x_2$, $y = x_1 \alpha x_2$ for some $x_1, x_2 \in (N \cup T)^*$, $(A \rightarrow \alpha, Q, R)$ is a triplet in P , all symbols of Q appear and no symbol of R appears in $x_1 A x_2$. We will refer Q as the permitting context and R as the forbidding context of the rule $A \rightarrow \alpha$. If the forbidding context is empty for every rule, then we speak about a random context grammar without appearance checking. The language generated by a random-context grammar $G = (N, T, P, S)$ is $L(G) = \{v \in T^* \mid S \xrightarrow{rc}^* v\}$, where

by \xRightarrow{rc}^* we have denoted the transitive and reflexive closure of \xRightarrow{rc} .

By λRC_p , λRC_f , and $\lambda RC_{p,f}$ we will denote the families of languages generated by random context grammars with only permitting fields and erasing rules, random context grammars with only forbidding rules and erasing rules, random context grammars with appearing checking and erasing rules, respectively. Omitting λ in the notation then, by RC_p , RC_f , and $RC_{p,f}$, we will denote the families of languages generated by random context grammars with only permitting fields, forbidding rules, and random context grammars with appearing checking, respectively.

The following results stand:

- $CF \subset RC_p \subseteq rC \subset RC_{p,f} = rC_{ac}$.
- $\lambda RC_f \subset RE$.
- $RC_p \subseteq \lambda RC_p \subseteq \lambda rC \subset \lambda RC_{p,f} = RE$.

For any random context grammar with appearance checking and λ rules there exists an equivalent random context grammar with appearance checking and λ rules $G' = (N', T', P', S')$ such that for the rules $(A \rightarrow \alpha, Q, R) \in P'$ we have $|\alpha| \leq 2$, $card(Q) \leq 1$, $card(R) \leq 1$.

Now, let us consider a more restrictive form of random context grammars.

Definition 2.2.1 *A random context grammar with limited checking and λ rules is a random context grammar $G = (N, T, P, S)$ with rules of the form $(A \rightarrow \alpha, Q, R)$, where $A \rightarrow \alpha$ is a context-free rule with $A \in N$, $\alpha \in (N \cup T)^*$, and $Q, R \subseteq N$, with $card(Q) + card(R) \leq 1$.*

We denote by λRC_{lc} the family of languages generated by random context grammars with limited checking and λ rules.

A natural question arises:

Open Problem: which is the generative power of random context grammars with limited checking?

Conjecture: random context grammars with limited checking are not universal, i.e. $\lambda RC_{lc} \subset RE$. One can remark that this is a “winning” problem since if such grammars prove to be universal, then we will have a new normal form for random context and matrix grammars with appearing checking. If they are not universal, then it implies as we will see that P system with promoters are not universal. We also suggest an approach that might be useful for proving the non universality of catalytic P systems with only one catalyst. Anyway, random context grammars with

limited checking are strictly more powerful than ETOL systems, as it is proved in what follows (details regarding ETOL systems can be found in Section 2.4).

Theorem 2.2.1 $\lambda RC_{lc} \supset ETOL$.

Proof. Consider an ETOL system $H = (V, T, \omega, \Delta)$ such that $V = V_N \cup \Delta$, $V_N \cap \Delta = \emptyset$, $T = \{T_1, T_2, \dots, T_k\}$ and all rules from T_i , $1 \leq i \leq k$ are in one of the forms $A \rightarrow \alpha$, $A \in V_N$, $\alpha \in V^*$, or $a \rightarrow a$, $a \in \Delta$. We will simulate the computation of the system H with a random context grammars with limited checking $G = (N_G, T_G, P_G, S_G)$.

Since in H the rules $a \rightarrow a$, $a \in \Delta$, do not affect the derivation process, we can remove them from tables. Therefore, let:

$$T_i = \{A_{(i,j)} \rightarrow \alpha_{(i,j)} \mid 1 \leq j \leq r_i, A_{(i,j)} \in V_N, \alpha_{(i,j)} \in V^*\}, 1 \leq i \leq k,$$

be the sets of rules representing the ETOL tables.

We denote:

$$N_G = (V \setminus \Delta) \cup \{t_{(i,j)} \mid 1 \leq i \leq k, 1 \leq j \leq 2 * r_i + 1\} \cup \{\bar{A} \mid A \in V \setminus \Delta\};$$

$$T_G = \Delta.$$

We define the set of rules P_G as follows. First we have the rules:

$$(S \rightarrow t_{(i,1)}\omega, \emptyset, \emptyset), \text{ with } 1 \leq i \leq k.$$

The starting symbol S is nondeterministically changed into one table “selector” ($t_{(i,1)}$, $1 \leq i \leq k$) and the string ω that corresponds to the axiom of the ETOL system to be simulated. One of these rules is applied only once at the beginning of computation.

Next we have the following rules that correspond to the rules of the tables:

$$(A_{(i,j)} \rightarrow \overline{\alpha_{(i,j)}}, \{t_{(i,j)}\}, \emptyset), \text{ with } 1 \leq i \leq k, 1 \leq j \leq r_i.$$

Basically, each rule produces “colored” (overlined) symbols in the presence of symbol $t_{(i,j)}$, $1 \leq i \leq k$, $1 \leq j \leq r_i$, representing the selected table.

The next task needed for a correct simulation is to check whether or not all symbols in the current sentential form were actually transformed into overlined ones (that corresponds to the simulation of the maximal parallelism of ETOL). In order to accomplish this issue we add to P_G the rules:

$$(t_{(i,j)} \rightarrow t_{(i,j+1)}, \emptyset, \{A_{i,j}\}), \text{ with } 1 \leq i \leq k, 1 \leq j \leq r_i.$$

In fact, there are needed even a smaller number of rules of the above type. This is due to the fact that our system might contain more rules with the same symbol on the left-hand side. However, the reason for adopting such numbers was just for having a simpler notation.

In case there are still remaining not overlined symbols, a symbol $t_{(i,r_i+1)}$, $1 \leq i \leq k$, $1 \leq j \leq r_i$, is not produced.

We also have the rules:

$$(\overline{A_{(i,j)}} \rightarrow A_{(i,j)}, \{t_{(i,r_i+1)}\}, \emptyset), 1 \leq i \leq k, 1 \leq j \leq r_i.$$

Their role is to change back the overlined symbols into “regular” ones. Now, in order to simulate the selection of a new table and to iterate the process, we have to be sure that all overlined symbols have been changed back into “regular” ones.

The procedure is accomplished, in a similar way, by the rules:

$$(t_{(i,j)} \rightarrow t_{(i,j+1)}, \emptyset, \{\overline{A_{(i,j-r_i)}}\}), 1 \leq i \leq k, r_i + 1 \leq j \leq 2 * r_i.$$

Finally, if everything worked as we expected (recall that the simulation is non-deterministic and if the derivation went in a “wrong” way, then it is blocked), then we have produced a symbol $t_{(i,2*r_i+1)}$, $1 \leq i \leq k$. Next, we can use this symbol to repeat the table selection mechanism or to stop nondeterministically the generative core. These actions are achieved by making use of the rules:

$$\begin{aligned} (t_{(i,2*r_i+1)} &\rightarrow t_{(h,1)}, \emptyset, \emptyset), \text{ with } 1 \leq i, h \leq k, \\ (t_{(1,2*r_i+1)} &\rightarrow \lambda, \emptyset, \emptyset). \end{aligned}$$

Because we consider in the language only terminal strings we have $L(H) = L(G)$, hence the inclusion $\lambda RC_{lc} \supseteq ET0L$ is proved.

In order to prove that the inclusion is strict we have to construct a language that cannot be generated by an ET0L system. To this aim, we construct $G = (V, T, P, S)$ that generates the language $\{(a^n b)^m \mid m \geq n \geq 1\}$ defined as follows:

- $V = \{S, A, B, C, D, X, Y, N, N', P, P', P'', R, Q, Z, a, b\}$,
- $T = \{a, b\}$,
- P is defined as:

$$\begin{array}{ll}
(S \rightarrow AX, \emptyset, \emptyset), & (P' \rightarrow P'', \emptyset, \{A\}), \\
(A \rightarrow BB, \{X\}, \emptyset), & (P'' \rightarrow Q, \emptyset, \{N'\}), \\
(X \rightarrow YN, \emptyset, \{A\}), & (C \rightarrow bD, \{Q\}, \emptyset), \\
(B \rightarrow A, \{Y\}, \emptyset), & (Q \rightarrow R, \emptyset, \{C\}), \\
(Y \rightarrow X, \emptyset, \{B\}), & (D \rightarrow C, \{R\}, \emptyset), \\
(X \rightarrow P, \emptyset, \emptyset), & (R \rightarrow Q, \emptyset, \{D\}), \\
(A \rightarrow aC, \{P\}, \emptyset), & (Q \rightarrow Z, \emptyset, \{C\}), \\
(P \rightarrow P', \{N'\}, \emptyset), & (D \rightarrow \lambda, \{Z\}, \emptyset), \\
(N \rightarrow N', \emptyset, \emptyset), & (Z \rightarrow \lambda, \emptyset, \emptyset). \\
(N' \rightarrow \lambda, \emptyset, \emptyset), &
\end{array}$$

The details of the construction are left to the reader. Consequently, we have that $\lambda RC_{lc} \supset ET0L$. \square

2.3 Register Machines

We will use in this work also Minsky's register machines, that is why we recall here this notion. Such a machine runs a program consisting of numbered (in a one to one manner) instructions of several simple types. Several variants of register machines with different number of registers and different instructions sets were shown to be computationally universal. For more details regarding this topic we refer the reader to [56], [55], and [89].

An n -register machine is a construct $M = (n, \mathcal{P}, l_0, l_h)$, where:

- n is the number of registers;
- \mathcal{P} is a set of labeled instructions of the form $(l_1 : op(r), l_2, l_3)$, where $op(r)$ is an operation on register r of M , and l_1, l_2, l_3 are labels from the set $Lab(\mathcal{P})$ (where $Lab(\mathcal{P})$ denotes the set of labels of the instructions from \mathcal{P});
- l_0 is the initial label;
- l_h is the final label.

The operations allowed by an n -register machine are:

$(l_1 : ADD(r), l_2, l_3)$ – increment the value stored into register r and proceed, in a non-deterministic way, to the instruction labeled l_2 or to the instruction labeled l_3

($l_2 = l_3$ for the deterministic variant and then the instruction is written in the form ($l_1 : \text{ADD}(r), l_2$));

($l_1 : \text{SUB}(r), l_2, l_3$) – jump to instruction labeled l_3 if the register r is empty; otherwise subtract one from the value stored into register r and jump to instruction labeled l_2 ;

($l_h : \text{HALT}$) – halts the computation (there is an unique halting instruction).

A register machine $M = (n, \mathcal{P}, l_0, l_h)$ with $n \geq 3$ accepts a vector $(r_1, \dots, r_{n-2}) \in \mathbb{N}^{n-2}$ iff, starting from the instruction labeled l_0 , with register j having value r_j for $1 \leq j \leq n-2$, and the contents of registers $n-1$ and n being empty, the machine halts (with all its registers empty). If the machine does not halt, the analysis was not successful. Deterministic register machines of this type accept exactly the family of Turing computable sets of vectors of natural numbers.

2.4 Lindenmayer Systems

Lindenmayer systems were proposed by Aristid Lindemayer as a tool for modeling the growth processes of plant development. Later on, they were used for the morphology of a variety of organisms. Much has been written on L systems; for an extensive overview on L systems we refer the reader to [36], [78], [77], and [76]. Here we present the basic definitions and several results that will be used during the present work.

A 0L system is a triplet $H = (V, P, \omega)$, where V is a finite alphabet, P is a set of context-free rules over V , and $\omega \in V^*$ is the axiom. The set of rules P has to be complete, i.e., for each symbol $a \in V$ there must be at least one rule $a \rightarrow \alpha \in P$ with this letter a on the left-hand side.

0L systems use parallel derivations, i.e., x directly derives y in a 0L system $H = (V, P, \omega)$, with $x, y \in V^*$, written as $x \xrightarrow{OL}_H y$, if $x = x_1x_2 \dots x_n, y = y_1y_2 \dots y_n$, where $x_i \in V, y_i \in V^*$, and $x_i \rightarrow y_i \in P, 1 \leq i \leq n$.

A T0L system is a triplet $H = (V, T, \omega)$, where V is a finite alphabet, $T = \{T_1, \dots, T_k\}$ is a finite set of tables over V , where each table $T_i, 1 \leq i \leq k$, is a complete set of context-free rules over V , and $\omega \in V^*$ is the axiom. We say that x directly derives y in a T0L system $H = (V, T, \omega)$, with $x, y \in V^*$, written as $x \xrightarrow{TOL}_H y$, if $x \xrightarrow{OL}_{H_i} y$ for some $i, 1 \leq i \leq k$, with the 0L system $H_i = (V, T_i, \omega)$.

An ET0L system is a quadruple $H = (V, T, \omega, \Delta)$, where $\overline{H} = (V, T, \omega)$ is a T0L system, and $\Delta \subseteq V, \Delta \neq \emptyset$, is the terminal alphabet. In an ET0L system

$H = (V, T, \omega, \Delta)$, x directly derives y , with $x, y \in V^*$, written as $x \xrightarrow{ETOL}_H y$, if $x \xrightarrow{TOL}_H y$. The transitive and reflexive closure of \xrightarrow{ETOL}_H is denoted by $\xRightarrow{*}_H$. The generated language of the ETOL system H (denoted by $L(H)$) is $L(H) = \{w \in \Delta^* \mid \omega \xRightarrow{*}_H w\}$. Thus, in an ETOL system only words over a distinguished sub-alphabet are in the generated language. A language is said to be an ETOL language if there is an ETOL system generating it.

TOL and 0L systems are particular cases of ETOL systems: $\Delta = V$ stands in both cases; in addition, for 0L systems $T = \{T_1\}$. In an EOL system there is only one table but Δ is not necessarily equal to V . Therefore the above definition gives the generated language for these systems as well.

A 0L system is a TOL system that has only one table. Adding a number of tables yields an infinite hierarchy of subclasses of the class of TOL languages. In contrast, every ETOL language can be generated by an ETOL system with only two tables.

The families of 0L and TOL languages are not closed with respect to AFL operations. The family of EOL languages is closed with respect to all AFL operations with the exception of inverse homomorphism. The family *ETOL* is closed under all AFL operations.

The membership problem for ETOL systems is NP-complete.

The following results stand: $PsREG = PsCF \subset PsETOL \subset PsRE$.

2.5 Deterministic and Confluent Rewriting

For a rewriting system M , we write $C \Rightarrow C'$ if the system allows a direct transition from an instantaneous description C to an instantaneous description C' (C' is then called a next instantaneous description of C). The relation \Rightarrow^* is a reflexive and transitive closure of \Rightarrow . For any rewriting system, we use the word *configuration* to mean any instantaneous description C , reachable from the starting one.

Definition 2.5.1 *A configuration of a rewriting system is called halting if no rules of the system can be applied to it. A computation is considered successful if it halts.*

In this dissertation we will only talk about the rewriting systems, which produce the result at halting. However, in membrane systems literature there exists also the concept of halting at equilibrium, namely the system is still able to execute rules but its configuration does not change.

Definition 2.5.2 A rewriting system is called *deterministic* if for every accessible non-halting configuration C the next configuration is unique.

Definition 2.5.3 A rewriting system is called *confluent* if for each starting configuration C_0 , either all the computations are non-halting, or there exists a configuration C_h and $m \in \mathbb{N}$ such that $C_0 \Longrightarrow^m C_h$.

Notice that for a successful computation, starting at some configuration C all the computations halt in at most $m \geq 0$ steps.

We will now introduce a weaker definition of a property of systems, with the successful computations “unavoidably leading” to the same result, but not necessarily bounded by the number of steps.

Definition 2.5.4 A rewriting system is called *ultimately confluent* if for each starting configuration C_0 , either all the computations are non-halting, or there exists a configuration C_h such that $C_0 \Longrightarrow^* C_h$.

For successful computations this property implies two facts:

- the halting configuration is unique (C_h),
- C_h is reachable from any configuration.

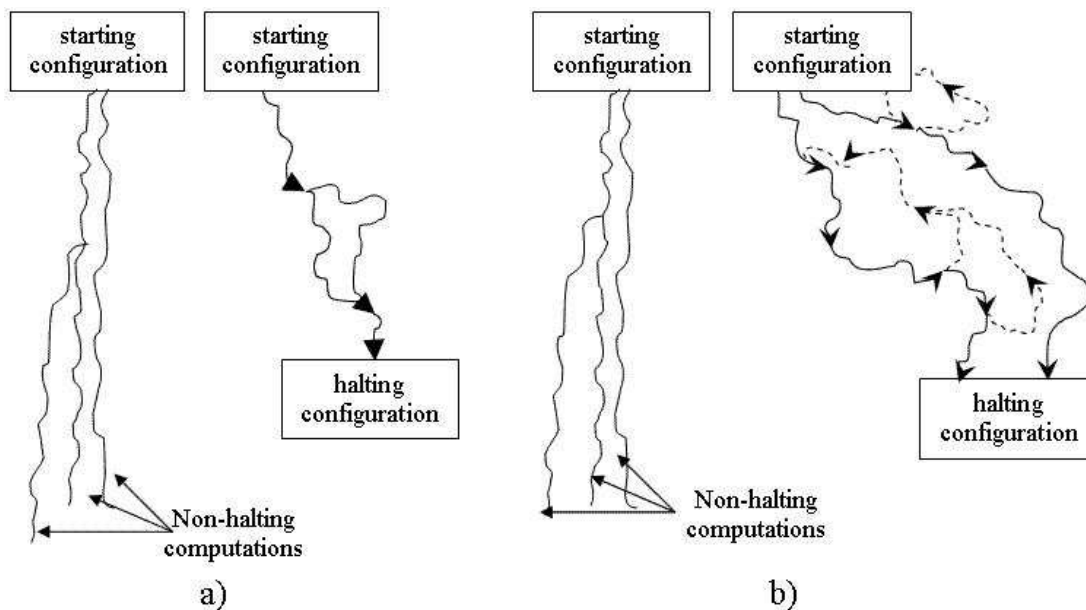


Figure 2.1: The confluent (a) and the ultimately confluent (b) computations

From now on we will only consider rewriting systems producing a result at halting. Let us consider the graph of all reachable configurations (the arc from configuration C to configuration C' means that C can derive C' in one step). The graph may be infinite. A node is called *final* if it has out-degree 0.

A system is *deterministic* if all nodes have the out-degree at most one (hence, there is at most one final node). A system is *confluent* if either there are no final nodes, or the final node is unique, and in that case the graph is finite and does not contain cycles. The graph of *ultimately confluent* systems may contain cycles, but either all nodes are non-final, or there is a final node reachable from any configuration.

In Figure 2.1 we graphically show the difference between confluent and ultimately confluent computations.

Example 2.5.1 : *Consider a system with the initial configuration S and the following rewriting rules:*

$$S \rightarrow SA,$$

$$A \rightarrow \lambda,$$

$$S \rightarrow a.$$

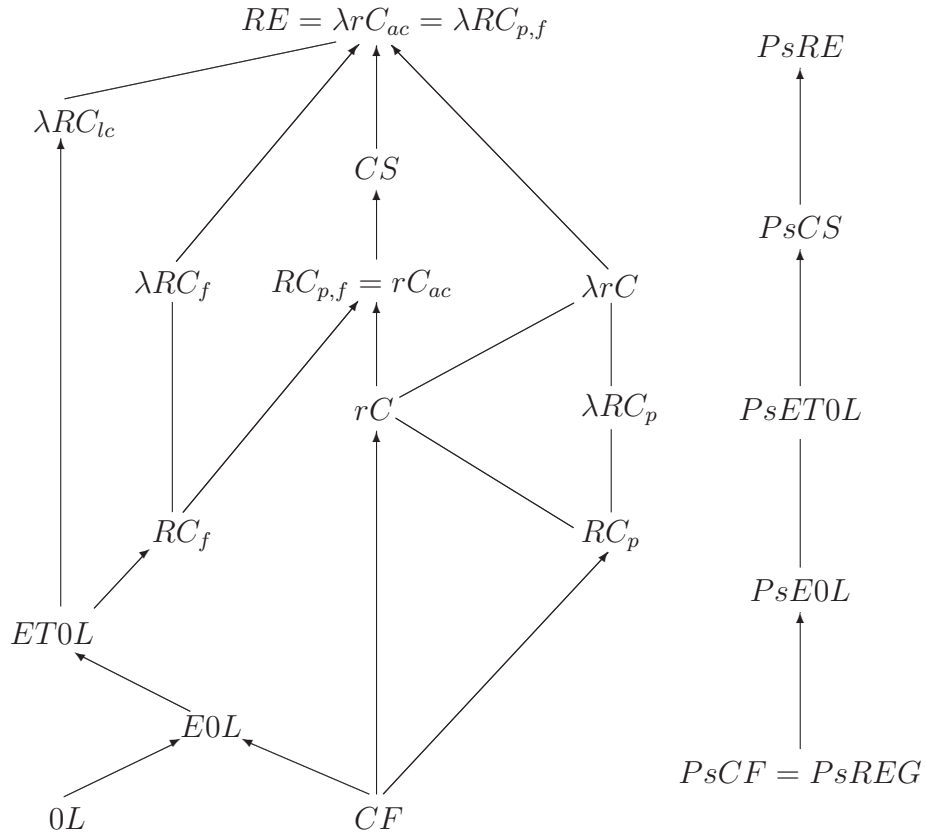
Note that the system is not deterministic, and one can choose to apply the first rule an unbounded number of times, but from any configuration it is possible to arrive to the halting one $C_h = a$ by erasing all symbols A and applying the second rule (this is equally true no matter if the system is sequential, concurrent or maximally parallel).

The definition of ultimately confluent systems can be further extended such that we allow more final nodes instead of only one. However, for the universal P systems we study in the present thesis, we conjecture that any such computation can be transformed into an ultimately confluent one (for instance, for deterministic P systems the time complexity for the transformation is linear).

2.6 Summary

Here we summarize the results mentioned in this chapter by expressing the existing relations between various families of languages. We also indicate which are the relations among several families of Parikh images of languages. These results will be extensively used during the present work. If two families are connected by a line

(an arrow), then the upper family includes (includes properly) the lower family; if two families are not connected then they are not necessarily incomparable.



Chapter 3

P Systems

A Simple Computational Model

This chapter introduces the standard computational model inspired by the way in which the cell “computes” – the membrane systems (P systems) model. Here we fix the general framework in which computations are performed by expressing the syntax and the semantics of the model.

We will focus mainly on a particular P systems model, namely *P systems with promoters and inhibitors*. For this model we will present several variants and we will investigate their computational capabilities.

3.1 P Systems with Promoters/Inhibitors

Being inspired by the membrane structure of living cells, the P system model provides a spatial structure for molecular computations.

A membrane structure consists of several membranes hierarchically embedded in an outermost membrane, called the *skin* membrane. Every membrane encloses a *region* which may contain other membranes. Any region may enclose also some *objects* (in the multiset sense) and operators – *evolution rules* for objects.

Let us define more formally what a membrane structure is. First consider the language MS over the alphabet $V = \{ [,] \}$, whose strings are recurrently defined as:

- $[] \in MS$;
- if $\mu_1 \dots \mu_n \in MS$, $n \geq 1$, then $[\mu_1 \dots \mu_n] \in MS$;
- nothing else is in MS .

Let us define the relation \sim over the elements of MS :

$x \sim y$ iff $x = [\cdots [\cdots]_2 [\cdots]_3 \cdots]_1$, $y = [\cdots [\cdots]_3 [\cdots]_2 \cdots]_1$ with $x, y \in MS$.

We will denote by \sim^* the equivalence relation obtained if we consider the reflexive and transitive closure of the relation \sim . By \overline{MS} we denote the set of equivalence classes of MS with respect to the relation \sim^* . The elements of \overline{MS} are called *membrane structures*. A *membrane* is a matching pair of parentheses which appear in a membrane structure. The most external membrane of a membrane structure is called the *skin membrane*. A membrane that does not contain inside other membranes is called an *elementary membrane*.

The number of membranes in a membrane structure μ is called the *degree* of μ and is denoted by $deg(\mu)$. The *depth* of a membrane structure μ is denoted by $dep(\mu)$ and is defined recurrently by:

- if $\mu = []$ then $dep(\mu) = 1$;
- if $\mu = [\mu_1 \cdots \mu_n]$, for some $\mu_1, \dots, \mu_n \in MS$, then

$$dep(\mu) = \max\{dep(\mu_i) \mid 1 \leq i \leq n\} + 1.$$

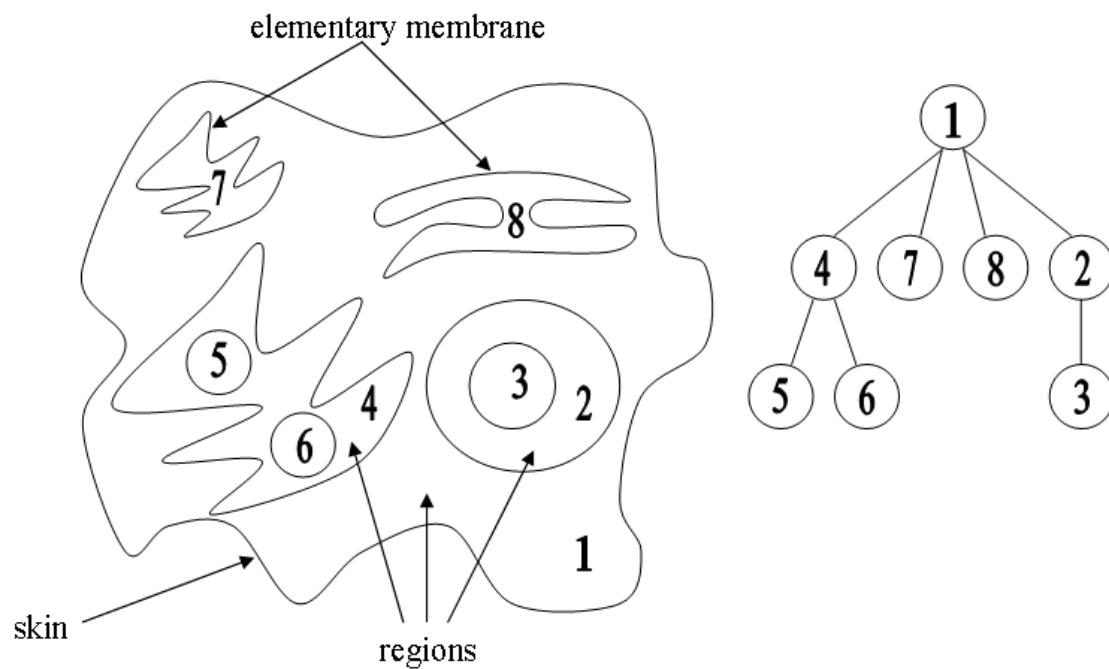


Figure 3.1: A P system diagram and its corresponding tree structure

In the regions of a membrane structure we have objects, corresponding to the chemicals and rules corresponding to the chemical reactions. The objects can be

identified by symbols from a given alphabet or by strings over a given alphabet. Then, we can work with sets or with multisets of objects. In this thesis we only deal with multisets of symbol objects.

P systems with symbol objects turn out to be a convenient framework to describe parallel computations based on molecular interaction. Such a device can be formally defined as follows.

Definition 3.1.1 *A *P* system (of degree $m \geq 1$) with symbol objects and rewriting evolution rules is a construct*

$$\Pi = (V, C, P, I, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

where:

- *V* is the alphabet of Π ; its elements are called objects;
- $C \subseteq V$ is the set of catalysts;
- $P \subseteq V$, $P \cap C = \emptyset$ is the set of promoters;
- $I \subseteq V$, $I \cap C = \emptyset$ is the set of inhibitors;
- μ is a membrane structure consisting of m membranes labeled $1, 2, \dots, m$;
- w_i , $1 \leq i \leq m$, specifies the multiset of objects present in the corresponding region i at the beginning of a computation;
- R_i , $1 \leq i \leq m$, are finite sets of evolution rules over V associated with regions $1, 2, \dots, m$ of μ ; we have rules of the following types:

non-cooperative rules, of the form $a \rightarrow v$ where a is an object from $V \setminus C$ and v is a string over $\{b_{\text{here}}, b_{\text{out}}, b_{\text{in}_j} \mid b \in V \setminus C, 1 \leq j \leq m\}$;

Catalytic rules, of type $ca \rightarrow cv$, where a is an object from $V \setminus C$, $c \in C$, and v is a string over $\{b_{\text{here}}, b_{\text{out}}, b_{\text{in}_j} \mid b \in V \setminus C, 1 \leq j \leq m\}$;

Promoted rules, of type $a \rightarrow v|_p$ and $ca \rightarrow cv|_p$, with $p \in P^+$, $c \in C$, a is an object from $V \setminus C$ and v is a string over $\{b_{\text{here}}, b_{\text{out}}, b_{\text{in}_j} \mid b \in V \setminus C, 1 \leq j \leq m\}$;

Inhibited rules, of type $a \rightarrow v|_{-i}$ and $ca \rightarrow cv|_{-i}$, with $i \in I^+$, $c \in C$, a is an object from $V \setminus C$ and v is a string over $\{b_{\text{here}}, b_{\text{out}}, b_{\text{in}_j} \mid b \in V \setminus C, 1 \leq j \leq m\}$;

A particular case is that one uses only target indications here, out, in (hence not in_j , $1 \leq j \leq m$);

- i_0 is a number between 0 and m and specifies the output region of Π (in case of 0, the environment is used for the output).

The result of applying a rule $u \rightarrow v$ (or $u \rightarrow v|_p$, $u \rightarrow v|_{-i}$) is determined by v :

- if an object a appears in v in the form a_{here} , then it will remain in the same region;
- if an object a appears in v in the form a_{out} , then it will exit the region to become an element of the immediately upper region;
- if an object a appears in v in the form a_{in_q} then it will be added to the multiset corresponding to region q , providing that a is adjacent to the membrane q .

A promoted rule $u \rightarrow v|_p$ can be applied only in the presence of multiset p . An inhibited rule $u \rightarrow v|_{-i}$ can be applied only in the absence of multiset i . In particular, promoters/inhibitors themselves can evolve according to some rules.

The difference between catalysts and promoters consists in the fact that the catalysts directly participate in rules (but are not modified by them), and they are counted as any other objects, so that the number of applications of a rule is as big as the number of copies of the catalyst, while in the case of promoters, the presence of the promoter objects makes it possible to use the associated rule as many times as possible, without any restriction; moreover, the promoting objects are not necessarily directly participating in the rules. As a consequence, one can notice that the catalysts inhibit the parallelism of the system while the promoters/inhibitors only guide the computation process.

Starting from an initial configuration, the system evolves according to the rules and objects present in the membranes, in a non-deterministic maximally parallel manner, and according with a universal clock. More formally, the m -tuple of multisets of objects present at any moment in the m regions of Π constitutes the *configuration* of the system at that moment. The m -tuple (w_1, w_2, \dots, w_m) constitutes the initial configuration of Π .

For two configurations $C_1 = (w'_1, w'_2, \dots, w'_m)$ and $C_2 = (w''_1, w''_2, \dots, w''_m)$ of Π , we can define the transition from C_1 to C_2 if we can pass from C_1 to C_2 by using the evolution rules from R_1, \dots, R_m in the regions $1, \dots, m$.

The objects that can evolve in a step as well as the rules by which they evolve are chosen in a non-deterministic manner and also, in case of rules, in a maximally parallel manner. 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 assignment, no further rule can be applied to the remaining objects. The objects which remain unassigned are left where they are, and they are passed unchanged to the next configuration.

A *computation* of a P system Π is a sequence of transitions between configurations. The system will make a successful computation if and only if it halts: there is no rule applicable to the objects present in the halting configuration. In case of generative P systems, the result of a successful computation is the number (or the vector of numbers) of objects present in the output membrane in a halting configuration of Π . If a fixed elementary membrane is specified as the input membrane and an initial multiset w_0 is present there at the beginning of computation then, a P system is said to accept the vector $\Psi(w_0)$ or the number $card(w_0)$ if it halts.

If the computation never halts, then we will have no output.

By $racOP_m(f)$ we denote the family vector sets ($\alpha = Ps$) or number sets ($\alpha = N$), which are generated (c is omitted) or accepted (with internal input, $c = I$) by P systems with symbol-objects, restricted to satisfy property r (omitted if none), with at most m membranes with the list of features f .

The features considered in this work are *ncoo* (with non-cooperative object rewriting rules), catalytic rewriting rules cat_k (using at most k catalysts), $proR_i$ (with promoters of weight at most i), and $inhR_i$ (with inhibitors of weight at most i). The P systems can be restricted to be deterministic ($r = D$). We also introduce the classes of confluent ($r = C$) and ultimately confluent ($r = U$) P systems.

For instance, in this chapter we will study the following classes:

$$\begin{aligned} &PsOP_1(ncoo, proR_1), PsOP_1(ncoo, inhR_1), \\ &DPsIOP_2(cat_1, proR_1), DPsIOP_2(cat_1, inhR_1), \\ &PsOP_1(ncoo, proR_2), PsOP_1(ncoo, inhR_2), \\ &UPsIOP_1(ncoo, proR_2), UPsIOP_1(ncoo, inhR_2). \end{aligned}$$

For more details concerning P systems with promoters/inhibitors at the level of rules or sets of rules we indicate the introductory paper [16]. Several other papers considered similar formalisms; just to mention few of them, we only cite [6], [83], and [84].

However, the method presented above for generating the languages (by counting the number of objects from an output region at the end of a successful computation) is not the only one possible. In fact, in Section ‘‘Sorting with P systems’’ we have considered that the language generated contains words obtained by catenation of objects eliminated in the environment in a successful computation (when two or more symbols leave the skin membrane in the same time, all possible combinations are in the language). For further information concerning different other methods of generating languages we indicate [66].

We will start our endeavor by recalling some results that can be found in *P* systems literature.

In [32] was shown that *P* systems with only non-cooperative rules generate exactly the length sets of context-free languages, hence the family of semilinear sets of numbers. In this case, the hierarchy on the number of membranes collapses at the first level. Therefore we have the result: $PsOP_1(ncoo) = PsCF$.

In addition, we have that *P* systems with cooperative rules generate/accept the family of Parikh images of recursive enumerable languages, i.e. $PsOP_1(coo) = PsRE$.

In [40] was proved that two catalysts suffice in order to obtain computationally complete generative systems, i.e. $NO P_2(cat_2) = NRE$.

Several open problems regarding the computational power of *P* systems with “mild” features (between the context sensitivity of catalytic rules and of promoting or inhibiting contexts) remained unsolved. In [16] was shown that *P* systems with promoted rules and one catalyst are computationally complete. Starting from this result we investigate a number of *P* systems with promoters/inhibitors variants.

3.2 *P* Systems with Non-Cooperative Promoted/Inhibited Rules

In this section we will consider *P* systems with symbol objects, non-cooperative rules, inhibitors at the level of rules, and generating sets of vectors of numbers. The family of sets of vectors generated by systems with at most $m \geq 1$ membranes is denoted by $PsOP_m(ncoo, inhR_1)$. In order to prove the main result of the section, that is $PsOP_m(ncoo, inhR_1) = PsETOL$, we will accomplish the following plan:

- show the equivalence between *P* systems with non-cooperative inhibited rules using m membranes, and *P* systems with non-cooperative inhibited rules and only one membrane;
- show that any *P* system with non-cooperative inhibited rules is equivalent with a *P* system with non-cooperative inhibited rules having the alphabet made out of two disjoint sets, the set of terminals and of non-terminals; in addition, all rules have a non-terminal on their left-hand side; moreover, the set is complete, i.e. for each nonterminal there exists at least one rule having it on the left-hand side;

- for a given set of inhibited rules, define saturated classes of rules, i.e, find the sets containing rules that do not mutually forbid each other;
- show that $PsOP_m(ncoo, inhR_1) = PsET0L$ by proving the double inclusion.

Here is how we proceed:

Theorem 3.2.1 $PsOP_m(ncoo, inhR_1) = PsOP_1(ncoo, inhR_1)$, for $m \geq 2$.

Proof. The inclusion $PsOP_m(ncoo, inhR_1) \supseteq PsOP_1(ncoo, inhR_1)$ is trivial. For the proof of the inclusion $PsOP_m(ncoo, inhR_1) \subseteq PsOP_1(ncoo, inhR_1)$, we construct a P system $\Pi_1 = (V, C, P, I, \mu, w, R, i_0)$ that simulates the computation of P system $\overline{\Pi}_m = (\overline{V}, \overline{C}, \overline{P}, \overline{I}, \overline{\mu}, \overline{w}_1, \dots, \overline{w}_m, \overline{R}_1, \dots, \overline{R}_m, i_0)$ in the following way.

First, denote by $\mathcal{L} = \{1, 2, \dots, m\}$ the set of labels of the regions in $\overline{\Pi}_m$. Then, we define:

- $V = \{a_i \mid a \in \overline{V}, i \in \mathcal{L}\}$;
- $C = \overline{C} = P = \overline{P} = \emptyset$;
- $I \subseteq V$;
- $\mu = []_1$;

Let $h : \overline{V}^* \times \mathcal{L} \rightarrow V^*$ be a mapping such that

- 1) $h(a, i) = a_i, a \in \overline{V}, i \in \mathcal{L}$;
- 2) $h(\lambda, i) = \lambda$, for all $i \in \mathcal{L}$;
- 3) $h(x_1x_2, i) = h(x_1, i)h(x_2, i), x_1, x_2 \in \overline{V}^*, i \in \mathcal{L}$.

- denote by $w = h(\overline{w}_1)h(\overline{w}_2) \dots h(\overline{w}_m)$, where \overline{w}_i is the multiset present in region $i \in \mathcal{L}$ of $\overline{\Pi}_m$ at the beginning of the computation;
- R is defined as follows. For each rule $a \rightarrow \alpha|_{-b} \in \overline{R}_i, a, b \in \overline{V}, \alpha$ is a string over $\{c, c_{out}, c_{in} \mid c \in \overline{V}\}, i \in \mathcal{L}$, we add to R the rule $h(a, i) \rightarrow \alpha'|_{-h(b, i)}$ where α' is the corresponding string over $\{h(c, i), h(c, j), h(c, k) \mid c \in \overline{V}, i, j, k \in \mathcal{L}\}, j$ being the label of the outer region of i , and k being the label of the inner region of i ;
- $i_0 = 1$.

In other words, for the P system with a single region that simulates a P system with m regions, we have encoded the region labels into objects (the subscript associated to an object indicates the region where the corresponding object belongs) and we have expressed the rules of regions by the corresponding encoded objects. In this way we ensured that, when simulating $\overline{\Pi}_m$ with Π_1 , both the parallelism at the level of regions and at the level of whole system $\overline{\Pi}_m$ is respected. In addition, one can remark that whenever $\overline{\Pi}_m$ halts, Π_1 halts as well. Moreover, when Π_1

halts, we will have in the output region of Π_1 all the objects corresponding to the multisets present in all regions of $\overline{\Pi_m}$. However, in the output multiset w_{Π_1} of Π_1 we can distinguish the output multiset $w_{\overline{\Pi_m}}$ of $\overline{\Pi_m}$ because we know which are the objects corresponding to the output region of $\overline{\Pi_m}$ (they are the objects that have as index $\overline{i_0}$). Therefore, we have to delete the unnecessary objects that remain in the output region of Π_1 in a halting configuration since we want to show that Π_1 and $\overline{\Pi_m}$ generate exactly the same set of vectors of numbers. We will modify the rules presented above in the following manner.

We add to the vocabulary V a new symbol D (the object D stands for the “deletion command”) and we replace each rule $a_i \rightarrow \alpha' |_{-b_i} \in R$ by

$$a_i \rightarrow \alpha' D |_{-b_i} \in R,$$

and each rule $a_i \rightarrow \alpha' \in R$ by

$$a_i \rightarrow \alpha' D \in R, \text{ respectively.}$$

In addition, we add the following rules

$$D \rightarrow \lambda,$$

$$a_i \rightarrow \lambda |_{-D}, \text{ for all } a_i \in V, i \neq \overline{i_0}.$$

One can remark that in this way we produce at each computational step at least one object D and also, in the same time, we delete the already existing object(s) D . If there exist rules that can be executed (i.e. there will be objects D) rules of type $a_i \rightarrow \lambda |_{-D}$ cannot be applied. When the computation halts, objects D are not produced anymore, and so, the deletion rules can start and erase the remaining unnecessary objects. Consequently we have shown that both systems generate the same family of vectors of natural numbers, hence we have $PsOP_m(ncoo, inhR_1) \subseteq PsOP_1(ncoo, inhR_1)$. \square

As a consequence of the proof above we have the following corollary:

Corollary 3.2.1 *For any P system Π with inhibited non-cooperative rules there exists an equivalent P system Π' with inhibited non-cooperative rules and with the same membrane structure like Π such that, for any halting configuration of Π' , all regions of Π' , excepting the output one, are empty.*

Remark 3.2.1 *Later we will show that the computational power of P systems with inhibited non-cooperative rules is exactly $PsET0L$. Therefore, one can prove the above result also by using the fact that the family $ET0L$ is closed under arbitrary morphisms (so, in P systems terms, we can delete the unnecessary objects in a halting configuration).*

Now, let us define the *non-excluding inhibiting relation* and *saturated sets* with respect to this relation.

First, for a given alphabet V , let $R = \{r_1, r_2, \dots, r_k\}$ be a set of productions of the form $r_i : (A_i \rightarrow \alpha_i |_{\neg B_i})$, $A_i, B_i \in V$, $A_i \neq B_i$, $\alpha_i \in V^*$, $1 \leq i \leq k$.

For a rule $r : (A \rightarrow \alpha |_{\neg B}) \in R$ we define $left(r) = A$ and $inh(r) = B$.

Two rules $r_i, r_j \in R$ are said to be in the *non-excluding inhibiting relation*, and we denote this by $r_i \equiv_{nei} r_j$, iff $left(r_i) \neq inh(r_j)$ and $left(r_j) \neq inh(r_i)$.

Remark 3.2.2 *We have the following properties:*

- \equiv_{nei} is reflexive (obvious),
- \equiv_{nei} is symmetric (obvious),
- \equiv_{nei} is not transitive.

For example, take $R = \{r_1 : (A \rightarrow \alpha |_{\neg B}), r_2 : (D \rightarrow \beta |_{\neg C}), r_3 : (B \rightarrow \gamma |_{\neg A})\}$. Observe that $r_1 \equiv_{nei} r_2$, $r_2 \equiv_{nei} r_3$, but $r_1 \not\equiv_{nei} r_3$.

Definition 3.2.1 *A subset $W \subseteq R$ is said to be saturated (or complete) with respect to non-excluding inhibiting relation \equiv_{nei} iff $(\forall) r_i, r_j \in W$, $r_i \equiv_{nei} r_j$, and $(\forall) r_i \in R \setminus W$, $(\exists) r_j \in W$ such that $r_i \not\equiv_{nei} r_j$.*

Remark 3.2.3 *The set R may contain more saturated subsets, say W_1, W_2, \dots, W_k , with respect to the non-excluding inhibiting relation \equiv_{nei} . Recall that we will have $W_1 \cup W_2 \cup \dots \cup W_k = R$ and $W_i \cap W_j$, $1 \leq i, j \leq k$, not necessarily empty.*

Example 3.2.1 *Consider the following set of rules*

$$\begin{aligned} R = & \{r_1 : (A \rightarrow \alpha_1 |_{\neg B}), r_2 : (C \rightarrow \alpha_2 |_{\neg D}), r_3 : (B \rightarrow \alpha_3 |_{\neg D})\} \\ & \cup \{r_4 : (A \rightarrow \alpha_3 |_{\neg C}), r_5 : (D \rightarrow \alpha_4 |_{\neg C})\}. \end{aligned}$$

Then we have:

$$\begin{array}{llll} W_1 : & W_2 : & W_3 : & W_4 : \\ r_1 : (A \rightarrow \alpha_1 |_{\neg B}) & r_3 : (B \rightarrow \alpha_3 |_{\neg D}) & r_1 : (A \rightarrow \alpha_1 |_{\neg B}) & r_2 : (C \rightarrow \alpha_2 |_{\neg D}) \\ r_2 : (C \rightarrow \alpha_2 |_{\neg D}) & r_4 : (A \rightarrow \alpha_3 |_{\neg C}) & r_4 : (A \rightarrow \alpha_3 |_{\neg C}) & r_3 : (B \rightarrow \alpha_3 |_{\neg D}) \\ & & & r_5 : (D \rightarrow \alpha_4 |_{\neg C}) \end{array}$$

Lemma 3.2.1 *For any P system $\overline{\Pi}_1 = (\overline{V}, \overline{C}, \overline{P}, \overline{I}, \overline{\mu}, \overline{w}, \overline{R}, \overline{i}_0)$ of degree 1, with non-cooperative rules and inhibitors of weight 1 there exists an equivalent P system $\Pi_1 = (V, C, P, I, \mu, w, R, i_0)$ of degree 1, with non-cooperative rules and inhibitors of weight 1 such that:*

- $V = V_N \cup V_T, V_N \cap V_T = \emptyset;$
- $C = \bar{C} = P = \bar{P} = \emptyset;$
- $I \subseteq V;$
- $\mu = \bar{\mu} = []_1;$
- $w \subseteq V_N;$
- $R = \{r_1, \dots, r_k\}$ is a set of rules of the form $A \rightarrow \alpha|_{-B}$ or $A \rightarrow \alpha$, where $A, B \in V_N, \alpha \in V^*;$
- $i_0 = 1.$

Proof. (A sketch) We define the set $V_N = \{A \mid a \in \bar{V}\} \cup \{D\}$ and $V_T = \bar{V}$. The set of rules is defined as:

$$\begin{aligned}
R &= \{A \rightarrow \Phi D \mid a \rightarrow \phi \in R, A, D \in V_N, \Phi \in V_N^*\} \\
&\cup \{A \rightarrow \Phi D|_{-B} \mid a \rightarrow \phi|_{-b} \in R, A, B, D \in V_N, \Phi \in V_N^*\} \\
&\cup \{D \rightarrow \lambda\} \\
&\cup \{A \rightarrow a|_{-D} \mid A, D \in V_N, a \in V_T\},
\end{aligned}$$

where by Φ we understand the image of ϕ through a morphism that maps all letters from \bar{V} (denoted here by lower case symbols) to corresponding capital letters. Basically we detect when the system halts and only at that time we change the nonterminals into terminals. \square

Remark 3.2.4 *A similar result stands also for *P* systems with non-cooperative rules and promoters. Of course, both results can be extended to *P* systems having m membranes and not with only one membrane as we have considered.*

Lemma 3.2.2 *For any *P* system $\bar{\Pi}_1 = (\bar{V}, \bar{C}, \bar{P}, \bar{I}, \bar{\mu}, \bar{w}, \bar{R}, \bar{i}_0)$ with non-cooperative rules and inhibitors given in the form specified by Lemma 3.2.1 there exists an equivalent *P* system $\Pi_1 = (V, C, P, I, \mu, w, R, i_0)$ with non-cooperative rules and inhibitors such that the set of rules R is complete (i.e., for each $A \in V$ there exists at least one rule of type $A \rightarrow \alpha \in R$ or $A \rightarrow \alpha|_{-B} \in R$).*

Proof. (A sketch)

Assume that in \bar{R} there is no rule with object B on its left-hand side. (i.e., there are no rules of type $B \rightarrow \beta$ or $B \rightarrow \beta|_{-C}$). In addition, assume that object B is produced by rules of type $A \rightarrow \alpha B$ or $A \rightarrow \alpha B|_{-C}$. Also, suppose there is at least one rule of type $A \rightarrow \alpha|_{-B}$.

We construct the P system Π_1 , having the set of rules R complete, that simulates the moves of $\overline{\Pi}_1$ in the following manner. First, let

- $V = \overline{V} \cup \{D, b\}$;
- $C = \overline{C} = P = \overline{P} = \emptyset$;
- $I \subseteq V$.

The set of rules R is defined as:

- for any rule of type $A \rightarrow \alpha \in \overline{R}$ we add to R the rules:

$$A \rightarrow \alpha D,$$

$$D \rightarrow \lambda.$$

- for any rule of type $A \rightarrow \alpha|_{-C} \in \overline{R}$ we add to R the rules:

$$A \rightarrow \alpha D|_{-C},$$

$$D \rightarrow \lambda.$$

• for any rule of type $A \rightarrow \alpha B \in \overline{R}$ or $A \rightarrow \alpha B|_{-C} \in \overline{R}$ (therefore rules that produce at least one object B) we add to R the rules:

$$B \rightarrow B,$$

$$B \rightarrow b|_{-D}.$$

The explanation of the simulation is rather easy: at each computational step Π_1 produces the same multiset of objects as $\overline{\Pi}_1$ and, in addition, several copies of object D (according to the number of rules applied in $\overline{\Pi}_1$). The object D will be used in Π_1 to trigger a signal that the corresponding computation in $\overline{\Pi}_1$ halts (the absence of object D from the current multiset means that no rules from R , that correspond to rules in \overline{R} , can be applied anymore).

Now, suppose the object B is produced by a rule at a certain moment. Then, since in any non-halting configuration there exists object(s) D , the only rule handling object B that can be executed is $B \rightarrow B$. Finally, if the computation stops in $\overline{\Pi}_1$, then it means that in Π_1 the rules that can be further executed (in a non-deterministic manner) are $B \rightarrow B$ and $B \rightarrow b|_{-D}$. In case rule $B \rightarrow b|_{-D}$ is executed for all existing objects B , then the computation stops, Π_1 producing the same multiset of objects as $\overline{\Pi}_1$. \square

Remark 3.2.5 *A similar result stands also for P systems with non-cooperative rules and promoters. Of course, both results can be extended to P systems having m membranes and not with only one membrane as we have considered.*

Now we can state the following result which expresses the equality between the family of sets of vectors generated by *P* systems with non-cooperative inhibited rules and the class of Parikh images of languages generated by ET0L systems:

Theorem 3.2.2 $PsOP_m(ncoo, inhR_1) = PsET0L$.

Proof. We prove this result by double inclusion.

Case 1. $PsOP_m(ncoo, inhR_1) \supseteq PsET0L$.

For each $L \in ET0L$ there is a ET0L system $H = (V, T = \{T_1, T_2\}, \omega, \Delta)$ such that the terminals are only trivially rewritten. We construct a *P* system $\Pi = (V_\pi, C, P, I, \mu, w, R, i_0)$ that simulates the derivations of H in the following way.

- $V_\pi = V \cup \{t_1, t_2\} \cup \{S\} \cup \{\#\}$;
- $C = \bar{C} = P = \bar{P} = \emptyset$;
- $I \subseteq V_\pi$;
- $\mu = []_1$;
- $w = S\omega$;
- $R = \{S \rightarrow St_1, S \rightarrow St_2\}$
 $\cup \{S \rightarrow t_1t_2, t_1 \rightarrow \lambda, t_2 \rightarrow \lambda\}$
 $\cup \{a \rightarrow \alpha|_{-t_1} \mid a \rightarrow \alpha \in T_1\}$
 $\cup \{a \rightarrow \alpha|_{-t_2} \mid a \rightarrow \alpha \in T_2\}$
 $\cup \{A \rightarrow \#|_{-S} \mid A \in V \setminus \Delta\} \cup \{\# \rightarrow \#\}$;
- $i_0 = 1$.

Here is how the system works. The initial multiset consists of the string ω which corresponds to the ET0L axiom and a symbol S . The rules of types $S \rightarrow St_1$ (or $S \rightarrow St_2$) and $S \rightarrow t_1t_2$ represent the “core engine” that selects non-deterministically the ET0L table to be simulated; at any moment, in a non-deterministic manner, the rule $S \rightarrow t_1t_2$ can be applied.

A rule of type $S \rightarrow St_1$ (or $S \rightarrow St_2$) forbids the applications of all rules $a \rightarrow \alpha|_{-t_1}$ (or $a \rightarrow \alpha|_{-t_2}$, respectively), but allows the execution of rules corresponding to the ET0L table T_2 (or the execution of rules corresponding to the ET0L table T_1 , respectively). In this way, we correctly have simulated the application of the table t_2 (or t_1). Also, in the same computational step, either object t_1 or object t_2 is deleted from the current multiset by one of the rules $t_1 \rightarrow \lambda$ or $t_2 \rightarrow \lambda$, and the whole process can be iterated.

If the “core engine” stops (because the rule $S \rightarrow t_1 t_2$ was applied) then we may have two cases:

(i) the current multiset is formed only by symbols from Δ and then the whole computation will stop and the system generates exactly the same vector of numbers as the Parikh vector of a corresponding successful computation of the ET0L system.

(ii) the current multiset is over $V \setminus \Delta$ and then the computation will not stop since rules of type $A \rightarrow \#|_{-S}$ are executed and the rule $\# \rightarrow \#$ will cycle forever.

In conclusion we have shown that $PsOP_m(ncoo, inhR_1) \supseteq PsET0L$.

Case 2. $PsOP_m(ncoo, inhR_1) \subseteq PsET0L$.

Given a P system Π_m with $m \geq 1$ membranes we construct an equivalent P system $\Pi_1 = (V_\pi, C, P, I, \mu, w, R, i_0)$ (see Theorem 3.2.1). Without a loss of generality we consider the system Π_1 as being in the form given by Lemma 3.2.1 and Lemma 3.2.2, i.e., with a complete set of context-free rules over the disjoint sets of nonterminals and terminals.

We construct the ET0L system $H = (V, T = \{T_1, T_2, \dots, T_k\}, \omega, \Delta)$ that simulates Π_1 , as follows:

- $V = V_\pi \cup \{\#\}$;
- Let \bar{R} be the set of all rules of type $a \rightarrow \alpha|_{-b} \in R$, $b \neq \lambda$, from R . Also, let \bar{T}_i , $1 \leq i \leq k$, be all saturated subsets (with respect to non-excluding inhibiting relation \equiv_{nei}) of \bar{R} . Then, we define

$$\begin{aligned} T_i &= \{a \rightarrow \alpha \mid a \rightarrow \alpha|_{-b} \in \bar{T}_i\} \\ &\cup \{b \rightarrow \# \mid a \rightarrow \alpha|_{-b} \in \bar{T}_i\} \\ &\cup \{a \rightarrow \alpha \mid a \rightarrow \alpha \in R \setminus \bar{R} \text{ and } a \neq b (\forall) a \rightarrow \alpha|_{-b} \in \bar{T}_i\} \\ &\cup \{\# \rightarrow \#\}, \\ &1 \leq i \leq k; \end{aligned}$$

- $\omega = w$;
- $\Delta = V_T$, where $V_T \subset V$ is the set of terminals.

Here is how the ET0L system H simulates the computation of Π_1 . First remark that, from the way we defined the saturated subsets, the conditions on the rules can be omitted (observe that two rules $r_1 : (a_1 \rightarrow \alpha_1|_{-b_1})$ and $r_2 : (a_2 \rightarrow \alpha_2|_{-b_2})$ can simultaneously rewrite symbols a_1 and a_2 iff $b_1 \neq a_2$ and $a_1 \neq b_2$) in case we divide them in different tables. In addition, we have added to each table all context-free rules of the P system that does not violate the saturation relation considered for the table. We also add rules of type $b \rightarrow \#$ if rules $\{a \rightarrow \alpha \mid a \rightarrow \alpha|_{-b} \in \bar{T}_i\} \in T_i$; in this

way we assure that if we choose the “wrong” table, the computation will never stop since the $\#$ is produced (and therefore $\# \rightarrow \#$ will always be executed no matter which table is chosen).

In this way, if the computation stops, then the ET0L generates a language whose Parikh image is the same with the set of vectors of numbers generated by Π_1 , and hence by Π_m .

In conclusion, we have shown that $PsOP_m(ncoo, inhR_1) = PsET0L$. \square

Here we will prove that *P* systems with symbol objects and non-cooperative promoted rules can generate at least the same family of vectors sets as *PsET0L*. In what concerns the upper bound we show how these *P* systems can be simulated by random context grammars with limited checking.

We start by proving that the class of sets of vectors of numbers generated by *P* systems with non-cooperative promoted rules contains the class of Parikh images of languages generated by ET0L systems.

Theorem 3.2.3 $PsOP_1(ncoo, proR_1) \supseteq PsET0L$.

Proof. We will simulate the computation performed by an arbitrary ET0L system $H = (V, T = \{T_1, T_2\}, \omega, \Delta)$ using a *P* system $\Pi_1 = (V_\pi, C, P, I, \mu, w, R, i_0)$ defined as follows:

- $V_\pi = V \cup \{t, t_1, t_2, K, K_1\}$;
- $C = I = \emptyset$;
- $P \subseteq V$;
- $w = \omega t$ where ω is the axiom of system H ;
- $\mu = []_1$;
- $i_0 = 1$.

and the set of rules R_π is defined as:

$t \rightarrow t_1$,

$t \rightarrow t_2$,

$A \rightarrow \alpha K|_{t_1}$ for all rules $A \rightarrow \alpha \in T_1$,

$A \rightarrow \alpha K|_{t_2}$ for all rules $A \rightarrow \alpha \in T_2$,

$K \rightarrow K_1$,

$t_1 \rightarrow t|_A$ for all $A \in V \setminus \Delta$,

$$\begin{aligned}
t_2 &\rightarrow t|_A \text{ for all } A \in V \setminus \Delta, \\
t_1 &\rightarrow \lambda|_{K_1}, \\
t_1 &\rightarrow t|_{K_1}, \\
t_2 &\rightarrow \lambda|_{K_1}, \\
t_2 &\rightarrow t|_{K_1}, \\
K_1 &\rightarrow \lambda.
\end{aligned}$$

At the beginning of simulation we have inside the region of the P system the input multiset, consisting of string ω (which corresponds to the axiom of the ETOL system H), and object t (which represents the starting trigger for the simulation of the nondeterministic table selection mechanism). Nondeterministically, object t is transformed into t_1 or t_2 . Once object t_1 (or object t_2) is produced, the simulation of the corresponding ETOL table application starts. All rules $A \rightarrow \alpha K|_{t_1}$ (or $A \rightarrow \alpha K|_{t_2}$, respectively) corresponding to ETOL rules $A \rightarrow \alpha \in T_1$ (or $A \rightarrow \alpha \in T_2$, respectively) are applied in the maximally parallel manner. One can notice that if we applied at least once such a rule, we have produced at least one object K . In this moment we can distinguish two cases: 1) the current configuration is represented by a multiset that contains objects corresponding to ETOL nonterminals; 2) the current configuration is represented by a multiset that contains only objects corresponding to ETOL terminals.

In the first case one of the rules $t_1 \rightarrow t|_A$ or $t_2 \rightarrow t|_A$ will be executed, as well as the rule $K \rightarrow K_1$. Since an object t is produced, the simulation of applying a table in ETOL is iterated (recall that we do not have a terminal string, therefore we do not have to stop).

In the second case, rules $t_1 \rightarrow t|_A$ or $t_2 \rightarrow t|_A$ cannot be executed because we assumed that the current configuration is represented by a multiset that contains only objects corresponding to ETOL terminals. Therefore, rule $K \rightarrow K_1$ is executed and afterward one of the rules $t_1 \rightarrow \lambda|_{K_1}$ (or $t_2 \rightarrow \lambda|_{K_1}$, respectively) and $t_1 \rightarrow t|_{K_1}$. Depending on which rule is chosen we have again two cases – we stop the simulation having in the output region a terminal string or we continue. In both cases, as a last step of the iteration, rule $K_1 \rightarrow \lambda$ is applied.

The above construction proves that $PsOP_1(ncoo, proR_1) \supseteq PsETOL$. \square

Lemma 3.2.3 *For any P system Π with promoted non-cooperative rules there exists an equivalent P system Π' with promoted non-cooperative rules and with the same membrane structure like Π such that, for any halting configuration of Π' , all regions of Π' excepting the output one are empty.*

Proof. (A sketch) We will proceed as in the proof of Theorem 3.2.1 by first simulating a *P* system with promoted non-cooperative rules and m membranes $\overline{\Pi}_m$ with a *P* system with non-cooperative promoted rules and one membrane $\Pi_1 = (V, C, P, I, \mu, w, R, i_0)$. Then, using the same coding technique as in the mentioned proof, the problem is reduced to the ability to apply a morphism that deletes all non-necessary objects. Without entering into details we present here just the set of rules R' of a *P* system $\Pi'_1 = (V \cup \{E, E_1, E_2, D, D_1\}, C, P, I, \mu, w, R', i_0)$ (by $\Delta \subset V$ we denote the set of objects to be deleted in a halting configuration of Π_1):

$$\begin{aligned}
& A \rightarrow \overline{A}|_E \text{ for } A \in V, \\
& \overline{A} \rightarrow \overline{\alpha}D|_B \text{ for all rules } A \rightarrow \alpha|_B \in R, \\
& E \rightarrow E_1|_D, \\
& E_1 \rightarrow E, \\
& D \rightarrow D_1, \\
& \overline{A} \rightarrow A|_{E_1}, \\
& E \rightarrow E_2|_{D_1}, \\
& D_1 \rightarrow \lambda, \\
& A \rightarrow \lambda|_{E_2} \text{ for all symbols } A \in \Delta, \\
& E_2 \rightarrow \lambda.
\end{aligned}$$

Observe that the technique is to use object D as a “witness”, indicating that rules of type $A \rightarrow \alpha$ were applied. Fundamental for the construction is the ability of promoters (and of inhibitors as well) to react in the same time with the rules they promote (or inhibit, respectively). Also, the synchronization based on the external clock that regulates the computation is important because it allows us to check whether or not a rule was applied. \square

We continue by showing that the class of Parikh images of languages generated by random context grammars with limited checking includes the family of sets of vectors generated by *P* systems using non-cooperative promoted rules.

Theorem 3.2.4 $PsRC_{lc}^\lambda \supseteq PsOP_m(ncoo, proR_1)$.

Proof. Here we will prove that using random context grammars with limited checking we can simulate a *P* system with promoted non-cooperative rules and only one membrane. For the sake of simplicity and because of the arguments exposed in Lemma 3.2.3 we will take a system $\Pi_1 = (V, C, P_\pi, I, \mu, w, R, i_0)$ and we will construct a random context grammar with limited checking and λ rules $G = (N, T, P, S)$ that simulates the work of Π_1 .

Before we start, without loosing the generality, let us distinguish a terminal alphabet $\Delta \subseteq V$ and let us suppose that R contains only rules of type $A \rightarrow \alpha|_B$, $A, B \in V \setminus \Delta$, $\alpha \in V^*$. Let

$$\begin{aligned} N &= (V \setminus \Delta) \cup \{S, t, t_{ok}\} \cup \{\bar{A}, \bar{\bar{A}}, \tilde{A}\} \cup \{t_i \mid 1 \leq i \leq 4 * |V| + 1\} \\ &\cup \{t_i \mid 4 * |V| + 2 \leq i \leq 4 * |V| + 2 + k, k \text{ is the number of rules} \\ &\quad A \rightarrow \alpha|_B \in R \text{ such that there is no } A \rightarrow \beta \in R\}; \\ T &= \Delta. \end{aligned}$$

The set of rules P is constructed in the following way.

First, we add to the set P the rule

$(S \rightarrow wt, \emptyset, \emptyset)$, where w is the input string of Π_1 .

Its role is to set up a sentential form corresponding to the input multiset of Π_1 ; the symbol t will be used as a signal indicating that the simulation of the parallel applications of rules in Π_1 is about to start. As in Theorem 2.2.1, the trick is to “paint” all objects respecting the conditions imposed for the rules in the P system definition, and then nondeterministically check whether or not all rules that correspond to the rules of the P system were actually applied. In this way we are able to simulate the maximal parallelism feature of the P systems. More formally, we add to the set P the rules:

$$\begin{aligned} &(A \rightarrow \bar{A}, \{t\}, \emptyset) \text{ for } A \in V, \\ &(t_i \rightarrow t_{i+1}, \emptyset, \{A_i\}), \text{ for } A_i \in V, 1 \leq i \leq |V|, \\ &(\bar{A} \rightarrow \bar{\alpha}\tilde{A}, \{B\}, \emptyset) \text{ for all rules } A \rightarrow \alpha|_B, \\ &(\bar{A} \rightarrow \bar{\alpha}\tilde{A}, \{\tilde{B}\}, \emptyset) \text{ for all rules } A \rightarrow \alpha|_B, \\ &(t_{|V|+i} \rightarrow t_{|V|+i+1}, \emptyset, \{\bar{A}_i\}), \text{ for } A_i \in V, 1 \leq i \leq |V|, \\ &(\tilde{A} \rightarrow \lambda, \{t_{2*|V|+1}\}, \emptyset), \\ &(t_{2*|V|+i} \rightarrow t_{2*|V|+i+1}, \emptyset, \{\tilde{A}_i\}), \text{ for } A_i \in V, 1 \leq i \leq |V|, \\ &(\bar{\bar{A}}_i \rightarrow A_i, \{t_{3*|V|+1}\}, \emptyset) \text{ for } A_i \in V, 1 \leq i \leq |V|, \\ &(t_{3*|V|+i} \rightarrow t_{3*|V|+i+1}, \emptyset, \{\bar{\bar{A}}_i\}), \text{ for } \bar{\bar{A}}_i, 1 \leq i \leq |V|. \end{aligned}$$

Now, if symbol $t_{4*|V|+1}$ is obtained, then we know that the parallelism of the P system was correctly simulated. However, we do not know whether or not the corresponding configuration of the P system admits a new transition (recall that in the P system we have objects, therefore there is no difference between terminals

and nonterminals). So, in our grammar we have to be sure when the simulation core stops (i.e., symbol $t_{4*|V|+1}$ appears), and only afterward we have to transform (in case it was a correct simulation) the nonterminals into terminals. Therefore, nondeterministically we have to check if we can apply at least one rule of type $A \rightarrow \alpha|_B$. This task can be achieved by sequences of “linked” rules that check the presence of objects A and B :

$$(t_{4*|V|+1} \rightarrow t_{4*|V|+2}, \{A\}, \emptyset),$$

$$(t_{4*|V|+2} \rightarrow t_{ok}, \{B\}, \emptyset).$$

Next, if object t_{ok} appears in the sentential form, then it means that conditions for applying other rules are fulfilled and so we can restart the whole process or we can nondeterministically stop. We have also the rules:

$$(t_{ok} \rightarrow t, \emptyset, \emptyset),$$

$$(t_{ok} \rightarrow \lambda, \emptyset, \emptyset).$$

In this way we have shown that random context grammars with limited checking are able to simulate *P* systems with promoted non-cooperative rules. \square

3.3 Catalytic *P* Systems with Promoted/Inhibited Rules

Here we will investigate the computational power of *P* systems with non-cooperative and catalytic promoted/inhibited rules; as we will see later on, one catalyst is enough to obtain universality. The purpose of the catalyst will be to block the parallelism while the promoters/inhibitors guide the computation in the desired manner. In this way we can achieve even deterministic computations.

3.3.1 Computational Universality – The Generative Case

Here, we present two universality results concerning *P* systems with promoters or inhibitors at the level of rules. The proofs are based on the simulations of regularly controlled context-free grammars with appearance checking.

As we presented in Section 2.2, the family of languages generated by such grammars, λrC_{ac} , is equal to the family of all recursively enumerable languages, RE . As a particular case, by NRE we denote the family of Turing computable sets of numbers. This family is equal to the family of length sets of languages generated

by regularly controlled context-free grammars with appearance checking over one letter terminal alphabet.

Recall also, that, because P systems with symbol-objects operate with multisets of objects (therefore we do not have the order given by strings) we at most can study the equivalence with a family of vectors of natural numbers.

The following result shows the equality between the class of sets of vectors generated by P systems using one catalyst and promoters at the level of rules, and the family of Parikh images of all recursively enumerable languages.

Theorem 3.3.1 $PsOP_m(cat_1, proR_1) = PsRE$, $m \geq 2$.

Proof. We will consider for this proof the implication $PsRE \subseteq PsOP_2(cat, proR_1)$; the opposite inclusion is assumed true by invoking the Church-Turing thesis.

Let $G_{reg} = (N_{reg}, T_{reg}, P_{reg}, S_{reg})$ be a regular grammar generating the regular set L_{reg} . We denote by r the number of rules in P_{reg} . The rules of P_{reg} are enumerated as $i : (M_i \rightarrow p_i Q_i)$ or $i : (M_i \rightarrow p_i)$ with $1 \leq i \leq r$, where $M_i \in N_{reg}$ and $p_i \in T_{reg}$, $1 \leq i \leq r$. For any such grammar G_{reg} we can construct an equivalent right-linear grammar $G' = (N', T', P', S')$ in the following way:

$$\begin{aligned} T' &= T_{reg}, \\ S' &= S_{reg}, \\ N' &= N_{reg} \cup \{M_{(i,1)}, M_{(i,2)}, M_{(i,3)} \mid 1 \leq i \leq r\}. \end{aligned}$$

For any rule $i : (M_i \rightarrow p_i Q_i) \in P_{reg}$ or $i : (M_i \rightarrow p_i) \in P_{reg}$, $1 \leq i \leq r$, we will have in P' the sequence of rules:

$$\begin{aligned} M_i &\rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i Q_i, \text{ and} \\ M_i &\rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i, \end{aligned}$$

respectively. Moreover, P' does not contain other rules excepting the rules considered above.

In other words, the only difference between the two grammars is that the production of a new terminal in grammar G' is done after each fourth step of a derivation.

Now let us construct a P system which simulates the derivation process of a regularly controlled grammar with appearance checking. The system will use only two membranes, one catalyst, and promoters. The innermost membrane will contain the generative mechanism and the results of computation will be send out to the skin membrane which will be the output membrane of the system (the reason is that the catalyst is used during the computation to inhibit the parallelism and it cannot

be removed, therefore we cannot obtain the number 0 as the result of computation if we use only one membrane). In what follows we will discuss only the rules in the innermost membrane since the skin membrane does not execute any task (its role is only to collect the objects obtained during computation).

The promoters will be generated by a mechanism like the one presented above (promoters will be actually terminal symbols from T' and, therefore, they will be generated at each forth step). They will permit the execution of “context-free” rules in the “right” order – the order given by the regular mechanism.

In order to correctly simulate the appearance checking mechanism we have to modify the rules in the grammar G' such that we replace each rule of type $M_{(i,3)} \rightarrow p_i Q_i$ by rules of type: $M_{(i,3)} \rightarrow p_i Q_i f$ or $M_{(i,3)} \rightarrow p_i Q_i a$ depending on how the object p_i indicates a rule from F (in the regularly controlled grammar definition, the set $F \subseteq P$ represents the appearance checking set of rules; we will use the object a to identify that a rule with the corresponding label p_i is in the appearance checking set; if not, we will produce in the rule the object f). We will consider also the same construction for the rules in G' of type $M_{(i,3)} \rightarrow p_i$, i.e., $M_{(i,3)} \rightarrow p_i f$ or $M_{(i,3)} \rightarrow p_i a$. This means that, in the definition of our *P* system, for the inner membrane we will have rules of the following types:

- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i Q_i f$ if p_i is not a label in the appearance checking set;
- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i Q_i a$ if p_i is a label in the appearance checking set;
- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i f$ if p_i is not a label in the appearance checking set;
- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i a$ if p_i is a label in the appearance checking set.

Up to this moment, we only have considered the regular mechanism which generates labels indicating the context-free rules that should be applied. Let us denote by $G_{CF} = (N_{CF}, T_{CF}, P_{CF}, S_{CF})$ a context-free grammar with productions labeled with the elements of T_{reg} . Now we will discuss how we can simulate (by using *P* systems means) the application of a context-free rule $p : A \rightarrow \alpha$ indicated by the regular mechanism.

For a context-free rule $(p : A \rightarrow \alpha) \in G_{CF}$ we will have in our *P* system the following sequence of rules:

$$cA \rightarrow cD\alpha|_p,$$

$$p \rightarrow p',$$

$$p' \rightarrow \lambda|_D,$$

$$D \rightarrow \lambda.$$

Here, we have considered, without loosing the generality, that $\alpha \in (N \cup T_{out})^*$ meaning that if we apply the rule $p : A \rightarrow \alpha$ we will send to the output region the terminal symbols (recall that we are interested only in the number of objects).

If promoter p , object A , and catalyst c are present at a certain moment together, then they will react only once in two consecutive computational steps. This is due to the fact that the promoter p is changed ($p \rightarrow p'$) in the same moment with the execution of the rule $cA \rightarrow cD\alpha|_p$. Moreover, the presence of the catalyst c in the rule inhibits the parallelism (we want that in one “round” the rule $A \rightarrow \alpha$ to be applied only once and not for all occurrences of object A that may exist in the region). Now, in order to be sure that the rule $cA \rightarrow cD\alpha|_p$ was executed, an object D is created; it will help to delete the object p' present in membrane (which if not deleted can cause problems in further steps). The object D will be also deleted by the rule $D \rightarrow \lambda$.

This sequence of rules stands for the case when the context-free rule $A \rightarrow \alpha$ can be applied and so, it must be applied. In the case when the rule mentioned cannot be applied we have to decide if the promoter present indicates a rule with the label in the appearance checking set or not.

First, let us consider the case where the promoter is not a label in the appearance checking set. In this case, recall that we deal with the following sequences of productions (from the regular mechanism):

- $M_i \rightarrow M_{(i,1)}; M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i Q_i f,$ or
- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i f.$

As the result of applying these rules we will have in the inner region, among others, the objects p and f . Let us consider that, for this case, we have the rules:

$$f \rightarrow f_1,$$

$$f_1 \rightarrow f_2,$$

$$p' \rightarrow \#|_{f_2},$$

$$f_2 \rightarrow \lambda,$$

$$\# \rightarrow \#',$$

$$\#' \rightarrow \#.$$

The first two rules from this group are meant to delay the execution of the third rule because we are not “sure” if the rule $cA \rightarrow cD\alpha|_p$ is or it is not applied. So, if the mentioned rule is applied, then the object f_2 will be deleted by the rule $f_2 \rightarrow \lambda$ and will not promote the rule $p' \rightarrow \#|_{f_2}$ since the object $p' \rightarrow \lambda|_D$ was “consumed”

in a previous step. As a consequence, there will be no effect (in terms of objects produced) if the previous set of rules is executed. In the opposite case (when the rule $cA \rightarrow cD\alpha|_p$ is not applied because there is no object A present in the region), then the object $\#$ will be generated. The rules $\# \rightarrow \#'$, $\#' \rightarrow \#$ will cycle forever and the computation will never halt and this will mean that the computation have failed (recall that we show the universality for the nondeterministic case).

With a similar construction like above we can solve the case when we deal with rules that have labels in the appearance checking set. This means that the rules to be applied are of the types:

- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i Q_i a,$ or
- $M_i \rightarrow M_{(i,1)}, M_{(i,1)} \rightarrow M_{(i,2)}, M_{(i,2)} \rightarrow M_{(i,3)}, M_{(i,3)} \rightarrow p_i a.$

Here the difference from the previous case consists in not generating the trap symbol $\#$ if the rule $cA \rightarrow cD\alpha|_p$ cannot be applied. We only have to delete the promoter p' . Its presence may interfere in the next steps of computation if it is not deleted. The rules below state the fact that if a rule is in the appearance checking set and it cannot be applied even if it is indicated by the regular mechanism, then it can be skipped.

$$\begin{aligned} a &\rightarrow a_1, \\ a_1 &\rightarrow a_2, \\ p' &\rightarrow \lambda|_{a_2}, \\ a_2 &\rightarrow \lambda. \end{aligned}$$

Finally, the initial configuration of the *P* system is composed by the starting symbol S_{Reg} of the regulating mechanism, the starting symbol S_{CF} of the context-free mechanism and the catalyst c . The system will evolve in a maximally parallel manner its behavior being controlled by the catalyst and promoters.

One can notice that we arrive at the same descriptive complexity in terms of number of membranes, number of catalysts and promoters as in the original proof in [16], but simulating a different computational universal mechanism. \square

As a particular case we have the following result:

Corollary 3.3.1 $NOP_2(cat_1, proR_1) = NRE$.

As it can be seen, the promoters combined with one catalyst are sufficient to prove the computational universal capabilities of the *P* systems when using only context-free object rewriting rules. Also a similar result, but concerning inhibitors instead of promoters, stands.

Theorem 3.3.2 $PsOP_m(cat_1, inhR_1) = PsRE$, $m \geq 2$.

Proof. Let us consider a similar construction like the one in the previous proof, but considering a sequence of rules as the one given below. Here we also will have two membranes and will discuss only the rules presented in the inner membrane, the skin membrane being used only to collect the result of computations. First let us recall that the rules from the context-free grammar are labeled with the symbols p_1, \dots, p_k . Consider also that if in the regular grammar which controls the derivation of the context-free grammar we have a rule $M_i \rightarrow p_j Q_i$, then in our simulation we will have:

$$\begin{aligned} M_i &\rightarrow p_1 \dots p_k M_{(i,1)}, \\ M_{(i,1)} &\rightarrow p_1 \dots p_k M_{(i,2)}, \\ M_{(i,2)} &\rightarrow p_1 \dots p_k M_{(i,3)}, \\ M_{(i,3)} &\rightarrow p_1 \dots p_{j-1} p_{j+1} \dots p_k Q_i fr. \end{aligned}$$

Moreover, in the above construction we have considered that the label p_j indicates a context-free rule which is not in the appearance checking set. As it can be seen, instead of indicating the rule with the symbol p_j (meaning that the context-free rule with label p_j should be applied), we have used all the symbols from the complementary set, i.e., $p_1, \dots, p_{j-1}, p_{j+1}, \dots, p_k$.

Now, let us consider the following set of rules which simulates the behavior of the context-free mechanism:

$$\begin{aligned} cA &\rightarrow CD\alpha|_{-p_j}, \\ p_j &\rightarrow \lambda, \\ f &\rightarrow f_1, \\ r &\rightarrow r_1, \\ D &\rightarrow D_1, \\ r_1 &\rightarrow r_2|_{-D}, \\ f_1 &\rightarrow f_2, \\ f_2 &\rightarrow \#|_{-D_1}, \\ f_2 &\rightarrow \lambda|_{-r_2}, \\ r_2 &\rightarrow \lambda, \\ D_1 &\rightarrow \lambda, \\ \# &\rightarrow \#', \\ \#' &\rightarrow \#. \end{aligned}$$

The first three rules of the “regular controller” produce the symbols p_1, \dots, p_k and so the rule $cA \rightarrow CD\alpha|_{-p_j}$ cannot be applied because always object p_j will be

among the mentioned objects. This means that during the first three steps in a cycle the rule $cA \rightarrow CD\alpha|_{\neg p_j}$ cannot be applied. Also, at each step we delete the objects p_1, \dots, p_k by rules of type $p_j \rightarrow \lambda$. In the last step in a cycle, we avoid to introduce the object p_j and so the rule $cA \rightarrow CD\alpha|_{\neg p_j}$ can be applied. With this occasion, we also introduce the objects f and r which are used to indicate a rule which is not in the appearance checking set.

The rule $cA \rightarrow CD\alpha|_{\neg p_j}$ will modify only one (if any) occurrence of object A present in the region because of the catalyst c . Moreover in the next step of computation the objects p_1, \dots, p_k will be again introduced, so they will forbid the execution of any rule of type $cA \rightarrow CD\alpha|_{\neg p_j}$.

Now, coming back, we can notice that the rule $cA \rightarrow CD\alpha|_{\neg p_j}$ can be applied only if there is an occurrence of object A . Recall that this rule is not in the appearance checking set and so, if it cannot be applied, then the computations must cycle forever in order not to accept. If the rule can be applied, it has to be applied and moreover all the symbols that were produced during the computation must be deleted in order not to interfere in the next cycle.

Now let us see which is the result of a computation in both cases. Consider the first case, when the rule $cA \rightarrow CD\alpha|_{\neg p_j}$ is applied. Then, in the same moment, the rules $f \rightarrow f_1$ and $r \rightarrow r_1$ are executed. In the next step the rules $D \rightarrow D_1$ and $f_1 \rightarrow f_2$ can be applied. The rule $r_1 \rightarrow r_2|_{\neg D}$ cannot be applied since the object D is present in the region. As an effect, the region will contain the objects D_1 , f_2 , and r_1 . So, the only applicable rules are: $D_1 \rightarrow \lambda$, $f_2 \rightarrow \lambda|_{\neg r_2}$, and $r_1 \rightarrow r_2|_{\neg D}$. Finally, the rule $r_2 \rightarrow \lambda$ deletes the last symbol that was created in this cycle. This means that we reestablish the initial configuration and the computation can continue.

Now, consider the opposite case, when the rule $cA \rightarrow CD\alpha|_{\neg p_j}$ has to be applied (because is indicated by the regular control), but it cannot be applied (because, there is no symbol A present in the region). So, the symbol D is not released. As an effect we will have in the region the objects f_1 and r_1 and the rules to be applied are $r_1 \rightarrow r_2|_{\neg D}$ and $f_1 \rightarrow f_2$. Next, the rules $f_2 \rightarrow \#|_{\neg D_1}$ and $r_2 \rightarrow \lambda$ will be applied and so the $\#$ symbol will be created and the computation will never halt.

For the rules that are in the appearance checking set we can consider for the regular mechanism rules of the following types:

$$\begin{aligned} M_i &\rightarrow p_1 \dots p_k M_{(i,1)}, \\ M_{(i,1)} &\rightarrow p_1 \dots p_k M_{(i,2)}, \\ M_{(i,2)} &\rightarrow p_1 \dots p_k M_{(i,3)}, \\ M_{(i,3)} &\rightarrow p_1 \dots p_{j-1} p_{j+1} \dots p_k Q_i, \end{aligned}$$

while for the “context-free” mechanism a rule

$cA \rightarrow C\alpha|_{\neg p_j}$ is considered.

In this way, if the rule can be applied, then it will be applied and if cannot be applied, then nothing will happen and the process will continue correctly simulating the appearance checking mechanism.

Another aspect one can notice is that we can generate the family of recursively enumerable sets of natural non-null numbers if we use only one membrane since we use the catalyst c which remains inside the region and it cannot be removed. \square

As a particular case we have the following result:

Corollary 3.3.2 $NOP_2(cat_1, inhR_1) = NRE$.

3.3.2 Computational Universality – The Accepting Case

The following theorems illustrate the computational universality (in their accepting variants) of P systems with object rewriting non-cooperative rules and promoters/inhibitors at the level of rules. The systems we propose simulate the moves of deterministic register machines. Moreover, the obtained P systems are also deterministic.

Theorem 3.3.3 $DPsIOP_2(cat_1, proR_1) = PsRE$.

Proof. The inclusion $DPsIOP_2(cat_1, proR_1) \subseteq PsRE$ is assumed true by invoking the Turing-Church thesis.

In order to prove the reverse inclusion we will simulate an n -register machine $M = (n, \mathcal{P}, l_0, l_h)$. At each time during the computation, the current contents of register $1 \leq i \leq n$ is represented by the multiplicity of the object a_i .

Formally, we define the P system $\Pi = (V, C, P, I, \mu, w_1, w_2, R_1, R_2, i_0)$, where:

$$\begin{aligned} V &= \{a_i, A_i, S_i \mid 1 \leq i \leq n\} \cup \{F, T\} \\ &\cup \{l_1, l'_1 \mid (l_1 : \text{ADD}(i), l_2) \in \mathcal{P}\} \\ &\cup \{l_1, l'_1, l''_1 \mid (l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}\}; \\ C &= \{c\}; \\ P &\subseteq V; \\ I &= \emptyset; \\ \mu &= [[]_2]_1; \end{aligned}$$

$$\begin{aligned}
w_1 &= \emptyset; \\
w_2 &= cl_0 a_1^{k_1} \dots a_n^{k_n}; \\
R_1 &= \emptyset; \\
i_0 &= 1;
\end{aligned}$$

and R_2 is defined as follows:

- for each instruction $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$, we add to R_2 the rules:

$$\begin{aligned}
l_1 &\rightarrow l'_1 A_i, \\
c &\rightarrow ca_i|_{A_i}, \\
A_i &\rightarrow \lambda, \\
l'_1 &\rightarrow l_2;
\end{aligned}$$
- for each instruction $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$, we add to R_2 the rules:

$$\begin{aligned}
l_1 &\rightarrow l'_1 T S_i, \\
ca_i &\rightarrow cF|_{S_i}, \\
S_i &\rightarrow \lambda, \\
l'_1 &\rightarrow l''_1, \\
T &\rightarrow T', \\
l''_1 &\rightarrow l_2|_F, \\
F &\rightarrow \lambda, \\
T' &\rightarrow T'', \\
l''_1 &\rightarrow l_3|_{T''}, \\
T'' &\rightarrow \lambda;
\end{aligned}$$
- for the instruction $(l_h : \text{HALT}) \in \mathcal{P}$, we add to R_2 the rules:

$$\begin{aligned}
a_i &\rightarrow \#|_{l_h}, 1 \leq i \leq n, \\
l_h &\rightarrow \lambda;
\end{aligned}$$
- the rule $\# \rightarrow \#$ is added to R_2 .

The system works as follows. Initially the P system Π starts the computation having in its input region (region 2) the objects $a_1^{k_1}, \dots, a_n^{k_n}$, the catalyst c and the label l_0 corresponding to the first instruction of the register machine we want to simulate. The vector (k_1, \dots, k_n) represents the vector that has to be accepted by our P system.

The P system starts the computation by simulating the first instruction of the register machine program. Let us suppose that the current instruction to be executed

is of type $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$. Then, in region 2 the corresponding rule $l_1 \rightarrow l'_1 A_i$ is executed. The object A_i indicates that the number of objects a_i has to be incremented. This will be realized, in the second computational step, by the promoted catalytic rule $c \rightarrow ca_i|_{A_i}$; the rules $A_i \rightarrow \lambda$ and $l'_1 \rightarrow l_2$ are executed in the same time with the previous one. The first rule, assures that $c \rightarrow ca_i|_{A_i}$ is not executed again in one iteration (therefore, i is just incremented), while the second rule allows the P system to further simulate the instruction of the register machine indicated by label l_2 .

If subtraction instruction $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$ is simulated, in region 2 the rule $l_1 \rightarrow l'_1 T S_i$ is executed (note that the object S_i stands for the subtraction command). As a result, the rule $ca_i \rightarrow cF|_{S_i}$ is executed and the number of objects a_i is decreased by 1 (if it is possible, i.e. the number of objects a_i is greater than 0). Meanwhile, the execution of rule $S_i \rightarrow \lambda$ guarantees that only one object a_i was deleted from the current multiset (again, if it is possible). If this is the case, then the sequence of rules $l'_1 \rightarrow l''_1, l''_1 \rightarrow l_2|_F, F \rightarrow \lambda$ was executed, and the label of the next register machine instruction was generated. Otherwise (i.e., there is no objects a_i in region 2) the rule $ca_i \rightarrow cF|_{S_i}$ is not executed therefore the object F is not produced and the rule $l''_1 \rightarrow l_2|_F$ cannot be applied. Meanwhile, the object T evolves to T' and then to T'' , in three consecutive steps. The last step of an iteration consists of simultaneous execution of the rules $l''_1 \rightarrow l_3|_{T''}$ and $T'' \rightarrow \lambda$. In conclusion, the label of the next register machine instruction is created.

The above presented simulations of the instructions, $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$ and $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$, are iterated according to the register machine program \mathcal{P} . The simulation stops when the object l_h (which corresponds to the label l_h of the register machine instruction $(l_h : \text{HALT})$) is generated and no other objects a_j , $1 \leq j \leq n$, are in region 2. In all other cases, the trap symbol $\#$ is generated and the computation will cycle forever.

However, all the rules involving $\#$ symbol can be removed from the presented P system definition since, as we mention in Section 2.3, a register machine can accept recursively enumerable sets of vectors of numbers even if it stops without having all registers empty.

In conclusion, we have shown that $DPsIOP_2(\text{cat}_1, \text{proR}_1) \supseteq PsRE$. Consequently, we have proved that $DPsIOP_2(\text{cat}_1, \text{proR}_1) = PsRE$. \square

Theorem 3.3.4 $DPsIOP_2(cat_1, inhR_1) = PsRE$.

Proof. The inclusion $DPsIOP_2(cat_1, inhR_1) \subseteq PsRE$ is assumed true by invoking the Turing-Church thesis.

The inclusion $DPsIOP_2(cat_1, inhR_1) \supseteq PsRE$ will be proved by simulating an n -register machine $M = (n, \mathcal{P}, l_0, l_h)$. The contents of register i will be represented in our simulation, as in the previous theorem, by the multiplicity of the object a_i .

Formally, we define the *P* system $\Pi = (V, C, P, I, \mu, w_1, w_2, R_1, R_2, i_0)$, where:

$$\begin{aligned}
V &= \{a_i, A_i, S_i \mid 1 \leq i \leq n\} \cup \{K, K_1, Q, Q_1, Q_2\} \\
&\cup \{l_1, l'_1 \mid (l_1 : \text{ADD}(i), l_2) \in \mathcal{P}\} \cup \{l_1 \mid (l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}\} \\
&\cup \{F_{(l_1,0)}, F_{(l_1,1)}, F_{(l_1,2)} \mid (l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}\}; \\
C &= \{c\}; \\
P &= \emptyset; \\
I &\subset V; \\
\mu &= [[]_2]_1; \\
w_1 &= \emptyset; \\
w_2 &= cl_0 a_1^{k_1} \dots a_n^{k_n}; \\
R_1 &= \emptyset; \\
i_0 &= 1;
\end{aligned}$$

and the set R_2 is defined as follows:

- for each instruction $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$, we add to R_2 the rules:
$$\begin{aligned}
l_1 &\rightarrow l'_1 A_1 \dots A_{i-1} A_{i+1} \dots A_n S_1 \dots S_n, \\
c &\rightarrow ca_i A_1 \dots A_n S_1 \dots S_n |_{\neg A_i}, \\
A_i &\rightarrow \lambda, \quad 1 \leq i \leq n, \\
S_i &\rightarrow \lambda, \quad 1 \leq i \leq n, \\
l'_1 &\rightarrow l_2 A_1 \dots A_n S_1 \dots S_n;
\end{aligned}$$
- for each instruction $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$, we add to the set R_2 the rules:
$$\begin{aligned}
l_1 &\rightarrow F_{(l_1,0)} Q A_1 \dots A_n S_1 \dots S_{i-1} S_{i+1} \dots S_n, \\
ca_i &\rightarrow cK A_1 \dots A_n S_1 \dots S_n |_{\neg S_i}, \\
A_i &\rightarrow \lambda, \quad 1 \leq i \leq n, \\
S_i &\rightarrow \lambda, \quad 1 \leq i \leq n, \\
F_{(l_1,0)} &\rightarrow F_{(l_1,1)} A_1 \dots A_n S_1 \dots S_n, \\
Q &\rightarrow Q_1 A_1 \dots A_n S_1 \dots S_n,
\end{aligned}$$

$$\begin{aligned}
K &\rightarrow K_1 A_1 \dots A_n S_1 \dots S_n, \\
Q_1 &\rightarrow Q_2 A_1 \dots A_n S_1 \dots S_n |_{-K}, \\
F_{(l_1,1)} &\rightarrow F_{(l_1,2)} A_1 \dots A_n S_1 \dots S_n, \\
F_{(l_1,2)} &\rightarrow l_3 A_1 \dots A_n S_1 \dots S_n |_{-K_1}, \\
K_1 &\rightarrow \lambda, \\
F_{(l_1,2)} &\rightarrow l_2 A_1 \dots A_n S_1 \dots S_n |_{-Q_2}, \\
Q_2 &\rightarrow \lambda.
\end{aligned}$$

As in the the proof of the previous theorem we start the computation having in region 2 the objects $a_1^{k_1}, \dots, a_n^{k_n}$, the catalyst c , and the label l_0 of the first instruction of the register machine we want to simulate.

Let us consider that an increment instruction $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$ has to be simulated. Then, the existing object l_1 (which represents the current instruction label) is rewritten by the rule $l_1 \rightarrow l'_1 A_1 \dots A_{i-1} A_{i+1} \dots A_n S_1 \dots S_n$. As it can be seen only the object A_i is missing from the right hand side of the rule. The absence of this object indicates that the number of objects a_i has to be incremented. Because of this, the rule $c \rightarrow ca_i |_{-A_i}$ can be applied. Meanwhile, the rules $A_i \rightarrow \lambda$, $1 \leq i \leq n$, $S_i \rightarrow \lambda$, $1 \leq i \leq n$, and $l'_1 \rightarrow l_2 A_1 \dots A_n S_1 \dots S_n$ are executed. Their role is to reconfigure the system for the next instruction that has to be simulated.

Now, let us consider that the system receives the command to subtract one object a_i from the current multiset (the rule $l_1 \rightarrow F_{(l_1,0)} Q A_1 \dots A_n S_1 \dots S_{i-1} S_{i+1} \dots S_n$ is executed).

If $|w|_{a_i} \geq 1$, where w is the current multiset present in region 2, then we can apply the rule $ca_i \rightarrow cK A_1 \dots A_n S_1 \dots S_n |_{-S_i}$ since the object S_i is missing. The rules $A_i \rightarrow \lambda$, $1 \leq i \leq n$, and $S_i \rightarrow \lambda$, $1 \leq i \leq n$, are executed at each step of computation. The objects $A_1, \dots, A_n, S_1, \dots, S_n$ are created always with only one exception – when we want to execute the instruction corresponding to the missing object. The computation continues until the rule $F_{(l_1,2)} \rightarrow l_2 A_1 \dots A_n S_1 \dots S_n |_{-Q_2}$ is executed and the object l_2 indicating the next instruction label is generated.

In a similar way as shown in the previous theorem proof, if $|w|_{a_i} = 0$, then the rule $ca_i \rightarrow cK A_1 \dots A_n S_1 \dots S_n |_{-S_i}$ is not executed. Notice that objects $F_{(l_1,0)}$ and Q are “witnesses” that the command to simulate the subtraction scheme was made. Similarly, the presence of object K indicates that, in region 2, was at least one object a_i . In order to correctly simulate the decrement instruction, we want to have the “right” missing object at the “right” time. This involves some delaying rules like $F_{(l_1,0)} \rightarrow F_{(l_1,1)} A_1 \dots A_n S_1 \dots S_n$, $Q \rightarrow Q_1 A_1 \dots A_n S_1 \dots S_n$, $K \rightarrow K_1 A_1 \dots A_n S_1 \dots S_n$, $F_{(l_1,1)} \rightarrow F_{(l_1,2)} A_1 \dots A_n S_1 \dots S_n$.

If everything worked well, then the rule $F_{(l_1,2)} \rightarrow l_3 A_1 \dots A_n S_1 \dots S_n |_{-K_1}$ was executed and the symbol l_3 indicating the next instruction was generated.

Finally, if the symbol l_h (that corresponds to the register machine (l_h : HALT) instruction) is generated, then computation stops and the system accepts the input. In conclusion, we have shown that $DPsIOP_2(cat_1, inhR_1) \supseteq PsRE$. Consequently, we have proved that $DPsIOP_2(cat_1, inhR_1) = PsRE$. \square

3.4 Ultimately Confluent Universality

The following theorem shows the computational universality of *P* systems with object rewriting context-free rules and promoters of weight two. The system we propose simulates again the moves of a register machine.

Even if the simulated machine is deterministic, because in our system we do not prevent a way to control the nondeterminism, the method used is to reestablish a previous configuration if the computation went in the “wrong way”.

The system may not stop even if there is a halting computation and the total number of reachable configurations is finite. This is due to the nondeterminism and this is the price paid to avoid the use of cooperative (or catalytic) rules which may inhibit the parallelism of the system. However, considering a fair computation (with equal probabilities when selecting the branches in non-deterministic computations), an endless simulation of a finite computation has probability zero. In this way, the notion of algorithm (in the framework of total functions) makes sense because the solution is ultimately confluent and the system will stop with probability 1 if the simulated register machine stops.

Theorem 3.4.1 $UPsIOP_1(ncoo, proR_2) = PsRE$.

Proof. Consider an n -register machine $M = (n, \mathcal{P}, l_0, l_h)$. The contents of register i is denoted in our simulation by the multiplicity of the object a_i .

We define the *P* system

$$\Pi = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where

$$\begin{aligned} V &= \{a_i, x_i, y_i \mid 1 \leq i \leq n\} \cup \{h, t_0, t_1, t_2, t_3, t_4\} \\ &\cup \{l_1 \mid (l_1 : \text{ADD}(i), l_2) \in \mathcal{P}\} \end{aligned}$$

$$\begin{aligned}
& \cup \{l_1, l_I, l_{II}, l_{III}, l_{IV}, l_V \mid (l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}\}; \\
C &= I = \emptyset; \\
P &\subseteq V; \\
\mu &= []_1; \\
w &= l_0 a_1^{k_1} \dots a_n^{k_n}; \\
i_0 &= 1;
\end{aligned}$$

and R_1 is defined as follows:

For each $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$, the set R_1 contains the rule $l_1 \rightarrow a_i l_2$.

For each $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$, the set R_1 contains the rules given in the table below (for clarity, the rules are structured according to the order of application and different cases that may occur).

Step & Case	Rules
1ABCD	$l_1 \rightarrow l_I t_0$
2ABCD	$t_0 \rightarrow t_1$
2BCD	$l_I \rightarrow l_{II} _{a_i}$
3ABCD	$t_1 \rightarrow \lambda$
3A	$l_I \rightarrow l_3 _{t_1}$
3BCD	$l_{II} \rightarrow l_{III} t_2 \quad a_i \rightarrow x_i _{l_{II}} \quad a_i \rightarrow y_i _{l_{II}}$
4BCD	$t_2 \rightarrow t_3$
4B	$l_{III} \rightarrow l_V _{y_i y_i}$
5B	$x_i \rightarrow a_i _{l_V} \quad y_i \rightarrow a_i _{l_V} \quad t_3 \rightarrow \lambda _{l_V} \quad l_V \rightarrow l_{II}$
5CD	$t_3 \rightarrow t_4 _{l_{III}}$
5D	$l_{III} \rightarrow l_{IV} _{t_3 y_i}$
6CD	$t_4 \rightarrow \lambda$
6C	$x_i \rightarrow a_i _{l_{III} t_4} \quad l_{III} \rightarrow l_{II} t_1 _{t_4}$
6D	$x_i \rightarrow a_i _{l_{IV}} \quad l_{IV} \rightarrow l_2 \quad y_i \rightarrow \lambda _{l_{IV}}$

For a better understanding, we give below the table of configurations when a subtraction instruction is simulated, structured with respect to the computational steps and cases (for simplicity we will represent only the content of register i and the related objects that appear during the simulation). Also, the rules that can be applied on actual multisets are specified.

	Case A	Case B	Case C	Case D
Step 1	l_1 $l_1 \rightarrow l_I t_0$	$l_1 a_i^{m_i}$ $l_1 \rightarrow l_I t_0$	$l_1 a_i^{m_i}$ $l_1 \rightarrow l_I t_0$	$l_1 a_i^{m_i}$ $l_1 \rightarrow l_I t_0$
Step 2	$l_I t_0$ $t_0 \rightarrow t_1$	$l_I t_0 a_i^{m_i}, m_i \geq 1$ $t_0 \rightarrow t_1$ $l_I \rightarrow l_{II} _{a_i}$	$l_I t_0 a_i^{m_i}, m_i \geq 1$ $t_0 \rightarrow t_1$ $l_I \rightarrow l_{II} _{a_i}$	$l_I t_0 a_i^{m_i}, m_i \geq 1$ $t_0 \rightarrow t_1$ $l_I \rightarrow l_{II} _{a_i}$
Step 3	$l_I t_1$ $l_I \rightarrow l_3 _{t_1}$ $t_1 \rightarrow \lambda$	$l_{II} t_1 a_i^{m_i}, m_i \geq 1$ $t_1 \rightarrow \lambda$ $l_{II} \rightarrow l_{III} t_2$ $a_i \rightarrow x_i _{l_{II}}$ $a_i \rightarrow y_i _{l_{II}}$	$l_{II} t_1 a_i^{m_i}, m_i \geq 1$ $t_1 \rightarrow \lambda$ $l_{II} \rightarrow l_{III} t_2$ $a_i \rightarrow x_i _{l_{II}}$ $a_i \rightarrow y_i _{l_{II}}$	$l_{II} t_1 a_i^{m_i}, m_i \geq 1$ $t_1 \rightarrow \lambda$ $l_{II} \rightarrow l_{III} t_2$ $a_i \rightarrow x_i _{l_{II}}$ $a_i \rightarrow y_i _{l_{II}}$
Step 4	l_3 ready for next instruction	$l_{III} t_2 x_i^r y_i^p,$ with $r \geq 0, p \geq 2$ $t_2 \rightarrow t_3$ $l_{III} \rightarrow l_V _{y_i y_i}$	$l_{III} t_2 x_i^{m_i}$ $t_2 \rightarrow t_3$	$l_{III} t_2 x_i^{m_i-1} y_i$ $t_2 \rightarrow t_3$
Step 5		$l_V t_3 x_i^r y_i^p,$ with $r \geq 0, p \geq 2$ $x_i \rightarrow a_i _{l_V}$ $y_i \rightarrow a_i _{l_V}$ $t_3 \rightarrow \lambda _{l_V}$ $l_V \rightarrow l_{II} t_1$	$l_{III} t_3 x_i^{m_i}$ $t_3 \rightarrow t_4 _{l_{III}}$	$l_{III} t_3 x_i^{m_i-1} y_i$ $t_3 \rightarrow t_4 _{l_{III}}$ $l_{III} \rightarrow l_{IV} _{t_3 y_i}$
Step 6		like Step 3	$l_{III} t_4 x_i^{m_i}$ $x_i \rightarrow a_i _{l_{III} t_4}$ $l_{III} \rightarrow l_{II} t_1 _{t_4}$ $t_4 \rightarrow \lambda$	$l_{IV} t_4 x_i^{m_i-1} y_i$ $t_4 \rightarrow \lambda$ $x_i \rightarrow a_i _{l_{IV}}$ $l_{IV} \rightarrow l_2$ $y_i \rightarrow \lambda _{l_{IV}}$
Step 7			like Step 3	$a_i^{m_i-1} l_2$ ready for next instruction

Before we start explaining the simulation of the subtraction instruction, let us have a glance to the main idea of the algorithm. We start the computation by checking if the register i is empty or not (i.e., we check whether there exists a symbol a_i). In case the register is empty, we can generate the label of the new instruction to be applied, namely l_3 , therefore we can execute a new instruction of the program (*Case A*). Otherwise (there exists at least one symbol a_i), in a nondeterministic way, we produce $x_i^{m_i-n} y_i^n$ from $a_i^{m_i}$. Now, if $n \neq 1$, then we reestablish the branching

configuration by changing back the objects x_i and y_i to objects a_i (*Cases B, C*); therefore the process can start again. This process will last up to the moment when, after splitting the objects a_i into objects x_i and y_i , we will have only one object y_i . Then, we can continue the computation by deleting the object y_i and changing back the remaining $m_i - 1$ objects x_i into objects a_i . Also, we produce a new object (say l_2) which represents the label of the new instruction to be executed (*Case D*). In this way, we have correctly simulated the decrement instruction.

We start the computation having inside the region the object l_0 representing the initial label which indicates the first instruction to be executed, and the objects $a_1^{k_1} \dots a_n^{k_n}$ representing the initial contents of registers.

More rigorously, let us see what happens step by step during the computation. Initially it is checked if the register i is empty. This is done by first generating objects l_I and t_0 when rule $l_1 \rightarrow l_I t_0$ is applied. In the second step, the object l_I will be transformed into l_{II} iff in the region there exists at least one object a_i .

In case there is no object a_i , only rule $t_0 \rightarrow t_1$ is applied. Next, object t_1 will act as a promoter for rule $l_I \rightarrow l_3|_{t_1}$ and, at the same time, will be deleted by rule $t_1 \rightarrow \lambda$.

If in the region there exists at least one object a_i , rules $t_0 \rightarrow t_1$ and $l_I \rightarrow l_{II}|_{a_i}$ are executed simultaneously in the second step of computation. Now, rule $l_I \rightarrow l_3|_{t_1}$ cannot be executed anymore because object l_I was already transformed into l_{II} in the previous computational step. Therefore, if in the region there exists an object l_{II} , we know for sure that in the region there is also at least one object a_i . As a consequence, in the same step, rules $a_i \rightarrow x_i$ or/and $a_i \rightarrow y_i$ are applied. Due to nondeterminism and because of the maximally parallel mode of functioning of the P systems, all objects a_i present in the region will be transformed into objects x_i and/or y_i .

At the same time, object l_{II} , which descends from object l_1 , will be transformed into l_{III} and t_2 (object t_2 represents a counter which is useful to reestablish the branching configuration if the computation did not work “well”). Now, the computation can split in three possible directions (the number of rules $a_i \rightarrow y_i|_{l_{II}}$ is zero, one or more). Let us consider the first case when we have inside the region objects $x_i^{m_i-n}$ and y_i^n such that $m_i - n \geq 0$, $n \geq 2$, and also objects l_{III} and t_2 .

Since $n \geq 2$, the rules to be applied in the second step are $t_2 \rightarrow t_3$ and $l_{III} \rightarrow l_V|_{y_i y_i}$. Next, the branching configuration is restored by the rules $x_i \rightarrow a_i|_{l_V}$, $y_i \rightarrow a_i|_{l_V}$, $t_3 \rightarrow \lambda|_{l_V}$, $l_V \rightarrow l_{II}$. Recall that promoters can react in the same time with the rules that they promote and also, because of the maximally parallel manner of

applying the rules, we successfully restore the branching configuration.

Let us consider the second case, when, in a similar fashion as before, we will have after two computational steps the multiset $l_I, t_0, a_i^{m_i}, m_i \geq 1$. Then, instead of executing both rules $a_i \rightarrow x_i|_{l_{II}}$ and $a_i \rightarrow y_i|_{l_{II}}$, only one of them is executed, say $a_i \rightarrow x_i|_{l_{II}}$. Therefore, the new configuration is $l_{III}, t_2, x_i^{m_i}$ and the rule to be applied is $t_2 \rightarrow t_3$. Now, since there exists the object l_{III} (which in the previous case is transformed in forth step because there exists two objects y_i), the rule $t_3 \rightarrow t_4$ is applied. Once we have the object t_4 we can restore as before the branching configuration because we know for sure that the rules $a_i \rightarrow x_i|_{l_{II}}$ and $a_i \rightarrow y_i|_{l_{II}}$ where not applied in a proper order.

The third case that may occur represents a successful computation. Recall that the difference between this case and the previous ones occurs after applying the rules $a_i \rightarrow x_i$ and $a_i \rightarrow y_i$ when we will have the objects $x_i^{m_i-1}$ and y_i . The computation is the same as in the third case up to the fifth step, when, in addition, the rule $l_{III} \rightarrow l_V|_{t_3y_i}$ is applied. After this, inside the region we will have the objects $x_i^{m_i-1}$, y_i , t_4 , and l_{IV} . The presence of the object l_{IV} will drive the computation in the “right” way. The rules that will be applied are as follows: $t_4 \rightarrow \lambda$, $x_i \rightarrow a_i|_{l_{IV}}$, $l_{IV} \rightarrow l_2$, $y_i \rightarrow \lambda|_{l_{IV}}$, $l_{IV} \rightarrow l_2$. In this way, starting from the objects $l_1, a_i^{m_i}$, we have successfully computed $l_2, a_i^{m_i-1}$. The object l_2 is useful to indicate that the subtraction instruction was successfully applied and to point out the new instruction to be executed.

The simulation of the register machine will continue until the halting instruction is reached (if the simulated machine halts). So, the *P* system halts on some input iff the simulated register machine accepts the corresponding vector.

In conclusion, we have shown that $UPsIOP_1(ncoo, proR_2) \supseteq PsRE$. By invoking Turing-Church thesis we have the reverse inclusion. Consequently, we have proved that $UPsIOP_1(ncoo, proR_2) = PsRE$. \square

A non-deterministic register machine can be simulated in a very similar way: the addition instruction $(l_1 : ADD(i), l_2, l_3) \in \mathcal{P}$ would correspond to rules $l_1 \rightarrow a_i l_2$, $l_1 \rightarrow a_i l_3 \in R_1$. Based on the proof above, the following corollary holds.

Corollary 3.4.1 $PsOP_1(ncoo, proR_2) = PsRE$.

The membrane contents in the halting configurations correspond to the value of the partial recursive function computed by the simulated register machine, together with its final label (a “witness” that the computation is finished).

In case we consider P systems with object rewriting context-free rules and inhibitors, the following result stands.

Theorem 3.4.2 $UPsIOP_1(ncoo, inhR_2) = PsRE$.

Proof. Consider a register machine $M = (n, \mathcal{P}, l_0, l_h)$. We define the P system

$$\Pi = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where:

$$\begin{aligned} V &= \{a_i, x_i, y_i, x'_i, y'_i, t_i, t'_i \mid 1 \leq i \leq n\} \cup \{l_1 \mid (l_1 : \text{ADD}(i), l_2) \in \mathcal{P}\} \\ &\quad \cup \{l_1, l_I, l_{II}, l_{III} \mid (l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}\} \cup \{c, c'\}; \\ C &= P = \emptyset; \\ I &\subseteq V; \\ \mu &= [\]_1; \\ w_1 &= e_i c a_1^{k_1} \dots a_n^{k_n} t_1 \dots t_n; \\ i_0 &= 1; \end{aligned}$$

R_1 is defined as follows. For simplicity, we will use the following notations:

$$\gamma = t_1 \dots t_n, \quad \gamma_i = t_1 \dots k_{i-1} k_{i+1} \dots t_n, \quad \gamma'_i = k'_1 \dots k'_{i-1} k'_{i+1} \dots k'_n.$$

- for each register i , $1 \leq i \leq n$, we add to R_1 the rules:

$$t'_i \rightarrow t_i, \quad t_i \rightarrow \lambda;$$

- for each instruction $(l_1 : \text{ADD}(i), l_2) \in \mathcal{P}$, we add to R_1 a rule:

$$l_1 \rightarrow l_2 a_i c \gamma;$$

- for each instruction $(l_1 : \text{SUB}(i), l_2, l_3) \in \mathcal{P}$, we add to R_1 the rules:

$$l_1 \rightarrow l_I c \gamma c' \gamma'_i,$$

$$l_I \rightarrow l_3 t_i |_{-a_i},$$

$$l_I \rightarrow l_{II} c \gamma c' \gamma'_i |_{-t_i}, \quad a_i \rightarrow x_i |_{-t_i}, \quad a_i \rightarrow y_i |_{-t_i},$$

$$l_{II} \rightarrow l_I |_{-y_i}, \quad x_i \rightarrow a_i |_{-y_i},$$

$$l_{II} \rightarrow l_{III} c \gamma c' \gamma'_i |_{-t_i}, \quad x_i \rightarrow x'_i |_{-t_i}, \quad y_i \rightarrow y'_i |_{-t_i},$$

$$l_{III} \rightarrow l_2 t_i |_{-y'_i y'_i}, \quad x'_i \rightarrow a_i |_{-y'_i y'_i}, \quad y'_i \rightarrow \lambda |_{-y'_i y'_i},$$

$$l_{III} \rightarrow l_I c \gamma_i |_{-t_i}, \quad x'_i \rightarrow a_i |_{-t_i}, \quad y'_i \rightarrow a_i |_{-t_i};$$

- we also add to R_1 the rules:

$$c' \rightarrow c, \quad c \rightarrow \lambda, \quad e_h \rightarrow \gamma,$$

$$a_i \rightarrow A_i |_{-c}, \quad 1 \leq i \leq n.$$

The four cases corresponding to a subtract instruction (register is zero, register is nonzero and rule $a_i \rightarrow y_i|_{-t_i}$ is applied 0, 1, ≥ 2 times) are illustrated by the table below. For simplicity, symbols $a_{i'}, k_{i'}, k'_{i'}$, $i' \neq i$, and c, c' are skipped and the subscript i is omitted for symbols $a_i, t_i, x_i, y_i, x'_i, y'_i$.

	Case A	Case B	Case C	Case D
Step 1	ek $e \rightarrow l_I k$ $k \rightarrow \lambda$	$ea^n k$ $e \rightarrow l_I k$ $k \rightarrow \lambda$	$ea^n k$ $e \rightarrow l_I k$ $k \rightarrow \lambda$	$ea^n k$ $e \rightarrow l_I k$ $k \rightarrow \lambda$
Step 2	$l_I k$ $l_I \rightarrow zk _{-a}$ $k \rightarrow \lambda$	$l_I ka^n$ $k \rightarrow \lambda$	$l_I k, a^n$ $k \rightarrow \lambda$	$l_I ka^n$ $k \rightarrow \lambda$
Step 3	zk ready for next instruction	$l_I a^n$ $l_I \rightarrow l_{II} k _{-k}$ $a \rightarrow x _{-k}$	$l_I a^n$ $l_I \rightarrow l_{II} k _{-k}$ $a \rightarrow x _{-k}$ $a \rightarrow y _{-k}$	$l_I a^n$ $l_I \rightarrow l_{II} k _{-k}$ $a \rightarrow x _{-k}$ $a \rightarrow y _{-k}$
Step 4		$l_{II} k x^n$ $x \rightarrow a _{-y}$ $l_{II} \rightarrow l_I _{-y}$ $k \rightarrow \lambda$	$l_{II} k x^m y^p$ $k \rightarrow \lambda$	$l_{II} k x^{n-1} y$ $k \rightarrow \lambda$
Step 5		like t_3	$l_{II} x^m y^p$ $x \rightarrow x' _{-k}$ $y \rightarrow y' _{-k}$ $l_{II} \rightarrow l_{III} k _{-k}$	$l_{II} x^{n-1} y$ $x \rightarrow x' _{-k}$ $y \rightarrow y' _{-k}$ $l_{II} \rightarrow l_{III} k _{-k}$
Step 6			$l_{III} x'^m y'^p k$ $k \rightarrow \lambda$	$l_{III} x'^{n-1} y' k$ $l_{III} \rightarrow f k _{-y' y'}$ $y' \rightarrow \lambda _{-y' y'}$ $x' \rightarrow a _{-y' y'}$ $k \rightarrow \lambda$
Step 7			$l_{III} x'^m y'^p$ $x' \rightarrow a _{-k}$ $y' \rightarrow a _{-k}$ $l_{III} \rightarrow l_I _{-k}$	$f a^{n-1} k$ ready for next instruction
Step 8			like t_3	

This construction is somewhat similar to that from the proof of Theorem 3.4.1. The intuitive idea is that instead of using some rule $a \rightarrow y|_p$, the “complement” q is

generated (such that q is absent if and only if p is present), and then $a \rightarrow y|_{-q}$ is used.

Let us explain what happens to the objects a_i . It is clear that one copy of a_i is created during the simulation of the addition instruction. To each register i we associate a symbol t_i which is always present (deleted and recreated) in the system except the halting configuration and the following three situations that may happen during the simulation of decrementing register i :

- The system determines that the value of register i is not zero (step t_3 , cases B, C, D of the table). The rule $l_I \rightarrow l_3 t_i |_{-a_i}$ was not applicable at the previous step, so a_i is rewritten into x_i and/or y_i .
- The system determines that at least one object y_i was produced (step t_5 , cases C, D). The rule $l_{II} \rightarrow l_1 t_i |_{-y_i}$ was not applicable at the previous step, so objects x_i and y_i are primed.
- The system determines that at least two objects y'_i were produced (step t_7 , case C). The rule $l_{III} \rightarrow l_2 t_i |_{-y'_i y'_i}$ was not applicable at the previous step, so objects x'_i and y'_i are rewritten into a_i .

In the last step of the derivation, objects t_i are present, but c is absent, so objects a_i are rewritten to A_i (“terminal symbols” in case we are interested in the final result). In all cases except these four, objects a_i do not evolve: in each step either t_i is produced from some object $l_1, l_I, l_{II}, l_{III}$ during simulation of the register machine instruction with label l_1 (any addition; decrementing register i step t_1 , or step t_2 case A, or step t_3 , or step t_4 case B, or step t_5 , or step t_6 case D, or step t_7), or t_i is produced from t'_i (decrementing register other than i , step t_2 cases B, C, D, step t_4 cases C,D, step t_6 case C).

As for object l_1 representing the label of the currently simulated instruction, in case of addition it is replaced with the label l_2 of the next instruction, in case of subtraction case A simulates the zero-test, changing l_1 to l_2 , case B represents an attempt to subtract zero, returning to the configuration of step t_3 , case C represents an attempt to subtract more than one, also returning to the configuration of step t_3 , and case D represents subtracting one, changing l_1 to l_2 . The label l_h of the halt instruction is erased, and the final configuration only has objects a_i , each in the multiplicity representing the value of the corresponding register i , at the time when the register machine halts.

In conclusion, we have shown that $UPsIOP_1(ncoo, inhR_2) \supseteq PsRE$. By invoking the Turing-Church thesis we have the reverse inclusion. Consequently, we have proved that $UPsIOP_1(ncoo, inhR_2) = PsRE$. \square

A non-deterministic register machine can be simulated in a very similar way: the addition instruction $(l_1 : ADD(i), l_2, l_3) \in \mathcal{P}$ would correspond to rules $l_1 \rightarrow l_2 a_i c \gamma$, $l_1 \rightarrow l_3 a_i c \gamma \in R_1$. Based on the proof above, the following corollary holds.

Corollary 3.4.2 $PsOP_1(ncoo, inhR_2) = PsRE$.

3.5 Open Problems and Forthcoming Research

In this chapter we have succeeded to characterize the exact computational power of almost all models of *P* systems with promoters and inhibitors. However there are still several open questions regarding, for instance, the lower bound of symbols that, acting as promoters/inhibitors, make the catalytic *P* systems models universal, or how many symbols suffice in order to generate/recognize *NRE*.

In proving the universality of *P* systems with promoters/inhibitors of weight two, we encounter another interesting problem: does such a system remain universal if we forbid promoters/inhibitors of weight two of the form xx , $x \in V$? We conjecture that such systems are not universal because it seems that the only way to simulate the subtract instruction is to use such promoted/inhibited rules.

In what concerns *P* systems with non-cooperative promoted rules we were not able to give the precise characterization of computational capabilities. However, we were able to set up an upper bound by simulating them with a particular sequential random context grammar, presumably not universal. Nevertheless, we conjecture that such *P* systems not only are not universal but even more, they equal the family of sets of vectors of numbers generated by ETOL systems. To support this conjecture we sketch some clues that might give a hint of how such a result can be established.

We will try to simulate the execution of promoted rules using inhibited rules. Fundamental for the simulation will be the universal clock and the maximal parallelism manner of executing the rules (recall that in sequential case, random context grammars with only permitting contexts are incomparable with random context grammars with only forbidding contexts). Let us assume that we want to simulate the executions of non-cooperative promoted rules:

$$A \rightarrow \alpha_1|_{p_1},$$

$$A \rightarrow \alpha_2|_{p_2},$$

...

$$A \rightarrow \alpha_k|_{p_k}.$$

Here is how the simulation is done. We will consider for each non-cooperative promoted rule a symbol r_i , $1 \leq i \leq k$. Then we will have the following non-cooperative (inhibited) rules.

$$r_i \rightarrow \bar{r}_i|_{\neg p_i}, 1 \leq i \leq k,$$

$$A \rightarrow \bar{A}X,$$

$$X \rightarrow \lambda,$$

$$\bar{A} \rightarrow \bar{\bar{\alpha}}_i|_{\neg \bar{r}_i}, 1 \leq i \leq k,$$

$$\bar{A} \rightarrow A|_{\neg X},$$

$$\bar{\bar{B}} \rightarrow B \text{ for any } B \in V,$$

$$\bar{r}_i \rightarrow r_i|_{\neg X}, 1 \leq i \leq k.$$

Assume that we start from a given configuration represented by the string $w = A^t r_1 r_2 \dots r_k$. A rule $r_i \rightarrow \bar{r}_i|_{\neg p_i}$ (that corresponds to $A \rightarrow \alpha_i|_{p_i}$) check whether or not the object p_i is present. If there is not present any object p_i , $1 \leq i \leq k$ (that means the symbol A should not be rewritten by any α_i , $1 \leq i \leq k$) then the objects \bar{r}_i , $1 \leq i \leq k$ are generated. Simultaneously, the rule $A \rightarrow \bar{A}X$ runs and produces objects $\bar{A}X$. In the next step, remark that any rule $\bar{A} \rightarrow \bar{\bar{\alpha}}_i|_{\neg \bar{r}_i}$ cannot be executed since the object \bar{r}_i is present. However, the rule $X \rightarrow \lambda$ can be executed allowing in the next step the execution of the rules $\bar{A} \rightarrow A|_{\neg X}$ and $\bar{r}_i \rightarrow r_i|_{\neg X}$. Therefore, the initial configuration is restored.

Let us see what is happening if at least one rule $r_i \rightarrow \bar{r}_i|_{\neg p_i}$ is not applied because there exists an object p_i . This means that in the second step the rule $\bar{A} \rightarrow \bar{\bar{\alpha}}_i|_{\neg \bar{r}_i}$ is executed since the object \bar{r}_i was not generated. Next, the rules $\bar{\bar{B}} \rightarrow B$ for any $B \in V$ accomplish the simulation.

Here we have presented a way to simulate the execution of promoted non-cooperative promoted rules by using non-cooperative inhibited rules. First observe that in three steps or we reestablish the initial configuration or we proceed with the rewriting providing that the conditions are fulfilled.

However, the problem we encountered is the following. When we simulate a rule $A \rightarrow \alpha_i|_{p_i}$ and we remark that there is no object p_i and thus we cannot continue, we reestablish the initial configuration since there might be other rules acting in the system (say $C \rightarrow \gamma|_h$ with $A \neq C$) in the mean time and therefore the computation continues. The simulation works perfectly until in the P systems with promoted

rules there are rules that can be still executed. Nevertheless, once the computation of the the *P* system with promoters is finished, in our simulation we will still cycle, changing and then reestablishing a configuration. So, the problem that we still have to overcome is how to halt correctly.

In any case, proving such result might allow us to handle a whole range of other problems within *P* systems framework, in a more intuitive manner than using *P* systems with inhibitors of weight 1 at the level of non-cooperative rules. Also, in a straightforward manner all these results can be extended to parallel rewriting of strings working under some context constraints (to Lindenmayer systems, for instance).

Chapter 4

P Systems Generalizations

The question of how errors spread and propagate in cooperative systems has been studied in a variety of fields. This chapter is devoted to several bio-inspired generalizations of the classical P system model and is motivated by the issue of resilience in face of errors. Here we present systems that are immune to errors regarding the execution times of rules and the degree of the rewriting parallelism.

We introduce and study *timed P systems with promoters/inhibitors* where a time is associated with the execution/application of any rule of the system in any given configuration. We then find a proper subclass of such systems able to generate the same family of sets of vectors of numbers regardless the values of the times associated to the rules.

In Section 4.2 we proceed by defining systems computing by using a variable parallelism instead of the maximal one. Similarly as above, we point out a proper subclass of such systems able to generate the same family of sets of vectors of numbers regardless the parallelism of the rewriting process in a given configuration.

4.1 Timed P Systems

A standard feature of membrane computing is the fact that each rule is executed in exactly one time-unit; however, this mathematical feature, in general, does not have a counterpart in cell biology. Chemical reactions may take certain times to be executed. In many cases, different reactions with different execution times are synchronized via biological signals that move across different areas present in the cell; some considerations on this topic can be found in [6] where a class of P systems that uses signals has been introduced. On the other hand, it is true that in a cell, when a reaction is applied, “almost” all the chemicals that can be subject of the reaction are transformed, in a parallel manner.

In cell biology, the execution time of a chemical reaction might be difficult to be known precisely and usually such a parameter is very sensitive to environmental factors that might be hard to control. For instance, a certain reaction whose execution time depends on the environment temperature will behave differently even in the same cell region because the propagation of heat is not uniform in non-homogeneous mediums.

Therefore, we believe that it is extremely interesting to construct systems that work in the way we expect, independently from the values associated to the execution times of the rules.

For this reason here we introduce the concept of *time-independent* *P* systems. Informally, a *time-independent* *P* system is a *P* system that always produces the same result, independently from the execution times of the rules.

Starting from these considerations, we initially define and investigate a model of *P* systems where to each rule r a certain positive integer value $e(r)$ is associated that indicates its execution time at a certain moment during the computation.

A *P* system that generates (or accepts) the same family of vectors of natural numbers, independently from the value assigned to the execution time of each rule r , is called *time-free*.

In this way, a time-free *P* system can be considered stable against environmental factors that might influence the execution times of the rules.

As in the real world where synchronization of independent events is fundamental and processes are governed by the principle of causality, also in the case of time-free *P* systems a similar problem must be considered: in order to get time-free systems that are computationally powerful enough it is important to synchronize rules with different execution times and that run in parallel.

Later we will consider another possible model of time-independent *P* systems. We will investigate *P* systems where each application of a rule r has associated a certain finite positive integer/rational/real value that indicates its execution time. In this respect, we define a class of *P* systems, called *clock-free* *P* systems, that produces the results independently from the times associated with the applications of the rules.

4.1.1 Time Free *P* Systems

Let us consider a *P* system with catalysts and promoters/inhibitors defined as presented in Chapter 3, $\Pi = (V, C, P, I, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$. In addition to

this we associate to each rule an integer representing its execution time, at a certain moment during computation.

Specifically, given a computable mapping

$$e : R_1 \cup \dots \cup R_m \longrightarrow \mathbb{N},$$

and a system Π as considered above, we construct a *timed P system* $\Pi(e) = (V, C, P, I, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0, e)$ working in the following way.

We suppose to have an external clock (that does not have any influence on the system) that marks time-units of equal length, starting from time 0.

To each region of the system is associated a finite number of objects and a finite number of rules.

At each time, in the regions of the system we have together rules in execution and rules not in execution. At each time, all the rules that can be applied (started) in each region, must be applied.

If a rule $r \in R_i$, $1 \leq i \leq m$, is applied, then all objects that can be processed by the rule have to evolve by this rule.

To apply a rule $u \rightarrow v$, $u \rightarrow v|_z$, or $u \rightarrow v|_{\neg z}$ in a region i means to remove the multiset of objects identified by u from region i , and to add the objects specified by the multiset v , into the regions specified by the target indications associated to each object in v .

When a rule r is started at time j , then its execution terminates at time $j + e(r)$ (the objects produced as well as the promoters moved by the rule can be used starting from the time $j + e(r)$).

If two rules start at the same time, then possible conflicts for using the occurrences of objects are solved by assigning the occurrences in a non-deterministic way. The rules are applied in the maximally parallel manner as usually defined in the P systems framework.

Notice that, when a rule r is started, the occurrences of objects used by this rule are not available for other rules during the entire execution time of r .

The computation stops when no rule can be applied in any region and there are no rules in execution (the system has reached an *halting configuration*).

The output of an halting computation is the vector of numbers representing the multiplicities of objects present in the output region in the halting configuration.

Collecting all the vectors obtained, for any possible halting computation, we get the set $Ps(\Pi(e))$ of vectors of natural numbers generated by the system $\Pi(e)$.

For shortness, in what follows, a P system not using promoters or inhibitors (i.e., the sets P and I are empty) is called basic.

Example 4.1.1 In Figure 4.1 we illustrate the first steps of a computation of a timed P system $\Pi = (V, C, P, I, \mu, w_1, R_1, i_0, e)$, defined as follows:

- $V = \{A, B, C\}$;
- $C = P = I = \emptyset$;
- $\mu = []_1$;
- $w_1 = AAB$;
- $R_1 = \{A \rightarrow AB, B \rightarrow C\}$;
- $i_0 = 1$;
- the first values of the mapping e are given in the Figure 4.1.

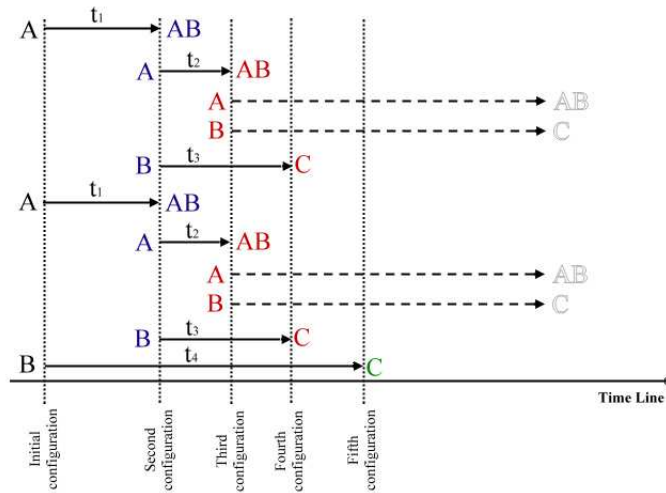


Figure 4.1: The computation of a timed P system

Observe that since in the first configuration we have in the region the multiset AAB , then after t_1 time-units, the rule $A \rightarrow AB$ terminates its execution and produces the multiset $ABAB$. Next, the same rule is executed once more and finishes its execution in t_2 time-units (with t_1 not necessarily equals with t_2) and the new configuration consists of the multiset $ABAB$. Meanwhile, the rule $B \rightarrow C$ (that acts on the object B present in the initial configuration) is still in execution and it will end after t_4 time-units. However, during all this time the rules $B \rightarrow C$ is executed once more (this rule acts on the object B that was produced by the first execution of the rule $A \rightarrow AB$). The system continues its computation by applying the rules according with the computable mapping. Finally remark that, in a given configuration, each application of a rule has the same execution time.

A P system $\Pi = (V, C, P, I, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$ is *time-free* if and only if every timed system in the set

$$\{\Pi(e) \mid e : R \longrightarrow \mathbb{N}, e \text{ computable}\},$$

where $R = R_1 \cup \dots \cup R_m$, produces the same set of vectors of natural numbers.

We use the notation

$$PsTOP_m^\gamma(\alpha, \beta), \gamma \in \{t\}, \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}, \beta \in \{proR_i, inhR_i\}$$

to denote the family of sets of vectors of natural numbers generated/accepted by *timed* P systems, having the *time-free* property ($\gamma = t$), with at most m membranes, evolution rules that can be non-cooperative (*ncoo*), or catalytic (*cat_k*), using at most k catalysts (as usual, $*$ is used if the corresponding number of membranes, or catalysts is not known) and promoters (*proR_i*) or inhibitors (*inhR_i*) of weight i at the level of rules.

We present a simple example of a time-free P system using two membranes, one promoter, and non-cooperative rules that generates the set $\{2^n \mid n \geq 0\}$. From this example, it is clear how promoters are useful to synchronize rules with different execution times.

We consider the system

$$\Pi = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where:

$$\begin{aligned} V &= \{a, p\}; \\ C &= I = \emptyset; \\ P &= \{p\}; \\ \mu &= []_1; \\ w_1 &= ap; \\ R_1 &= \{a \rightarrow aa|_p, p \rightarrow p, p \rightarrow \lambda\}; \\ i_0 &= 1. \end{aligned}$$

The rule $a \rightarrow aa|_p$ is activated by the promoter p which is present at the beginning of the computation in region 1. If the rule $p \rightarrow p$ is executed then the rule $a \rightarrow aa|_p$ is applied an arbitrary number of times in the maximally parallel way. In case promoter p is deleted from the region by the rule $p \rightarrow \lambda$, then the computation halts with a number of objects in the output region that is a power of 2. It is easy to see that the system generates the Parikh image of $\{a^{2^n} \mid n \geq 0\}$ independently from the execution times of the rules and, therefore, the system Π is time-free.

Now we will show that there exist systems using catalytic rules that are not time-free.

Example 4.1.2 Consider the following *P* system with non-cooperative and catalytic rules (with only one catalyst):

$$\Pi = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where:

$$V = \{B', B'', c, X, D, b, a\};$$

$$C = \{c\};$$

$$P = I = \emptyset;$$

$$\mu = []_1;$$

$$w_1 = B'B''c;$$

$$R_1 = \{r_1 : B' \rightarrow b_{out}X, r_2 : cB'' \rightarrow c, r_3 : X \rightarrow D, r_4 : cX \rightarrow c, r_5 : D \rightarrow a_{out}\};$$

$$i_0 = 0.$$

The system Π is not time-free. To prove this assertion, we show that it generates two different sets of vectors in case two different time-mappings (e' and e'') are considered.

These mappings are defined as follows:

$$e'(r_1) = 1,$$

$$e'(r_2) = 2,$$

$$e'(r_i) = k', \quad \text{for some } k' \in \mathbb{N}, \text{ for } i = 3, 4, 5.$$

$$e''(r_1) = 2,$$

$$e''(r_2) = 1,$$

$$e''(r_i) = k'', \quad \text{for some } k'' \in \mathbb{N}, \text{ for } i = 3, 4, 5.$$

If the execution times of the rules in Π are given by the mapping e' , then one can see that $Ps(\Pi(e')) = \{(1, 1)\}$. In fact, both rules r_1 and r_2 are started in parallel. When the rule r_1 terminates its execution, the objects b and X are produced. Then, at step 2, the rule r_3 is applied and the object D is produced. Notice that, in step 2, the rule r_3 is the only one that can be applied, and this is because rule r_2 is still in execution (therefore, the catalyst c is busy). In step 3, the object a is produced by $D \rightarrow a_{out}$ and sent to the environment. Therefore this is the only possible computation and the output of the system is the set $\{(1, 1)\}$.

If the execution times of the rules in Π are associated by using the mapping e'' , then $Ps(\Pi(e'')) = \{(1, 0), (1, 1)\}$.

In fact, rules r_1 and r_2 are started in parallel as in the previous case but, because of the time-mapping e'' , rule r_2 ends after rule r_1 . Then, after two steps, object X is produced and b is sent to the environment; in the next step, X can be rewritten in two possible ways. In the first way the catalyst c can be used to apply rule r_4 and then the computation halts producing as output the vector $(1, 0)$. In the second way rules r_3 and then r_5 are used. Rule r_5 produces and sends the object a to the environment. Therefore, in this case, we have as the output of the system the set $\{(1, 1), (1, 0)\}$. Notice that for any computable time-mapping e , $Ps(\Pi(e)) \neq \emptyset$.

4.1.2 Clock-Free P Systems

The concept of timed P system can be further more extended. However, for the sake of clarity, we will present here only an informal description of such model and its way of performing computations.

In a similarly fashion as in the case of timed P systems we can define *clocked* P systems by associating a time value (via a computable mapping) to each application of the rule (and not for the execution, i.e. for all applications, of a rule like it was defined for timed P systems), at a certain moment during computation. This value constitutes the time needed for that application to take place and can be a natural, rational, or real value.

Starting from an *initial configuration*, the system evolves according to the rules and objects present in the membranes, in a non-deterministic maximally parallel manner. The m -tuple of multisets of objects present at any moment in the m regions of Π constitutes the *configuration* of the system at that moment. The m -tuple (w_1, w_2, \dots, w_m) constitutes the initial configuration of Π . Starting from a given configuration and having at least one rule in execution, we will consider the *next configuration* the instant description of the machine when a previous initiated reaction ends its application.

The objects that can evolve from a given configuration as well as the rules by which they evolve are chosen in a non-deterministic manner and also, in case of rules, in a maximal parallel manner. 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. The objects which remain unassigned are left where they are,

and they are passed unchanged to the *next configuration*.

The system will make a successful computation if and only if it halts: there is no rule applicable to the objects present in the halting configuration, nor reactions in execution. The result of a successful computation is the vector of numbers representing the multiplicities of objects present in the output region in the halting configuration of Π . If the computation never halts, then we will have no output. Collecting all the vectors obtained, for any possible halting computation, we get a set of vectors of natural numbers generated by the system Π .

Example 4.1.3 In Figure 4.2 we illustrate a computation of a clocked *P* system $\Pi = (V, C, P, I, \mu, w_1, R_1, i_0, e)$, defined as follows:

- $V = \{A, B, C\}$;
- $C = P = I = \emptyset$;
- $\mu = []_1$;
- $w_1 = AAB$;
- $R_1 = \{A \rightarrow AB, B \rightarrow C\}$;
- $i_0 = 1$;
- the first values of the mapping e are given in the Figure 4.2.

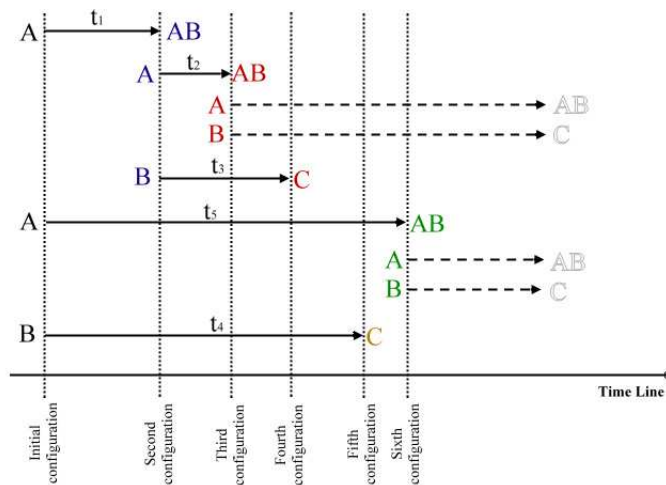


Figure 4.2: The computation of a clocked *P* system

Observe that since in the first configuration we have in the region the multiset AAB , then after t_1 time-units of the external clock the rule $A \rightarrow AB$ terminates its first application. Therefore the second configuration is represented by the multiset AB . Meanwhile the second application of the rule $A \rightarrow AB$ is running and will finish only after t_5 time-units. However, during all this time, the rules $A \rightarrow AB$ and $B \rightarrow C$ are executed once more (they are acting on the objects produced by the first application of the rule $A \rightarrow AB$). The system continues its computation by applying the rules according with the computable mapping. Finally remark that, in a given configuration, different applications of the same rule might have different execution times.

Here we propose a second variant of time-independent P systems, slightly different from the previously considered time-free P systems: in this case, the execution time is associated directly to the applications of the rules. In particular, we consider the class of *clock-free* P systems producing the result independently from the times associated to the applications of rules.

Definition 4.1.1 A clock-free P system (*in short, a P^c system*) of degree $m \geq 1$, with catalysts and promoters/inhibitors is a construct

$$\Pi = (V, C, P, I, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

with the components defined as usual and with the computation defined as follows.

The rules are used as usual, but each application of a rule, in a given configuration, has an execution time that is an arbitrary positive integer (rational or real) number. Different applications (even of the same rule) may have different execution times. All objects produced by a rule are used (if they can be) in the same time as soon as they appear. Under these conditions, the system Π produces the same set of vectors of natural numbers regardless the time values associated with different applications of the rules.

As usual we denote by $Ps(\Pi)$ the set of vectors of natural numbers generated by the system Π at the end of any possible halting computation.

We use the notation:

$$PsCOP_m^\gamma(\alpha, \beta, S),$$

where $\gamma \in \{c\}$, $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}$, $\beta \in \{proR_i, inhR_i \mid i \geq 1\}$, and $S = \mathbb{N}, \mathbb{Q}^+$, or \mathbb{R}^+ , to denote the family of sets of vectors of natural numbers generated by clocked P systems, having the clock-free property ($\gamma = c$), having at most m membranes, evolution rules that can be non-cooperative (*ncoo*), cooperative (*coo*), or catalytic (*cat_k*), using at most k catalysts, and promoters (*proR_i*) or inhibitors (*inhR_i*) of weight i at the level of rules. The execution time for any application of any rule is a value from the set S .

The following example shows how a clock-free P system can generate the set $\{(2^n, 1) \mid n \geq 1\}$. The example anticipates the universality result given later on. Notice that, in case of clock-free P systems, applications of the same rule might have associated different execution times; therefore the approach used in the example presented in Section 4.1.1 cannot be used anymore.

Example 4.1.4 *Let us consider the clock-free *P* system*

$$\Pi = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where:

$$\begin{aligned} V &= \{S, S_1, S_2, \overline{S_1}, \overline{S_2}, \overline{T}, \overline{F}\}; \\ C &= \{c\}; \\ P &= \{S_1, \overline{S_1}, F, \overline{F}, A, B, T_1, \overline{T_1}\}; \\ \mu &= [\]_1; \\ I &= \emptyset; \\ w_1 &= \{A, S, c\}; \\ R_1 &= \{S \rightarrow S_1, S \rightarrow \lambda, cA \rightarrow cBBF|_{S_1}\} \\ &\cup \{S_1 \rightarrow S_2T|_F, F \rightarrow \lambda|_{S_1}, S_2 \rightarrow S_1|_A\} \\ &\cup \{cT \rightarrow cT_1, S_2 \rightarrow \overline{S_1}|_{T_1}, T_1 \rightarrow \lambda\} \\ &\cup \{cB \rightarrow cA\overline{F}|_{\overline{S_1}}, \overline{S_1} \rightarrow \overline{S_2}\overline{T}|_{\overline{F}}, \overline{F} \rightarrow \lambda|_{\overline{S_1}}\} \\ &\cup \{\overline{S_2} \rightarrow \overline{S_1}|_B, c\overline{T} \rightarrow c\overline{T_1}, \overline{S_2} \rightarrow S|_{\overline{T_1}}, \overline{T_1} \rightarrow \lambda\}; \\ i_0 &= 1. \end{aligned}$$

Here is how the system performs the computation. Rules $S \rightarrow S_1$, $S \rightarrow \lambda$ represent a selector, i.e., they decide whether the generation should stop or continue. In the case generation should continue ($S \rightarrow S_1$ is applied), then, after the appearance of object S_1 , the rule $cA \rightarrow cBBF|_{S_1}$ is executed. It will take an arbitrary time (but finite) up to the moment when objects B and F appear simultaneously. In that moment, rules $S_1 \rightarrow S_2T|_F$, $F \rightarrow \lambda|_{S_1}$ can be applied and will be started simultaneously. We have the following situation: object F will be eventually deleted so it will not count in further computations; objects S_2 and T appear synchronously. After this, rule $cT \rightarrow cT_1$ is started; however we do not know if the rule starts in the same time with rule $S_2 \rightarrow S_1|_A$ (because we do not know if all objects A were already rewritten). The usage of catalyst c by rule $cT \rightarrow cT_1$ assures us that rule $cA \rightarrow cBBF|_{S_1}$ cannot be executed before rule $cT \rightarrow cT_1$ ends (this guarantees that no other object T is produced before rule $cT \rightarrow cT_1$ ends).

If there still exist objects A , then object S_1 is generated (by rule $S_2 \rightarrow S_1|_A$) and object T_1 will be eventually deleted.

Otherwise, object $\overline{S_1}$ is generated and we can start the transformation of all the B s into A s (with a quite similar construction).

In case the process should stop, the object S is deleted ($S \rightarrow \lambda$ is executed), the transformation of objects A into B is not repeated and the computation ends having in the output region the set $\{(2^n, 1) \mid n \geq 1\}$.

As a final remark, one can observe that without the parallelism involved while executing the rules $S \rightarrow S_2T|_F$ and $F \rightarrow \lambda|_{S_1}$ the system presented is sequential; therefore the constructed system is clock-free.

The following theorem shows the computational capabilities of clock-free P systems when non-cooperative rules are used.

Theorem 4.1.1

$$\begin{aligned} PsCOP_m^c(ncoo, \mathbb{N}) &= PsCOP_m^c(ncoo, \mathbb{Q}^+) = PsCOP_m^c(ncoo, \mathbb{R}^+) \\ &= PsCF, \text{ for } m \geq 1. \end{aligned}$$

Proof. Observe that $PsCOP_m^c(ncoo, S) = PsCOP_1^c(ncoo, S)$ for $S = \mathbb{N}, \mathbb{Q}^+$, or \mathbb{R}^+ . Also, remark that each computation of such P systems can be described by a tree storing on the yield the result of the computation. Assume now that the lengths of the edges encode the execution time of the corresponding rules. Because the yield of the tree does not depend on the length of the edges then we have that $PsCOP_m^c(ncoo, S) = PsCF$ for $S = \mathbb{N}, \mathbb{Q}^+$, or \mathbb{R}^+ . \square

Based on the above theorem we have the following result regarding the computational power of time-free P systems with non-cooperative rules.

Corollary 4.1.1 $PsTOP_m^t(ncoo) = PsCF$.

Here, we present a universality result concerning clock-free P systems using promoters at the level of rules and one catalyst. The proof is based on the simulation of register machines.

Theorem 4.1.2 $PsCOP_2^c(cat_1, proR_1, \mathbb{N}) = PsRE$.

Proof. In order to prove this assertion, we consider an n -register machine $M = (n, \mathcal{P}, l_0, l_h)$ and we define the P system

$$\Pi = (V, C, P, I, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

$$\begin{aligned}
V &= \{a_r, A_r, S_r \mid 1 \leq r \leq n\} \cup \{E, T, F, F_1, F_2\} \cup \{l, \bar{l} \mid l \in \text{Lab}(\mathcal{P})\}; \\
C &= \{c\}; \\
P &= \{a_r, A_r, S_r \mid 1 \leq r \leq n\} \cup \{E, T, F_1, F_2, h\}; \\
I &= \emptyset; \\
\mu &= [[]_2]_1; \\
w_1 &= \emptyset; \\
w_2 &= cl_0; \\
i_0 &= 1; \\
R_1 &= \emptyset;
\end{aligned}$$

and the rules of the set R_2 are defined as follows:

- for each instruction $(l_1 : \text{ADD}(r), l_2, l_3) \in \mathcal{P}$, we add to R_2 the rules:

$$\begin{aligned}
l_1 &\rightarrow \bar{l}_1 A_r, \\
c &\rightarrow ca_r E|_{A_r}, \\
A_r &\rightarrow \lambda, \\
\bar{l}_1 &\rightarrow l_2|_E, \\
\bar{l}_1 &\rightarrow l_3|_E, \\
E &\rightarrow \lambda;
\end{aligned}$$

- for each instruction $(l_1 : \text{SUB}(r), l_2, l_3) \in \mathcal{P}$, we add to R_2 the rules:

$$\begin{aligned}
l_1 &\rightarrow \bar{l}_1 S_r T F, & S_r &\rightarrow \lambda, \\
ca_r &\rightarrow cE|_{S_r}, & \bar{l}_1 &\rightarrow l_2|_E, \\
T &\rightarrow \lambda|_{a_r}, & \bar{l}_1 &\rightarrow l_3|_{F_2}, \\
T &\rightarrow \lambda|_{F_2}, & E &\rightarrow \lambda|_{F_1}, \\
F &\rightarrow F_1, & F_1 &\rightarrow F_2|_T, \\
F_1 &\rightarrow \lambda|_E, & F_2 &\rightarrow \lambda.
\end{aligned}$$

- for instruction $(l_h : \text{HALT})$ we add to R_2 the rules:

$$\begin{aligned}
a_1 &\rightarrow a_{1_{out}}|_{l_h}, \\
&\dots \\
a_k &\rightarrow a_{k_{out}}|_{l_h}, \\
l_h &\rightarrow \lambda.
\end{aligned}$$

Before we start analyzing the work of Π , let us recall the following. Objects l_1 , l_2 , l_3 correspond to the register machine instruction labels l_1 , l_2 and l_3 respectively; the multiplicity of object a_r represents the value stored in register r ; object A_r represents the incrementation command (it corresponds to the *ADD* operation in the register machine definition); object S_r represents the subtraction command (it correspond to *SUB* operation in the register machine definition).

Here is how the simulation of the register machine increment instruction ($l_1 : \text{ADD}(r), l_2, l_3) \in \mathcal{P}$ works.

Suppose that the current configuration of the region 2 of Π is represented by the multiset $ca_1^{n_1} \dots a_r^{n_r} \dots a_k^{n_k} l_1$. Obviously, only rule $l_1 \rightarrow \bar{l}_1 A_r$ can be applied. After a while a the systems reaches a configuration represented by the multiset $ca_1^{n_1} \dots a_k^{n_k} \bar{l}_1 A_r$; therefore the rules to be further applied are $c \rightarrow ca_r E|_{A_r}$ and $A_r \rightarrow \lambda$. Both rules are started at the same time.

Object A_r will be deleted so we do not have to worry when this will actually happen. After some time, objects a_r and E will appear simultaneously; then, the configuration of the region 2 of Π will be the one represented by the multiset $ca_1^{n_1} \dots a_r^{n_r+1} \dots a_k^{n_k} \bar{l}_1 E$. Next, the rules to be executed simultaneously will be $\bar{l}_1 \rightarrow l_2|_E$ or $\bar{l}_1 \rightarrow l_3|_E$, and $E \rightarrow \lambda$. Because we have obtained the next instruction label and moreover we deleted useless objects, we have correctly simulated the register machine increment instruction.

Here is how the simulation of the register machine subtraction instruction ($l_1 : \text{SUB}(r), l_2, l_3) \in \mathcal{P}$ works. Suppose that the current configuration of the system is given by the multiset $ca_1^{n_1} \dots a_r^{n_r} \dots a_k^{n_k} l_1$. As it can be seen, only the rule $l_1 \rightarrow \bar{l}_1 S_r T F$ can be applied. This means that after a while we will have the multiset $ca_1^{n_1} \dots a_j^{n_j} \dots a_k^{n_k} \bar{l}_1 S_r T F$ (recall that objects \bar{l}_1 , S_r , T , and F have appeared simultaneously since they were produced from the “same” object l_1). Next, the rules $ca \rightarrow cE|_{S_r}$, $T \rightarrow \lambda|_{a_r}$, $F \rightarrow F_1$, and $S_r \rightarrow \lambda$, will start their execution in the same time. However, we do not know if the objects E and F_1 will appear simultaneously. If object F_1 appears before object E it cannot be rewritten by rule $F_1 \rightarrow F_2|_T$ because object T is missing (object T is involved in rule $T \rightarrow \lambda|_{a_r}$); only rule $F_1 \rightarrow \lambda|_E$ could be further applied, but only after object E has appeared. If object E appears into region, then the following rules are applied $F_1 \rightarrow \lambda|_E$, $\bar{l}_1 \rightarrow l_2|_E$, $E \rightarrow \lambda|_{F_1}$.

In this way, the next instruction label l_2 is generated and the simulation can continue. We do not need to know when objects F_1 and E will be removed since they cannot be involved in other rules.

Now, suppose that current configuration of the system is represented by the

multiset $ca_1^{n_1} \dots a_{r-1}^{n_{r-1}} a_{r+1}^{n_{r+1}} \dots a_k^{n_k} l_1$. As above, only the rule $l_1 \rightarrow \bar{l}_1 S_r T F$ can be applied. This means that after a while we will have in the region the multiset $c\bar{l}_1 S_r T F a_1^{n_1} \dots a_{j-1}^{n_{j-1}} a_{j+1}^{n_{j+1}} \dots a_k^{n_k}$.

Next, the rules $F \rightarrow F_1$, $D_a \rightarrow \lambda$ will start their execution simultaneously. Once the object F_1 has appeared the rule to be further applied is $F_1 \rightarrow F_2|_T$.

If object F_2 has appeared, then the system will execute the rules $T \rightarrow \lambda|_{F_2}$, $F_2 \rightarrow \lambda$, $\bar{l}_1 \rightarrow l_2|_{F_2}$.

After some time the next instruction label will be generated and the computation can continue. Finally, if object l_h is generated then all the following rules are executed $a_1 \rightarrow a_{1_{out}}|_{l_h}$, \dots , $a_k \rightarrow a_{k_{out}}|_{l_h}$, $l_h \rightarrow \lambda$. The reason for sending all objects a_r , $1 \leq r \leq n$, into region 1 is that we do not want to consider in the halting configuration the catalyst c .

In conclusion, we have shown that: $PsCOP_2^c(cat_1, proR_1, IN) \supseteq PsRE$. By invoking Turing-Church thesis we have the reverse inclusion. Consequently, we have proved that $PsCOP_2^c(cat_1, proR_1, IN) = PsRE$. \square

Since the above proof works also for time-free P systems we have the following corollary:

Corollary 4.1.2 $PsTOP_2^t(cat_1, proR_1) = PsRE$.

Notice that not all P systems with non-cooperative and catalytic promoted rules are *clock-free*. The following theorem shows that when considering rational or real values for the execution times of the rules we can accept non-computable (by classical Turing machines) sets of vectors.

Theorem 4.1.3 $PsCOP_2(cat_1, proR_1, Q^+) = PsCOP_2(cat_1, proR_1, \mathbb{R}^+) \supset PsRE$.

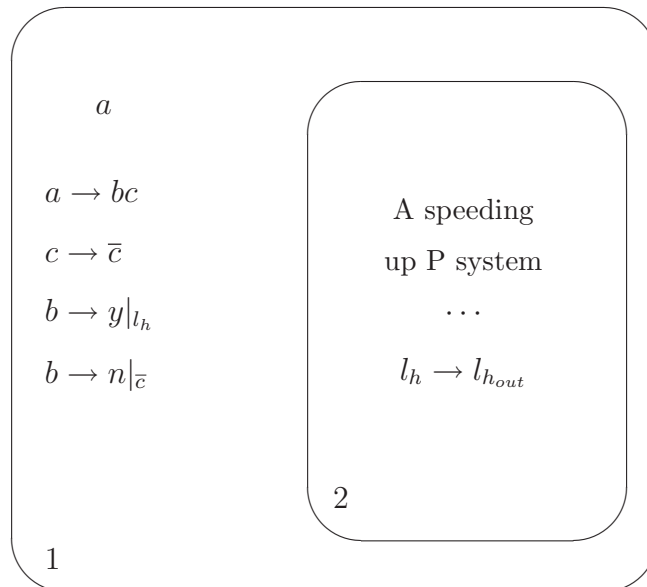
Proof. In order to prove this result we will show that on such P systems (that allows rational or real values for the execution times of each application of a rule) the halting problem formulated for the classical Turing Machine is decidable.

Remember that, in the proof of Theorem 4.1.2, a constructed clock-free P system with one catalyst and promoters at the level of rules simulates a given non-deterministic register machine. In a straightforward manner, using a similar construction, we can make a clock-free P system with one catalyst and promoters at the level of rules that deterministically simulates the computation of an arbitrarily deterministic register machine; in the last step of a successful computation the object labeled l_h is produced.

Consider now such a deterministic P system Π and let each application of a rule is sped up such that its execution time is the half of the previous executed rule. This means that the time for all computations (halting or not) are described by a sum of type $1 + \frac{1}{2} + \frac{1}{4} + \dots + \frac{1}{2^{k-1}} + \frac{1}{2^k}$. In case we have a halting computation, k is finite. Observe that series is convergent and

$$\lim_{n \rightarrow \infty} \sum_{k=0}^n \frac{1}{2^k} = 2.$$

For the halting computations we have produced the symbol l_h . Encode now a universal Turing machine into the multiset (use Gödel numbering to express this) and see that the halting problem is decidable, i.e. the l_h object has appeared at the end of computation. We have a diagram like the following one:



In region 1, let us consider for the rule $a \rightarrow bc$ that its application time is strictly grater than 2 and let any execution times for the rest of the rules. In this case, if the system produces the object l_h (which is the halting label of the simulated register machine), then it will stop by producing the y symbol, otherwise the system will produce the n symbol. Hence, the halting problem for Π is decidable.

Since, the set Q is dense in \mathbb{R} , that is between any two real numbers there exists a rational number, then we have the following results:

$$PsCOP_2(cat_1, proR_1, Q^+) = PsCOP_2(cat_1, proR_1, \mathbb{R}^+) \supset PsRE \quad \square$$

Every theorem presented above can be reformulated in terms of P systems with inhibitors at the level of rules and the results are maintained. Without entering in the details of the proofs we only mention the following results:

Theorem 4.1.4 $PsCOP_2^c(cat_1, inhR_1, \mathbb{N}) = PsRE$.

Corollary 4.1.3 $PsTOP_2^t(cat_1, inhR_1) = PsRE$.

Theorem 4.1.5 $PsCOP_2(cat_1, inhR_1, \mathbb{Q}^+) = PsCOP_2(cat_1, inhR_1, \mathbb{R}^+) \supset PsRE$.

4.2 Modeling the Dynamical Parallelism of Bio-Systems

In nature, we often find biological systems (but not only) that are not necessarily homogeneous, consisting of many interacting entities that have a certain physical spatial distribution. This fact suggests that even if these entities interact in a parallel manner, they obey to some local conditions (concentration, for instance) and therefore the interaction parallelism cannot be considered as maximal or fixed, but as a variable that depends on the state of the system.

For example, at the cell level, biomolecular mechanisms are the result of many different chemical reactions that take place with a certain degree of mutual independence but such that they finally (and amazingly) exhibit an overall co-ordination. Traditionally, these behaviors were modeled using the theory of partial differential equations and nonlinear dynamical systems. However, this approach usually gave the general evolution and the dynamics of the system but not always giving the exact solution. From the discrete point of view, the interest was mainly in inferring the properties of the evolutions paths of such bio-inspired models. Although discrete models of complex phenomena may generate errors, the magnitude of the errors can be arbitrarily reduced by considering a better granularity of the phenomenon. This approach leads in general to a high computational effort, so a more efficient way of studying properties of bio-systems might be to consider discrete formal systems that have embedded into their formal description a certain degree of randomness that describes the way systems evolve.

One such property regards the measure of parallelism. From this point of view, using biochemical reasoning, one might predict for instance that given a particular state of a bio-system and the rules that make it evolves, an approximate next state is reached after a particular time. Basically, even if one does not know the exact number of times the rules are applied, one knows that after a particular time the reactions that had the potential to be applied, were actually accomplished in an approximate rate with respect to the state of the system.

Moreover, from the computer science point of view, in case we are trying to make use of bio-systems as computational devices we should be able to control their behaviors no matter the rate of parallelism occurring within them. Therefore, we are interested in systems that one might call “parallel fault tolerant” meaning that they produce the same output no matter which is the “evolution” of the parallelism. This assumption might also have a biological counterpart, namely natural sub-systems are able to regulate themselves and replace, in case is needed, the functions of other sub-systems such that the overall system can realize the same task. From this point of view one can assume that a complex bio-system (like a cell or whatever organism) has the ability to reach a “desired” state, no matter how local decisions were made.

Here we will consider two bio-mimetic models, namely Lindenmayer systems, inspired by the development of multi-cellular organisms and presented in Chapter 2, and P systems with promoters/inhibitors, inspired by the enzyme activation/deactivation process, occurring in the cells. We will start with a generalization of the classical Lindenmayer systems and we will show how one can use the results within P systems framework.

4.2.1 On the Dynamical Parallelism of L Systems

In this section we extend the classical definitions of Lindenmayer systems in order to fit a more general perspective. Such systems model biological developments in which parts of organism change simultaneously but not in the total parallel manner as in the classical Lindenmayer theory, but with respect to the current state of the organism. Several results regarding the computational power of these systems are also presented.

Generally speaking, computing formal systems make usually use of rewriting rules to perform their computations. The semantics of such formal models give the ways the rules are applied. Here, in order to capture the most general case, we will consider computable functions that, depending on the current state of the system, control the applications of the rules. This assertion can be more intuitively understood if we express this mathematical formalism by means of a biological motivation. More specifically, for a given state of a bio-system (represented by a multiset w), one can predict that a certain rule, say $a \rightarrow \alpha$, is to be applied on the multiset w in a rate specified by a value in the interval $(x, y) \subseteq (0, 1)$. Therefore, in that computational step, the rule $a \rightarrow \alpha$ is applied a number of times i , such that $[x \cdot |w|_a] \leq i \leq [y \cdot |w|_a]$. However, when generalizing this concept, we might assume

that there are more than one interval that control the applications of rules (even more, generalizing, we actually have more than one set), and this brings us to the following formalism.

Definition 4.2.1 *An M -rate 0L system, denoted by $M0L$, is a quadruple $H = (V, R, \omega, f)$, where $V = \{a_1, \dots, a_m\}$ is a finite alphabet, R is a finite set of productions of the form $i : (a \rightarrow \alpha)$, $a \in V$, $\alpha \in V^*$, f is a computable multi-valued function $f : V^* \rightarrow \mathcal{P}(R^*)$, $f(x) \in \mathcal{P}(R_x^{SAP})$, $i \in \mathbb{N}$, and $\omega \in V^*$ is the axiom. The set of productions R has to be complete, i.e., for each symbol $a \in V$ there must exist at least one production $i : (a \rightarrow \alpha) \in R$ with this letter a on the left side.*

$M0L$ systems use M -rate parallel derivations, i.e., x directly derives y in a $M0L$ system $H = (V, R, \omega, f)$, with $x, y \in V^*$, written as $x \xrightarrow{w} y$ if $x = x_1 x_2 \dots x_n$, $y = y_1 y_2 \dots y_n$, $x_i \in V$, $y_i \in V^*$, $1 \leq i \leq n$, and the following conditions hold:

- for every j , $1 \leq j \leq n$, either $y_j = x_j$ (the j -th symbol remains unchanged), or $r : (x_j \rightarrow y_j) \in P$ (some production of R is applied to the j -th symbol).
- the productions are applied according to $R_x^{sap} = r_1^{t_1} r_2^{t_2} \dots r_k^{t_k} \in f(x)$ (i.e., the production r_i is applied t_i times, $1 \leq i \leq k$).

This manner of derivation is called the “weak mode”. In the “strong mode”, $f : V^* \rightarrow \mathcal{P}(R^*)$, $f(x) \in \{X \subseteq R_w^{MSAP} \mid Pr(X) = Pr(R_w^{MSAP})\}$.

For both derivation modes, the *local degree of parallelism* for a given derivation step is defined by

$$\max_{y \in f(x)} (\text{card}(\text{supp}(y))).$$

Remark 4.2.1 *For the weak mode of derivation, a degree of parallelism k , $1 \leq k \leq 2$, means that at most two distinct productions are applied simultaneously. The number of times each production is applied is given by the computable multi-valued function f .*

Definition 4.2.2 *An M -rate T0L system, denoted by $MT0L$, is a triplet $H = (V, T, \omega)$, where V is a finite alphabet, $T = \{(T_1, f_1), \dots, (T_k, f_k)\}$ is a finite set of pairs, where each T_i , $1 \leq i \leq k$, is a complete set of context-free productions over V , and each f_i , $1 \leq i \leq k$ is a computable multi-valued function $f_i : V^* \rightarrow \mathcal{P}(T_i^*)$, $f_i(x) \in \mathcal{P}((T_i)_x^{SAP})$ (for the weak mode, or $f_i : V^* \rightarrow \mathcal{P}(T_i^*)$, $f_i(x) \in \{X \subseteq (T_i)_w^{MSAP} \mid Pr(X) = Pr((T_i)_w^{MSAP})\}$, $1 \leq i \leq k$ for the strong mode), and $\omega \in V^*$*

is the axiom. We say that x directly derives y in a MT0L system $H = (V, T, \omega)$, with $x, y \in V^*$, written as $x \xrightarrow{MTOL}_H y$, if $x \xrightarrow{MOL}_{H_i} y$ for some i , $1 \leq i \leq k$, with the M0L system $H_i = (V, T_i, \omega, F_i)$.

Definition 4.2.3 An M -rate ET0L system, denoted by MET0L, is a quadruple $H = (V, T, \omega, \Delta)$, where $\bar{H} = (V, T, \omega)$ is an MT0L system, and $\Delta \subseteq V$, $\Delta \neq \emptyset$, is the terminal alphabet. In an MET0L system $H = (V, T, \omega, \Delta)$, x directly derives y , with $x, y \in V^*$, written as $x \xrightarrow{METOL}_H y$, if $x \xrightarrow{MTOL}_{\bar{H}} y$. The transitive and reflexive closure of \xrightarrow{METOL}_H is denoted by \xrightarrow{METOL}^*_H . The generated language of the MET0L system H (denoted by $L(H)$) is $L(H) = \{w \in \Delta^* \mid \omega \xrightarrow{METOL}^*_H w\}$.

Definition 4.2.4 An M0L (or MT0L, MET0L) system generating the same language independently of the functions associated with the set(s) of productions is called parallel-free M0L (or MT0L, MET0L, respectively) system.

The families of languages generated by M0L (or MT0L, MET0L) systems working in the strong mode are denoted by $M0L^s$ ($MT0L^s$, $MET0L^s$, respectively).

The families of languages generated by M0L (MT0L, MET0L, respectively) systems working in the weak mode are denoted by $M0L^w$ ($MT0L^w$, $MET0L^w$, respectively).

When we speak about above mentioned families of languages, we will denote the parallel-free property by adding the superscript *pf*.

We denote by $U = \{L \mid |L| = 1\}$ the family of all singleton languages.

The following result characterizes the computational power of extended, interactionless Lindenmayer systems when working in the weak mode and being parallel-free.

Theorem 4.2.1 $ME0L^{w,pf} = MET0L^{w,pf} = U \cup \{\emptyset\}$.

Proof. The assertion is trivial, because a system working in parallel-free mode means that whatever set/sets of multi-valued functions one can choose, the system produces the same output; in addition, because the system works in a weak mode then it means that certain productions might not be applied at all even if there are symbols (but not enough) that are within the scope of them. So, one can choose the multi-valued functions in such a manner that the productions cannot be applied at all. Therefore, such systems generate languages containing at most the systems axiom. \square

Using a similar argument one can prove that:

Theorem 4.2.2 $MT0L^{w.pf} = M0L^{w.pf} = U$.

Example 4.2.1 Let $H = (V, P, \omega, F)$ be an M0L system working in the strong mode such that:

$$\begin{aligned} V &= \{a, b, c\}, \\ P &= \{r_1 : a \rightarrow aa, r_2 : b \rightarrow bb, r_3 : c \rightarrow cc\}, \\ \omega &= abc, \\ f(w) &= \{r_1 r_2 r_3\}, \text{ for } w \in V^*. \end{aligned}$$

The system H generates the language $L_H = \{a^n b^n c^n \mid n \geq 1\}$.

Here we will prove that extended, interactionless Lindenmayer systems can generate any language over a given alphabet V .

Theorem 4.2.3 $ME0L^w = MET0L^w = \mathcal{P}(V^*)$.

Proof. Let us prove that using such systems we can generate any language L (computable or not). For the sake of simplicity but without losing the generality we will generate languages over one letter alphabet. For a given $L \subseteq \{a\}^+$, we construct the system $H = (V, P, \omega, \Delta, F)$ where $V = \{a, A\}$, $\omega = A$, $\Delta = \{a\}$, and the set P contains the following productions:

$$\begin{aligned} r_1 : A &\rightarrow aA, \\ r_2 : A &\rightarrow \lambda, \\ r_3 : a &\rightarrow a. \end{aligned}$$

and f is defined as follows.

For a given string $w = a^n A$, $n \geq 0$, we have:

- $f(w) \subseteq \{r_1 r_3^i \mid 0 \leq i \leq n\}$ iff $a^n \notin L$,
- $f(w) \subseteq P_w^{SAP}$ and there exists $z \in \{r_2 r_3^i \mid 0 \leq i \leq n\}$ such that $z \in f(w)$ iff $a^n \in L$.

Observe that any time $a^n \notin L$, the number of symbols a grows (the production $r_1 : A \rightarrow aA$ is executed since there exists $z \in \{r_1 r_3^i \mid 0 \leq i \leq n\}$, such that $z \in f(a^n A)$).

In case $a^n \in L$ then f indicates that the production $r_2 : A \rightarrow \lambda$ might be executed (because of the way the function f was defined, also the production $r_1 : A \rightarrow aA$ might be executed). Therefore, non-deterministically, the system H generates a string $w \in L$.

In this way, the constructed system can generate any subset of V^* . Hence, we have that $ME0L^w = MET0L^w = \mathcal{P}(V^*)$. \square

Example 4.2.2 *The language $L = \{a, a^3\}$ is not $MOL^{s,pf}$ (or $MTOL^{s,pf}$) language. This is proved by contradiction as follows. If there exists a $MOL^{s,pf}$ system $H = (V, P, \omega)$ such that $L(H) = \{a, a^3\}$ then, since obviously $V = \{a\}$, we have two cases: (i) $\omega = a$ and $a \Rightarrow a^3$, hence $a^3 \Rightarrow a^k$, $k \neq 1, 3$, therefore a contradiction; (ii) $w = a^3$, hence $a \Rightarrow a$ and $a \Rightarrow \lambda$. Thus $a^3 \Rightarrow a^2$ and so $a^2 \in L(G)$, therefore a contradiction.*

Obviously, the following results stand:

Proposition 4.2.1 $MTOL^{s,pf} \subset RE$.

Proposition 4.2.2 $MOL^{s,pf} \subset RE$.

Now, we will prove that there exists a class of ETOL systems that are independent of the multi-valued functions associated with the sets of rules and which are able to generate the whole class of ETOL languages.

Theorem 4.2.4 $METOL^{s,pf} = ETOL$.

Proof. We prove this result by double inclusion.

(1) $METOL^{s,pf} \supseteq ETOL$.

Consider an ETOL system $\tilde{H} = (\tilde{V}, \tilde{T}, \tilde{\omega}, \tilde{\Delta})$ such that $\tilde{T} = \{\tilde{T}_1, \tilde{T}_2\}$.

Let $h_1 : \tilde{V}^* \rightarrow \bar{V}^*$ be a morphism such that $h_1(a) = \bar{a}$, $a \in \tilde{V}$. Also, let $h_2 : \tilde{V}^* \rightarrow \bar{\bar{V}}^*$ be a morphism such that $h_2(a) = \bar{\bar{a}}$, $a \in \tilde{V}$.

We will simulate the computation of the system \tilde{H} with an METOL system $H = (V, T, \omega, \Delta)$ defined as follows.

- $V = \tilde{V} \cup \{h_1(A), h_2(A) \mid A \in \tilde{V}\} \cup \{t_1, t_2, t_3, t_4\} \cup \{\#\}$;
- $T = \{(T_1, f_1), (T_2, f_2), (T_3, f_3), (T_4, f_4)\}$, where

$$\begin{aligned}
T_1 &= \{(A \rightarrow h_1(\alpha)t_1) \text{ for all } A \rightarrow \alpha \in \tilde{T}_1\} \\
&\cup \{(h_1(A) \rightarrow h_1(A)) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_2(A) \rightarrow \#) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(\# \rightarrow \#), (t_1 \rightarrow \lambda), (t_2 \rightarrow \#), (t_3 \rightarrow \#), (t_4 \rightarrow \#)\}, \\
T_2 &= \{(A \rightarrow A) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_1(A) \rightarrow At_2) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_2(A) \rightarrow \#) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(\# \rightarrow \#), (t_1 \rightarrow \#), (t_2 \rightarrow \lambda), (t_3 \rightarrow \#), (t_4 \rightarrow \#)\},
\end{aligned}$$

$$\begin{aligned}
T_3 &= \{(A \rightarrow h_2(\alpha)t_3) \text{ for all } A \rightarrow \alpha \in \tilde{T}_2\} \\
&\cup \{(h_1(A) \rightarrow \#) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_2(A) \rightarrow h_2(A)) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(\# \rightarrow \#), (t_1 \rightarrow \#), (t_2 \rightarrow \#), (t_3 \rightarrow \lambda), (t_4 \rightarrow \#)\}, \\
T_4 &= \{(A \rightarrow A) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_2(A) \rightarrow At_4) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(h_1(A) \rightarrow \#) \text{ for all } A \in \tilde{V}\} \\
&\cup \{(\# \rightarrow \#), (t_1 \rightarrow \#), (t_2 \rightarrow \#), (t_3 \rightarrow \#), (t_4 \rightarrow \lambda)\};
\end{aligned}$$

and $f_i : V^* \rightarrow \mathcal{P}(R^*)$, $f_i(x) \in \{X \subseteq R_w^{MSAP} \mid Pr(X) = Pr(R_w^{MSAP})\}$, $1 \leq i \leq 4$, arbitrarily.

- $\omega = \tilde{\omega}$;
- $\Delta = \tilde{\Delta}$.

Here is how the construction is done. We want to simulate the applications of \tilde{H} tables; to this aim let us assume, without loosing the generality, that \tilde{T}_1 is simulated first. So, in H , at the beginning, one table is chosen nondeterministically and is applied on the initial sentential form ω ; in case table T_2 or T_4 is chosen, then the current sentential form is left unchanged (only the productions of type $A \rightarrow A$, $A \in V$ are applied). If table T_1 (or T_3) is chosen then productions of type $A \rightarrow h_1(\alpha)t_1$ will be executed. However because the parallelism is not necessarily maximal but depends on the function f_1 (or f_3 , respectively), then not necessarily all symbols A from the current sentential form will be rewritten. However, since in table T_1 there exists only one production having the symbol A on the left-hand side (i.e., $A \rightarrow h_1(\alpha)t_1$) and the system is working in the strong mode, then the symbol A will be changed at least once (remember that $f_i(x) \in \{X \subseteq R_w^{MSAP} \mid Pr(X) = Pr(R_w^{MSAP})\}$). Consequently, a symbol t_1 is produced. Assume now that there are still symbols A not rewritten by T_1 despite the existence of a production handling symbol A . If this is the case, observe that if any other table (except T_1) is chosen for a next application, then the symbol $\#$ is generated (henceforth the system will not be able anymore to generate a string over Δ because the production $\# \rightarrow \#$ is present in every table of H and will be always executed). Therefore, the only table that can be applied and that does not produce the symbol $\#$ is again T_1 . Finally, all symbols $A \in V$ will be rewritten by table T_1 . In addition, table T_1 is also responsible for deleting all symbols t_1 . After completing these tasks the sentential form will have only images by morphism h_1 of symbols from V (let us

call them “overlined” symbols). At that moment, if we choose to apply any other table except T_2 the symbol $\#$ is again produced. In case table T_2 is applied, then with a similar mechanism as presented before, the system checks whether or not all “overlined” symbols are rewritten into “regular” ones. Again, during this checking procedure if we choose to apply another table other than T_2 , symbol $\#$ is produced. The simulation of the application of table \tilde{T}_2 follows a similar pattern. In this way, the computation take place step by step, simulating the computation of \tilde{H} if the “right” tables are chosen, and always producing the trap symbol if a “wrong” table is chosen.

Observe that the strong working mode feature is essential because if at a certain moment a wrong table is chosen for application, then we have to be sure that at least one symbol $\#$ is produced, hence a production has to be applied at least once if it can be applied.

Concluding, we have that the constructed system H generates the same language as the arbitrarily considered ETOL system \tilde{H} . Consequently, we have that $METOL^{s,pf} \supseteq ETOL$.

(2) $ETOL \supseteq METOL^{s,pf}$.

In order to prove this inclusion, we will simulate the computation of an arbitrary $METOL^{s,pf}$ system $\bar{H} = (\bar{V}, \bar{T}, \bar{\omega}, \bar{\Delta})$ with an ETOL system $H = (V, T, \omega, \Delta)$ we construct. First, remark that because the system \bar{H} is parallel-free, then, no matter how the multi-valued functions associated with the sets of productions are chosen, the result of the computation is the same. In particular, one can associate with all the sets of productions of the system the multi-valued functions such that, during the computation, the productions are applied in a total parallel manner (as for L systems).

Therefore, H is defined as follows:

- $V = \bar{V}$;
- $T = \{T_1, T_2, \dots, T_k\}$ providing that $\bar{T} = \{\bar{T}_1, \bar{T}_2, \dots, \bar{T}_k\}$;
- $\omega = \bar{\omega}$;
- $T_i = \{A \rightarrow \alpha \mid A \rightarrow \alpha \in \bar{T}_i, 1 \leq i \leq k\}$;
- $\Delta = \bar{\Delta}$.

Observe that in an ETOL system, when a certain table is applied, if a production can be applied then it will be applied (of course, respecting the non-determinism if exists). This corresponds to the strong mode of derivation for METOL systems.

Consequently, we have that $ETOL \supseteq METOL^{s,pf}$ and therefore we can conclude that $METOL^{s,pf} = ETOL$. □

4.2.2 On the Dynamical Parallelism of P Systems

In this section we will extend the definition of dynamical parallelism to P systems. As in previous chapters we will focus only on the particular model motivated by the cell enzyme activation/deactivation mechanism, namely P systems with promoters/inhibitors. However, for the sake of generality we will consider P systems with cooperative rules. Similarly as defined for MOL systems in Section 4.2.1 we can define the *M-strong rate* and the *M-weak rate* modes of derivation for P systems.

Consider a P system $\Pi = (V, C, P, I, \mu, w_1, \dots, w_n, R_1, \dots, R_n, i_0)$ defined using the standard notation. Let $f_i : V^* \rightarrow \mathcal{P}(R^*)$, $f_i(x) \in \mathcal{P}(R_x^{SAP})$, $1 \leq i \leq n$, be computable multi-valued functions

Consider now the system

$$\Pi(f_1, \dots, f_n) = (V, C, P, I, \mu, w_1, \dots, w_n, R_1, \dots, R_n, i_0, f_1, \dots, f_n)$$

and let us describe how the computations are performed.

Starting from the initial configuration represented by the n-tuple (w_1, \dots, w_n) , w_i , $1 \leq i \leq n$ multisets over V , the system evolves according to the rules and objects present in the membranes, in a non-deterministic parallel manner. The selection of the rules as well as the number of applications of the selected rules on the multiset w_i , $1 \leq i \leq n$ are given by a computable function f_i $1 \leq i \leq n$ (we have that $f_i(w_i) \in \mathcal{P}((R_i)_{w_i}^{SAP})$ in case of weak mode derivation, or $f_i(w_i) \in \{X \subseteq R_w^{MSAP} \mid Pr(X) = Pr(R_w^{MSAP})\}$ for the strong mode derivation). As usually, the system computes according with a universal clock.

For a given configuration (x_1, x_2, \dots, x_n) with x_i , $1 \leq i \leq n$ multisets over V , the rules are applied according with $(R_i)_{x_i}^{sap} = r_{(i,1)}^{t_1} r_{(i,2)}^{t_2} \dots r_{(i,k)}^{t_k} \in f_i(x_i)$, $1 \leq i \leq n$ (i.e., given a multiset x_i , the rule $r_{(i,j)}$ is applied t_j times $1 \leq j \leq k$).

For two configurations $C_1 = w'$ and $C_2 = w''$ of Π , we can define the transition from C_1 to C_2 if we can pass from C_1 to C_2 by using the evolution rules from R_i , $1 \leq i \leq n$ applied according with the functions f_i , $1 \leq i \leq n$.

As usual, a *computation* of a P system Π is a sequence of transitions between configurations. The system will make a successful computation if and only if it halts. In case of generative P systems, the result of a successful computation is the number (or the vector of numbers) of objects present in the membrane, in a halting configuration of Π . If the computation never halts, then we will have no output.

A P system Π is *parallel-free* if and only if for any set of functions f_i , $1 \leq i \leq n$, the system Π produces the same set of vectors of natural numbers.

We use the notation:

$$PsOP_m^{\alpha,\beta}(\gamma, \theta)$$

where $\alpha \in \{s, w\}$, $\beta \in \{pf\}$, $\gamma \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}$, $\theta \in \{proR_i, inhR_i \mid i \geq 0\}$ to denote the family of sets of vectors of natural numbers generated by P systems with strong mode derivations ($\alpha = s$), or with weak mode derivations ($\alpha = w$) respectively, having the parallel free property ($\beta = pf$), having at most m membranes, evolution rules that can be non-cooperative ($ncoo$), cooperative (coo), or catalytic (cat_k), using at most k catalysts, and promoters ($proR_i$) or inhibitors ($inhR_i$) of weight i , at the level of rules.

Let us examine the following example:

Example 4.2.3 Consider the P system $\Pi = (V, C, P, I, \mu, w, R, \vartheta, f)$ defined as follows:

$$\begin{aligned} V &= \{a, p\}; \\ C &= \emptyset; \\ P &= \{p\}; \\ I &= \emptyset; \\ \mu &= []_1; \\ w &= a^2p; \\ R &= \{r_1 : a \rightarrow aaa|_p, r_2 : p \rightarrow p, r_3 : p \rightarrow \lambda\}; \\ \vartheta &= 1; \end{aligned}$$

$$f(w) = \{r_1^{[0.5*|w|_a]+1} r_2^i r_3^j \mid i, j \in \{0, 1\}, i + j = 1\}, w \in V^*.$$

Let us see how this system works. Observe first that the execution of rule r_1 is controlled by the promoter p , while the number of applications of r_1 on the current multiset is given by the function f . Assume that the rule $r_1 : a \rightarrow aaa|_p$ is executed k times in k derivation steps. Then we have:

$$\begin{aligned} |w|_a = |w_1|_a &= 2; \\ |w_2|_a &= 6; \\ |w_3|_a &= 14; \\ &\dots \dots \dots \\ |w_k|_a &= ((|w_{k-1}|_a * 0.5) + 1) * card(right(r_1)) \\ &+ |w_{k-1}|_a - ((|w_{k-1}|_a * 0.5) + 1) \\ &= 2 * (1 + [|w_{k-1}|_a * 0.5]) + |w_{k-1}|_a \end{aligned}$$

Observe that if $|w_1|_a : 2 \Rightarrow |w_k|_a : 2$, then this means that $[|w_i|_a * 0.5] = |w_i|_a * 0.5$, $1 \leq i \leq k$. We have the following recurrent formulas.

$$\begin{array}{l|l} |w_k|_a = 2 * |w_{k-1}|_a + 2 & *2^0 \\ |w_{k-1}|_a = 2 * |w_{k-2}|_a + 2 & *2^1 \\ \dots\dots\dots & \dots\dots\dots \\ |w_2|_a = 2 * |w_1|_a + 2 & *2^{k-2} \end{array}$$

In order to obtain the general term $|w_k|_a$ we will multiply each recurrent formula by a corresponding constant (as shown above) and we will sum the results. Then, we have:

$$|w_k|_a = 2^{k-1} * |w_1|_a + 2^1 + 2^2 + \dots + 2^{k-1} = 2 + \dots + 2^k = \frac{2*(2^k-1)}{2-1} = 2^{k+1} - 2.$$

This means that Π generates the set $\{2^{k+1} - 2 \mid k \geq 2\}$.

Now we will give several results regarding P systems with symbol-objects and promoters/inhibitors at the level of the rules. In a certain sense, we will show that the original definition of P systems can be relaxed in the following respect: for each class of P systems with cooperative or non-cooperative promoted/inhibited rules we will find a proper class of P systems, with the same features and generating the same class of Parikh images of languages, but working in the parallel-free manner.

Theorem 4.2.5 $PsOP_m^{s,pf}(ncoo, inhR_1) = PsET0L$.

Proof. In Chapter 3 it is proved that $PsOP_1(ncoo, inhR_1) = PsET0L$. Moreover, it is easy to see that $PsOP_1^{s,pf}(ncoo, inhR_1) = MET0L^{s,pf}$. Since in Theorem 4.2.4 we have shown that $MET0L^{s,pf} = ET0L$, we have that $PsET0L = PsOP_1^{s,pf}(ncoo, inhR_1)$. \square

Theorem 4.2.6 $PsOP_2^{s,pf}(cat_1, proR_1) = PsRE$.

Proof. In Chapter 3 was shown that $DPsIOP_2(cat_1, proR_1) = PsRE$. There the inclusion $DPsIOP_2(cat_1, proR_1) \supseteq PsRE$ was proved by simulating a deterministic register machine in a deterministic manner. The role of the catalyst was to sequentialize when needed the behavior of the P system. Basically, at each time during the computation, one or more rules were executed, but only once at a time (and not maximally rewriting all occurrences of a given object by the same rule).

Therefore, using exactly the same construction as in Theorem 3.3.3, we have that $PsOP_2^{s,pf}(cat_1, proR_1) = PsRE$. \square

In Chapter 3 it was proved that $DPsIOP_2(cat_1, inhR_1) \supseteq PsRE$ by simulating a deterministic register machine in a deterministic manner; the same argument can be invoked to show that:

Theorem 4.2.7 $PsOP_2^{s,pf}(cat_1, inhR_1) = PsRE$.

4.3 Open Problems and Forthcoming Research

Several open problems and research proposals can be easily distinguished for time-free P systems. As a general suggestion, it would be interesting to see which are the results known for standard P systems that remain true also for time-free P systems. For instance, as already mentioned, it would be interesting to know if time-free catalytic P systems are universal. In this respect we have conjectured that the answer to this question is negative. Such an answer would imply that for the class of catalytic P systems *time is important* to get universality.

Another interesting way to investigate time-free P systems is to impose certain conditions on the time of execution of the rules (this class of systems has been refereed as *partially time-free P systems* in [21]). How to formalize and to use such conditions? Which are the conditions that should be used (that means, “realistic”)? These are only two of the many questions concerning this class.

It is true that even using only one catalyst is possible to construct non time-free P systems. A rather natural question is the following one: given an arbitrary system is it possible to decide if it is time-free? This problem has been addressed in [19] where it has been shown that it is undecidable if a P system using bi-stable catalysts is time-free. It would be useful to find a class C of P systems with enough computational power, with two more features: given any system in C there exists an equivalent time-free system in C , producing the same output or accepting the same input; given an arbitrary system Π in C it is possible to decide (in an easy way) if Π is time-free. The class of catalytic P systems seems to be a good candidate.

Moreover, something that can be further investigated concerns new methods to synchronize the computation in time-free P systems. We have mainly used signal-promoters; are there other synchronizing devices (maybe inspired from cell functioning)? This work might also have impact on the more general field of asynchronous parallel computing.

It is also possible to add other parameters to construct a “more realistic” P system; for instance, it would be very interesting to associate to each rule a time of delay that indicates the time to wait before a rule is started and then to study

a class of delay-free systems. This might model the fact that, sometimes, chemical rules are not started immediately, even in the presence of the necessary chemicals.

Chapter 5

An Algorithmic Overview

In many applications P systems have considerable advantages. Among them we mention their inherently discrete nature, parallelism, transparency, scalability, nondeterminism, and especially their ability and expressivity to model phenomena.

In this chapter we study two applications of membrane computing, namely *static sorting* and *Boolean circuits simulations*.

Sorting numbers represents one of the most studied problems in computer science. Section 5.1 covers a range of implementations for the sorting problem using P systems with promoters and inhibitors.

Boolean circuits embody the notion of massively parallel signal processing and are usually encountered in many parallel algorithms. Many important problems such as sorting, integer arithmetic, and matrix multiplication are known to be computable by small size Boolean circuits much faster than by ordinary sequential digital computers. Section 5.2 shows that membrane computing allows one to simulate such Boolean circuits in a polynomial time.

5.1 Sorting with P Systems

This section deals with the application of P systems to sorting problems. Traditional studies of sorting assume a constant time for comparing two numbers and compute the time complexity with respect to the number of components of a vector to be sorted. Here we assume the number of components to be a fixed number k , and study various algorithms with distinct generative features based on P systems with promoters/inhibitors, and their time complexity. Massively parallel computations that can be realized within the framework of P systems offer a premise to major improvements of the classical integer sorting problems.

Computing an (ordered) word from an (unordered) multiset can be a goal not

only for computer science, but also, at least, for bio-synthesis (separating mixed objects according to some characteristics).

5.1.1 Sorting Preliminaries

The sorting problem is an important problem in computer science. For it, many, both sequential and parallel, algorithms were developed but the time complexity remains at least $O(n \log(n))$ for the sequential case and $O(\log^2 n)$ for the parallel case.

Studying sorting within P systems framework is also a challenging task not only because it can produce better results (in some respect) than in the classical sequential case, but also, because we can compute an (ordered) string from an (unordered) multiset of objects. Moreover, one can remark that in the case of cooperative rules (in P systems with objects rewriting framework) the order of symbols in the left/right hand side of a production does not count. So, we deal with two “types of unordered” and still can compute an “order”.

One of the first approaches of sorting with P systems was done in [8] by considering a bead sort algorithm. There, the sorting procedure was constructed on a tissue P system with symport-antiport rules such that the objects were considered beads and the membranes were considered rods. The idea of the algorithm was that beads start to slide in the membrane structure to their appropriate places. The time complexity for this case was linear, which constitutes an improvement of the classical sequential sorting algorithm. However, the payoff for this approach was that it requires a number of membranes proportional to $m \times k$, where m is the maximal number from the vector that we would like to sort, and k is the dimension of the vector.

Another study on this topic was done in [25], where the P systems with inhibitors/promoters and symport/antiport rules were used to develop comparators and then to organize them in a sorting network. As in the previous case, the input data was placed in different membranes and the computation starts operating on already dissociated elements. Also, the result was not obtained in a halting configuration but in a stable one, meaning that there are still rules applicable, but their application does not change the contents of the membranes, nor the membrane structure itself. The time complexity for the algorithms presented was also linear with respect to the number of components. The number of membranes used in the computation was also proportional to the number of components.

In this work we propose other methods and algorithms for the sorting problem by considering P systems with promoters/inhibitors. The feature shared by the algorithms presented is that the input components will be placed in an initial input membrane and the computation will dissociate this input according to the relation order between the multiplicities of components. In this way, we will interpret the sorting as the order of elimination of the objects. The idea behind the algorithms presented will be to consume objects from all components at once and, when one component is exhausted, to trigger a signal meaning that we find the next component that must be eliminated.

Let $V = \{\underline{i} \mid 1 \leq i \leq k\}$ be an alphabet. A word over V is denoted by

$$w = \prod_{j=1}^m a_j = a_1 a_2 \cdots a_m,$$

$m \in \mathbb{N}$, where $a_j \in V$ for each $1 \leq j \leq m$. Here, the symbol of product represents the concatenation.

Example 5.1.1 $w = \underline{23} \underline{9} \underline{157}$.

The reason why we denote the alphabet symbols by underlined numbers is that in this way we can distinguish the symbols and also have the implicit order associated with natural numbers.

Let also $ord : V \rightarrow \{1, \dots, k\}$ be a bijective function such that $i = ord(\underline{i})$, $1 \leq i \leq k$. Then $i_j = ord(a_j)$ is an ordinal number of the j -th letter of w and $a_j = \underline{i_j}$.

Let $v = \prod_{j=1}^k \underline{j}$ be the “alphabet word”, made by concatenating elements of the alphabet V in alphabetical order.

Since in the multiset processing we represent multisets by strings, we will need a few formal language definitions and notations.

Let $w = \prod_{j=1}^m a_j$ be a string. We denote by

$$Perm(w) = \left\{ \prod_{j=1}^m a_{i_j} \mid 1 \leq i_j \leq m, 1 \leq j \leq m; i_j \neq i_l, 1 \leq j, l \leq m \right\}$$

the set of *permutations* of string w , i.e., the set of all strings that can be obtained from w by changing the order of symbols. We denote by

$$SSub(w) = \left\{ \prod_{j=1}^l a_{i_j} \mid 1 \leq i_{j-1} < i_j \leq m, 2 \leq j \leq l; 0 \leq l \leq m \right\}$$

the set of *scattered subwords* of w , i.e., the set of all strings that can be obtained from w by deleting some (0 or more, possibly all) of its symbols, and concatenating the remaining ones, preserving the order.

For example, the permutations of the *alphabet word* v are the strings having exactly one occurrence of each letter of V , in arbitrary order. Also, the scattered subwords of v are the strings, consisting of some of the letters of V , in alphabetic order.

Now, let us go back to the initial scope of this section – the sorting. According to the classical definitions of what integer sorting means we can give an “equivalent” definition adapted to the P systems. In this framework we can sort only the numbers represented by the multiplicities of the objects but not considering the corresponding objects, or we can consider both characteristics. Therefore we can define:

Definition 5.1.1 *Let $v = \prod_{j=1}^k \underline{j}$ be the alphabet word. The word $w = \prod_{j=1}^k a_j \in Perm(v)$, $k = card(V) \in \mathbb{N}^+$, where $a_j \in V$ such that $M(a_j) \leq M(a_{j+1})$, for each $1 \leq j \leq k - 1$, is called the ranking string of the multiset M .*

Definition 5.1.2 *The word $w = \prod_{j=1}^k \underline{j}^{M(a_j)}$ is called weak sorting string of the multiset M if $\prod_{j=1}^k a_j$ is the ranking string of M . Also, $M' : V \rightarrow \mathbb{N}$ defined as $M'(\underline{j}) = M(a_j)$ is the weak sorting multiset of M .*

Remark 5.1.1 *This definition stands for the case when we are interested in sorting the multiplicities of the objects and not having to look at the corresponding objects. In other words we sort only “properties” and not “objects and properties”. Practically, the symbols from the initial multiset are considered to be in a complete relation order, and, after performing the computation, we will obtain as the result these objects sorted according to the relation order and having multiplicities sorted.*

Definition 5.1.3 *The word $w = \prod_{j=1}^k a_j^{M(a_j)}$ is called strong sorting string of M if $\prod_{j=1}^k a_j$ is the ranking string of M .*

Remark 5.1.2 *In this case, we are interested to have as the output of the computation the objects with the same associated multiplicities, present in a string in the increasing order of their multiplicities.*

Example 5.1.2 *For the alphabet $V = \{\underline{1}, \underline{2}, \underline{3}\}$ and the multiset $M = \{(\underline{1}, 20), (\underline{2}, 10), (\underline{3}, 30)\}$, we have:*

- *ranking string : $\underline{2} \underline{1} \underline{3}$*
- *weak sorting string : $\underline{1}^{10} \underline{2}^{20} \underline{3}^{30}$*
- *strong sorting string : $\underline{2}^{10} \underline{1}^{20} \underline{3}^{30}$*

We will consider, in several cases, as the input of the system the multiset S with symbols in a set $U \subset V$, where V is the system's alphabet, such that in the initial configuration, S is present in one region and there are no other symbols from U present in other regions. In the present work, we will treat only the static strong sorting case (for more details regarding static weak sorting algorithms we indicate [3]).

5.1.2 Static Sorting with P Systems with Promoters/Inhibitors

We present an algorithm for integer sorting problem using promoters and cooperating rules. One can note that despite the fact that the “degree” of cooperation is high and we use other powerful ingredients (promoters) the problem is not trivial because we want to extract the order from the unordered. Moreover, the procedure must be, in a certain sense (as we will see later there are some algorithms for which the non-determinism exists but will not affect the computation), deterministic. First, we will give the general algorithm which will work for a number $k > 1$ of integers.

Algorithm 5.1.1 Let $v = \prod_{i=1}^k i$. Consider

$$\Pi = (O, [1]_1, p_k \prod_{j=1}^k \underline{j}^{n_j}, R_1, 0),$$

where

$$\begin{aligned} O &= \{\underline{i}, X_i \mid 1 \leq i \leq k\} \cup \{y_i^{(j)} \mid 1 \leq i, j \leq k\} \cup \{p_i \mid 0 \leq i \leq k\} \\ &\cup \{\langle w \rangle \mid w \in SSub(v) - \{v\}\}, \\ R &= \{w \rightarrow \prod_{j=1}^{|w|} x_j \langle \prod_{j=1, j \notin alph(w)}^k \underline{j} \rangle \Big|_{p_{|w|}} \mid w \in SSub(v) - \{\varepsilon\}\} \\ &\cup \{p_i \rightarrow p_{i-1} \mid 1 \leq i \leq k\} \cup \{x_j \rightarrow \underline{j}_{out} \mid 1 \leq j \leq k\} \cup \{y_j^l \rightarrow y_j^{l-1} \mid 2 \leq l \leq k\} \\ &\cup \{\prod_{j=1}^{k-1} \langle \prod_{l=1}^k \underline{i}_j \rangle \rightarrow \prod_{j=1}^k y_i^{(j)} \mid \prod_{j=1}^k \underline{i}_j \in Perm(v)\}. \end{aligned}$$

For a better understanding we present an example illustrating the behavior of the system. Let us consider the case of sorting 3 numbers and also, for simplicity, let us rename variables present in the general algorithm:

Example 5.1.3 $\underline{1} = a$, $X_1 = a'$, $y_1^{(j)} = A_j$, $\langle \underline{1} \rangle = \langle a \rangle$, $p_i = p_i$, etc.

Initial data: $a^{n_1} b^{n_2} c^{n_3} p_3$

The rules are:

$$\begin{aligned}
& p_3 \rightarrow p_2, & p_2 \rightarrow p_1, & p_1 \rightarrow p_0, & abc \rightarrow a'b'c'\langle\varepsilon\rangle|_{p_3}, \\
& ab \rightarrow a'b'\langle c\rangle|_{p_2}, & ac \rightarrow a'c'\langle b\rangle|_{p_2}, & bc \rightarrow b'c'\langle a\rangle|_{p_2}, \\
& a \rightarrow a'\langle bc\rangle|_{p_1}, & b \rightarrow b'\langle ac\rangle|_{p_1}, & c \rightarrow c'\langle ab\rangle|_{p_1}, \\
& \langle a\rangle\langle ab\rangle \rightarrow A_1B_2C_3, & \langle a\rangle\langle ac\rangle \rightarrow A_1C_2B_3, & \langle b\rangle\langle bc\rangle \rightarrow B_1C_2A_3, \\
& \langle b\rangle\langle ab\rangle \rightarrow B_1A_2C_3, & \langle c\rangle\langle ac\rangle \rightarrow C_1A_2B_3, & \langle c\rangle\langle bc\rangle \rightarrow C_1B_2A_3, \\
& A_3 \rightarrow A_2, A_2 \rightarrow A_1, & B_3 \rightarrow B_2, B_2 \rightarrow B_1, & C_3 \rightarrow C_2, C_2 \rightarrow C_1, \\
& a' \rightarrow a_{out}|_{A_1}, & b' \rightarrow b_{out}|_{B_1}, & c' \rightarrow c_{out}|_{C_1}.
\end{aligned}$$

Description of the model: The system starts with objects $a^{n_1}, b^{n_2}, c^{n_3}, p_3$ and let us suppose, without loss of generality, that we have $n_1 \geq n_2 \geq n_3$. First, in the presence of the promoter p_3 the rule $abc \rightarrow a'b'c'\langle\varepsilon\rangle|_{p_3}$ will be used. In this way we apply this rule in one step a number of times equal to the smallest multiplicity of the objects (n_3) and we will add in the membrane the objects $a^{n_3}, b^{n_3}, c^{n_3}, \langle\varepsilon\rangle^{n_3}$. In the same step, the promoter p_3 will be used by the rule $p_3 \rightarrow p_2$. In the second step of computation, the promoter p_2 , being present in the membrane, will let the rule $ab \rightarrow a'b'\langle c\rangle|_{p_2}$ to be applied for $n_2 - n_3$ times. As above, in the same time, the promoter p_2 will change to p_1 and so the configuration of the system for the region will contain the objects $a^{(n_3+(n_2-n_3))}, b^{(n_3+(n_2-n_3))}, c^{(n_3)}, \langle c\rangle^{(n_2-n_3)}, \langle\varepsilon\rangle^{n_3}, a^{(n_1-n_2)}, p_1$. In the third step, remaining objects a will be transformed into objects a' and $\langle bc\rangle$. Since the number of objects a which remained after the second step was $a^{(n_1-n_2)}$, then we will add to the membrane $n_1 - n_2$ copies of a' and the same number of $\langle bc\rangle$. The objects present in the membrane will be: $a^{n_1}, b^{(n_2)}, c^{(n_3)}, \langle\varepsilon\rangle^{n_3}, \langle c\rangle^{(n_2-n_3)}, \langle bc\rangle^{(n_1-n_2)}, p_0$. Now, there are only the objects $\langle c\rangle$ and $\langle bc\rangle$ that can react and so, the symbols $C_1B_2A_3$ will be produced. These symbols, with a corresponding “delay” (some renaming of the symbols) will be used (as promoters) to throw out the symbols in the right order.

Of course, the model being symmetrical in what concerns the rules, its behavior does not depend on the order between the initial multiplicities and will produce as output the objects in the right order.

Remark 5.1.3 *The time complexity for the strong sorting algorithm with P systems with promoters is $2k+1$ where k is the number of elements to be sorted. It is constant with respect to the values of the elements.*

Using inhibitors to solve the sorting problem is also a challenging aspect that can be discussed. By using this feature we can specify when the execution of some rules should not happen, and so, we can drive the production of objects in the right order. This feature of forbidding the execution of certain rules seems to decrease the “cooperativeness” of the rules compared to the algorithm using promoters. The inhibitors are very important only after we made the computation and have the solution at the corresponding ranking problem, but we did not sort actually. Then, by using inhibitors we can drive the elimination process in the right way. The algorithm proposed for the general case is as follows.

Algorithm 5.1.2 Let $v = \prod_{i=1}^k i$. Consider

$$\Pi = (O, [1]_1, \prod_{i=1}^{k-1} \prod_{j=2}^k \langle i, 2, j \rangle \prod_{j=1}^k (y_j^j I_j) \prod_{j=1}^k \underline{j}^{n_j}, R_1, 0),$$

where

$$\begin{aligned} O &= \{\underline{i}, I_i, y_i \mid 1 \leq i \leq k\} \cup \{x_i^{(j)} \mid 1 \leq i, j \leq k\} \\ &\cup \{\langle i, l, j \rangle \mid 1 \leq i < j \leq k, 0 \leq l \leq 2\}, \\ R &= \{\underline{i} \rightarrow \prod_{j=1}^k x_i^{(j)} \mid 1 \leq i \leq k\} \\ &\cup \{x_i^{(j)} x_j^{(i)} \rightarrow \varepsilon, x_i^j \langle i, 0, j \rangle \rightarrow y_i, x_j^i \langle i, 0, j \rangle \rightarrow y_i \mid 1 \leq i < j \leq k\} \\ &\cup \{\langle i, l, j \rangle \rightarrow \langle i, l-1, j \rangle \mid 1 \leq i < j \leq k, l \in \{1, 2\}\} \\ &\cup \{I_j y_j \rightarrow I_j, x_i^{(i)} \rightarrow \underline{i}_{out} |_{-y_j} \mid 1 \leq j \leq k\}. \end{aligned}$$

As in the previous cases for an easier understanding we will consider the 3 integer sorting problem.

Example 5.1.4 Initial data: $a^{n_1} b^{n_2} c^{n_3} \langle ab \rangle_2 \langle ac \rangle_2 \langle bc \rangle_2 \overline{A} \overline{A} \overline{A} \overline{B} \overline{B} \overline{B} \overline{C} \overline{C} \overline{C} \overline{C}$

Rules: $a \rightarrow a_b a_c a_0$, $b \rightarrow b_a b_c b_0$, $c \rightarrow c_a c_b c_0$, $a_b b_a \rightarrow \varepsilon$, $a_c c_a \rightarrow \varepsilon$, $b_c c_b \rightarrow \varepsilon$,
 $\langle ab \rangle_2 \rightarrow \langle ab \rangle_1$, $\langle ac \rangle_2 \rightarrow \langle ac \rangle_1$, $\langle bc \rangle_2 \rightarrow \langle bc \rangle_1$, $\langle ab \rangle_1 \rightarrow \langle ab \rangle$, $\langle ac \rangle_1 \rightarrow \langle ac \rangle$, $\langle bc \rangle_1 \rightarrow \langle bc \rangle$,
 $a_b \langle ab \rangle \rightarrow A$, $a_c \langle ac \rangle \rightarrow A$, $b_a \langle ab \rangle \rightarrow B$, $b_c \langle bc \rangle \rightarrow B$, $c_b \langle ac \rangle \rightarrow C$, $c_b \langle bc \rangle \rightarrow C$,
 $\overline{A} \overline{A} \rightarrow \overline{A}$, $\overline{B} \overline{B} \rightarrow \overline{B}$, $\overline{C} \overline{C} \rightarrow \overline{C}$, $a_0 \rightarrow a_{out} |_{-A}$, $b_0 \rightarrow b_{out} |_{-B}$, $c_0 \rightarrow c_{out} |_{-C}$.

Practically, in the example it is presented how the sorting algorithm works in the case of three numbers represented as multiplicities of the three objects a , b and c . The system, which has only one membrane, starts with the multiset

$$a^{n_1}, b^{n_2}, c^{n_3}, \langle ab \rangle_2, \langle ac \rangle_2, \langle bc \rangle_2, \overline{A}, A^3, \overline{B}, B^3, \overline{C}, C^3$$

representing the objects for which we want to sort their multiplicities, plus some other objects which will be used during the computation for delaying the execution of certain rules or for eliminating the objects in the right order. As above, we will consider that $n_1 \geq n_2 \geq n_3$. First the device starts by executing in the maximum parallel manner, in one step, the rules $a \rightarrow a_b a_c a_0$, $b \rightarrow b_a b_c b_0$, $c \rightarrow c_a c_b c_0$. Also, at the same time, the rules $\overline{AA} \rightarrow \overline{A}$, $\overline{BB} \rightarrow \overline{B}$, $\overline{CC} \rightarrow \overline{C}$ are executed as well as $\langle ab \rangle_2 \rightarrow \langle ab \rangle_1$, $\langle ac \rangle_2 \rightarrow \langle ac \rangle_1$, $\langle bc \rangle_2 \rightarrow \langle bc \rangle_1$. The first group of rules mentioned will produce in the membrane the objects $a_b^{n_1}, a_c^{n_1}, a_0^{n_1}, b_a^{n_2}, b_c^{n_2}, b_0^{n_2}, c_a^{n_3}, c_b^{n_3}, c_0^{n_3}$ (all the objects a, b, c being transformed) while the other groups are used only for delaying activities.

The transition to the next configuration is made by the rules $a_b b_a \rightarrow \varepsilon$, $a_c c_a \rightarrow \varepsilon$, $b_c c_b \rightarrow \varepsilon$, which will delete from the membrane n_2 copies of a_b and b_a , n_3 copies of a_c and c_a and finally there will be deleted n_3 copies of b_c and c_b . Simultaneously the rules $\overline{AA} \rightarrow \overline{A}$, $\overline{BB} \rightarrow \overline{B}$, $\overline{CC} \rightarrow \overline{C}$, as well as $\langle ab \rangle_1 \rightarrow \langle ab \rangle_0$, $\langle ac \rangle_1 \rightarrow \langle ac \rangle_0$, $\langle bc \rangle_1 \rightarrow \langle bc \rangle_0$, are applied. This means that the configuration of the P system in terms of the objects will be the following: $a_b^{n_1-n_2}, a_c^{n_1-n_3}, a_0^{n_1}, b_c^{n_2-n_3}, b_0^{n_2}, c_0^{n_3}, \overline{A}, A, \overline{B}, B, \overline{C}, C$.

Now is the time when the rules $a_b \langle ab \rangle \rightarrow A$, $a_c \langle ac \rangle \rightarrow A$, $b_a \langle ab \rangle \rightarrow B$, $b_c \langle bc \rangle \rightarrow B$, $c_b \langle ac \rangle \rightarrow C$, $c_b \langle bc \rangle \rightarrow C$, can be applied. Their goal is to produce objects A , B or C (in our case A^2, B) that will be used as inhibitors in the rules and so will forbid, in one step, the execution of corresponding rules. This finishes the execution of the algorithm, because the objects will be eliminated in the right order.

Remark 5.1.4 *The time complexity for the strong sorting algorithm with P systems with inhibitors is $2k$, where k is the number of elements to be sorted.*

5.1.3 Open Problems and Forthcoming Research

In this section we have studied the possibility to solve the sorting problem in the P system framework by considering two variants of membrane devices: P systems with promoters and with inhibitors. The interesting result concerning this topic is that starting with objects that do not have any order and being mixed together in what formally we call a multiset, we constructed the order by computing.

The common feature shared by presented algorithms is that we sort by “carving” (consuming objects iteratively, one symbol from all the components at once) and signaling when a modification occurs in the system (usually we trigger a signal

when a certain component was eliminated). In this way, we were able to compare the multiplicities of objects and to sort them.

The improvements of current algorithms (by reducing the number of membranes when this is the case, reducing the “sensitivity” of features used) are left open. Yet another interesting research topic is designing *dynamic* sorting P systems i.e., systems that solve the sorting problem in case of an arbitrary number of components.

5.2 Simulating Boolean Circuits with P Systems

5.2.1 Motivations

When dealing with biologically inspired models of computation, as opposed to the existing *in silico* devices, it is interesting to make a direct comparison between the two, using simulation: either to simulate an *in vitro/in vivo* process using *in silico* devices, or, vice-versa, to simulate classical *in silico* devices using the new bio-molecular device.

Boolean circuits are well known classical computing devices, which incorporate features of parallelism. In [59] a model for simulating Boolean circuits with DNA algorithms is proposed. This section proposes a simple model for simulating Boolean circuits with P systems.

In Subsection 5.2.2 some notions on Boolean gates and circuits are presented. Subsections 5.2.3 and 5.2.4 are devoted to the main topic, the simulation of Boolean circuits with P systems. The simulation is done in two steps. First, we simulate the classical logical gates, NOT, AND and OR (considered as unary, and respectively binary gates). This will be done in Subsection 5.2.3 using P systems with symbol objects and context-free rewriting rules, promoters and only one *catalyst*.

All the models proposed to simulate the logical gates have a nested membrane structure. This allows us to have the input data in the innermost membrane and the functioning of each simulated gate is such that the correct result is expelled in the environment. Subsection 5.2.4 presents a way to combine individual P systems which serve as logical gates in a hierarchical tree structure, and obtain P systems which simulate an entire circuit.

The first approach of simulating Boolean circuits in the P system framework was done in [26]. Since then, many papers dealt with this topic, the interest of scientific community being toward obtaining possible models of circuits for implementing into bio-ware. Here we only mention [7], [48], [51], and [52].

5.2.2 Boolean Circuits Preliminaries

Boolean circuits are a formal model of the combinational logic circuits.

Circuits consist of wires able to carry one bit, and logical gates connecting such wires and computing the elementary logical functions, \neg (NOT), \wedge (AND), \vee (OR).

Consider the smallest Boolean algebra $B = \langle \{0, 1\}, \wedge, \vee, \neg, 0, 1 \rangle$ with support $\{0, 1\}$, binary operations \wedge, \vee , unary operator \neg , and zero-ary operations (or constants) 0 and 1.

Binary and unary operations on finite sets can be entirely described by operation tables, depicting the result of the operation on any possible input. The operation tables for binary \wedge, \vee , and unary \neg are given in Figure 5.1:

\wedge	0	1	\vee	0	1	\neg	
0	0	0	0	0	1	0	1
1	0	1	1	1	1	1	0

Figure 5.1: Value tables for \wedge (AND), \vee (OR), \neg (NOT)

Consider $B_k = \{f \mid f : \{0, 1\}^k \rightarrow \{0, 1\}\}$ the set of k -ary Boolean functions. Note that: $0, 1 \in B_0$; $\neg \in B_1$; $\wedge, \vee \in B_2$, and they are called the *elementary Boolean functions*. Note also that, because \vee is associative, one can define a 3-ary (ternary) OR, $\vee^3 : \{0, 1\}^3 \rightarrow \{0, 1\}$ by

$$\vee^3(x_1, x_2, x_3) = (x_1 \vee x_2) \vee x_3 = x_1 \vee (x_2 \vee x_3).$$

This can be extended to an m -ary OR, $\vee^m : \{0, 1\}^m \rightarrow \{0, 1\}$, $\vee^m(x_1, x_2, \dots, x_m) = x_1 \vee x_2 \vee \dots \vee x_m$. The same holds for the AND operation which can be extended to an m -ary AND, \wedge^m , with $m \geq 2$.

Definition 5.2.1 A Boolean circuit $\alpha = (V, E, \lambda)$ is a finite directed acyclic graph (V, E) , with set of vertices V , and set of directed edges E , and $\lambda : V \rightarrow \{I\} \cup \{\wedge, \vee, \neg\}$ a vertex labeling, where I is a special symbol. A vertex $x \in V$ with $\lambda(x) = I$ has indegree 0 and is called an input. A vertex $y \in V$ with outdegree 0 is called an output. Vertices with labels \wedge and \vee have indegree 2 and outdegree 1, while vertices with the label \neg have indegree and outdegree 1.

The inputs of α are given by n -tuples $\langle x_1, x_2, \dots, x_n \rangle$ of distinct vertices, and the output by m -tuples $\langle y_1, y_2, \dots, y_m \rangle$. In circuit theory the vertices with labels not I

are called *gates*, the indegree of a vertex is also called the fan-in, and the outdegree the fan-out.

Above, we have defined a circuit whose gates are labeled with the elementary Boolean functions $\{\wedge, \vee, \neg\} \subseteq B_1 \cup B_2$. More generally, we can speak of circuits with vertex labeling $\lambda : V \rightarrow \{I\} \cup B_0 \cup B_1 \cup \dots \cup B_k$. A vertex x with $\lambda(x) \in B_i$ has indegree i .

Definition 5.2.2 *A circuit α with input $\langle x_1, x_2, \dots, x_n \rangle$ and output $\langle y_1, y_2, \dots, y_m \rangle$ computes a function $f : \{0, 1\}^n \rightarrow \{0, 1\}^m$ in the following way: for every i , $1 \leq i \leq n$, to the input x_i is assigned a value $\text{val}(x_i) \in \{0, 1\}$ representing the i -th bit argument of the function. Every other vertex x is assigned the unique value $\text{val}(x) \in \{0, 1\}$ obtained by applying the operation $\lambda(x)$ to the values of the vertices incoming into x . The m -tuple $\langle \text{val}(y_1), \dots, \text{val}(y_m) \rangle$ is the value of function f ; every output vertex y_j gives the j -th bit of the output.*

For $m = 1$, the circuits will compute functions $f : \{0, 1\}^n \rightarrow \{0, 1\}$, and we will consider here only such circuits and such functions.

Measures of interest for circuits are *the size* and *the depth*.

Definition 5.2.3 *The size of a circuit α is the $\text{size}(\alpha) = |V|$ (the number of vertices). The depth of a circuit α is $\text{depth}(\alpha) =$ the length of the longest path in α from an input to an output.*

A special class of circuits is obtained when we take $m = 1$ and the underlying graph of the circuit is a rooted tree. The root of the tree will correspond to its unique output gate, and the input gates are its leaves. Every input gate will have outdegree 1 – which means that the circuit will not be able to compute functions such as $(x_1 \wedge x_2) \vee (x_1 \vee x_3)$ (actually, this function can be computed, introducing another variable x_4 : we then compute $(x_1 \wedge x_2) \vee (x_4 \vee x_3)$, which has a tree circuit, and we ensure that $x_1 = x_4$). Also, there will be no feedback to any of the gates – which means that the gates are not reusable. In the rest of the work we will consider only such circuits and, moreover, with gates labeled in $\{\wedge, \vee, \neg\}$.

5.2.3 Simulating Boolean Gates

We present in this subsection P systems which simulate the logical gates. All the gates (and other auxiliary devices) will be implemented with nested membrane structures. We will consider that the input for a gate is given in the inner membrane,

while the output will be computed and sent to the outer region. Practically, the gate will act as a filter that receives two symbols synchronously, and, after some computation steps, sends the result in the environment.

The easiest gate to simulate is the NOT gate.

Simulation of a NOT gate

NOT diagram, also known as the “Inverter” diagram, just switches the value that enters the gate into its complement value. We can do this with one membrane and non-cooperative rules, as shown in Figure 5.2.

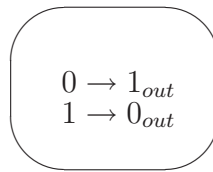


Figure 5.2: Simulation of NOT gate

Formally, we define this by:

$$\Pi_{NOT} = (V, C, P, I, \mu, w_1, R_1, i_0),$$

where:

- $V = \{0, 1\}$;
- $C = P = I = \emptyset$;
- $\mu = []_1$;
- $R_1 = \{1 \rightarrow 1_{out}, 0 \rightarrow 0_{out}\}$.
- $i_0 = 1$

Simulation of an AND gate

First, let us note that both the AND and the OR operations can be very easily simulated if we allow cooperative rules in our membranes.

The P system with only one membrane, and context-sensitive rules from Figure 5.3, computes the AND of two (not necessarily synchronized) input values (the rules are inspired directly from the table of the AND operation).

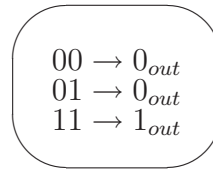


Figure 5.3: Simulation of an AND gate using cooperative rules

In Figure 5.4 we present a P system with promoters and one catalyst that models the AND gate. In this simulation we will consider that the input to the gates arrives in the innermost membrane, considered as an input membrane for the whole mechanism. The result of the computation will be sent out into the environment.

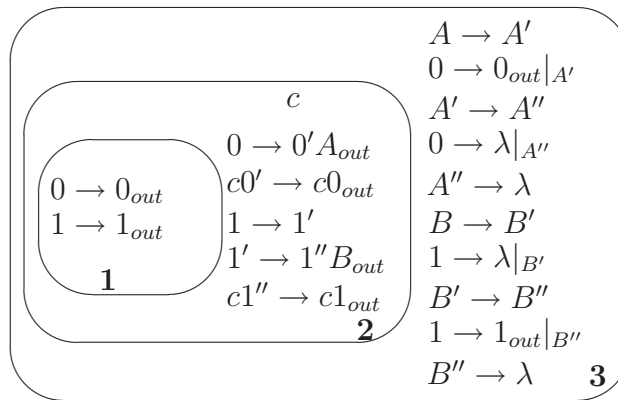


Figure 5.4: Simulation of the AND gate using promoters and one catalyst

Formally, we define the following P system

$$\Pi_{AND} = (V, C, P, I, \mu, w_1, w_2, w_3, R_1, R_2, R_3, 0),$$

where:

- $V = \{0, 1, 0', 1', 1'', A, A', A'', B, B', B'', c\}$;
- $C \subseteq V = \{c\}$ the set of catalysts;
- $P = \{A', A'', B', B''\}$;
- $I = \emptyset$;
- $\mu = [[[]_1]_2]_3$;

- $w_2 = \{c\}$;
- $R_1 = \{1 \rightarrow 1_{out}, 0 \rightarrow 0_{out}\}$;
 $R_2 = \{0 \rightarrow 0'A_{out}, c0' \rightarrow c0_{out}, 1' \rightarrow 1''B_{out}, 1 \rightarrow 1', c1'' \rightarrow c1_{out}\}$;
 $R_3 = \{A \rightarrow A', 0 \rightarrow 0_{out}|_{A'}, A' \rightarrow A'', 0 \rightarrow \lambda|_{A''}, A'' \rightarrow \lambda, B \rightarrow B',$
 $1 \rightarrow \lambda|_{B'}, B' \rightarrow B'', 1 \rightarrow 1_{out}|_{B''}, B'' \rightarrow \lambda\}$;
- $i_0 = 0$.

The AND gate uses the catalyst c to inhibit the parallelism and to separate the entrance time of objects 0 and 1 into region 3. According to the entrance time, objects will be either deleted, or sent out into the environment. More specifically, if we consider that initially we had two objects 0 inside region 2, the rule $0 \rightarrow 0'A_{out}$ is executed. Its role is to introduce the object A into region 3 to set up the “right” configuration of the region. Next, in region 2 the only applicable rule is $c0' \rightarrow c0_{out}$, which will introduce one object 0 into region 3. At the same time, in region 3 the rule $A \rightarrow A'$ is executed. Now, we will have in region 3 the objects A' and 0, and the rules that will be applied are $0 \rightarrow 0_{out}|_{A'}$ and $A' \rightarrow A''$. These rules guarantee that an object 0 is sent out into the environment. In the meantime, in region 2, the remaining object $0'$ reacts with the catalyst c and an object 0 will be introduced into region 3 (the rule used is again $c0' \rightarrow c0_{out}$). Here, the object 0 will find a different context since now, in region 3 there is no object A' . Therefore, the rules $0 \rightarrow \lambda|_{A''}$ and $A'' \rightarrow \lambda$ are applied, hence the initial configuration of the system is restored. Basically, a similar method stands for the other cases, with some minor changes: objects 1 enter into region 3 with one computational delay (because of the rule $1 \rightarrow 1'$ present in region 2) in order not to influence the processes executing in region 3; the first object 1 that enters into region 3 is deleted (as opposed to the above case when the first object 0 that arrives in region 3 is sent out) by using the rule $1 \rightarrow \lambda|_{B'}$.

We have the following result:

Lemma 5.2.1 *The P system Π_{AND} , with promoters and one catalyst, acting on pairs of input values x_1, x_2 from $\{0, 1\}$, is deterministic, and produces into the environment the value $x_1 \wedge x_2$.*

Finally, one can notice that the number of membranes can be reduced by one if we use more symbols. However, for the sake of clarity, we have considered this 3 membrane system in order to distinguish the “logical” tasks.

5.2.4 Simulating Boolean Circuits

In this subsection we show that any Boolean circuit whose underlying graph structure is a binary tree structure, can be simulated with P systems, using the P systems Π_{AND} , Π_{OR} , and Π_{NOT} which model the gates. The fact that all the individual gates have a nested membrane structure is essential: if the output of two gates, let us say a Π_{AND} and a Π_{OR} , has to be the input of another gate, let us say Π'_{AND} , then we can embed the system Π_{AND} and Π_{OR} in a common membrane, in which the results of their individual computations will be expelled. Further, this membrane will be the innermost membrane (input membrane) for the hierarchically higher Π'_{AND} . There is only one problem to be solved, namely, to synchronize two output values when they become input values for a gate.

We have to design a module that synchronizes the input for the gates. This is done because all the gates (with the exception of the NOT gate) are binary operators and, if the input consists of only one operand, the computation might not work well. For this we designed a system which, if the input consists of only one symbol, will block the computation and will not produce any output up to the time when the second input symbol enters the system; then, both input symbols will leave the system at the same time and so, they can become the input for a gate. Let us denote this system by “SYNC” (Figure 5.5).

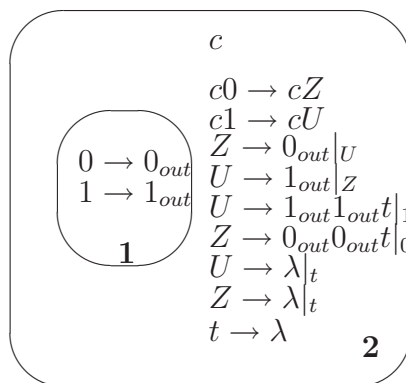


Figure 5.5: A P system which synchronizes the input

Formally, we define the SYNC module as :

$$\Pi_{SYNC} = (V, C, P, I, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = \{0, 1, Z, U, t\}$;
- $C = \{c\}$ the set of catalysts;
- $P = \{Z, U, 0, 1, t\}$ the set of promoters;
- $I = \emptyset$;
- $\mu = [[]_1]_2$;
- $w_2 = \{c\}$;
- $R_1 = \{0 \rightarrow 0_{out}, 1 \rightarrow 1_{out}\}$;
 $R_2 = \{c0 \rightarrow cZ, c1 \rightarrow cU, Z \rightarrow 0_{out}|_U, U \rightarrow 1_{out}|_Z, U \rightarrow 1_{out}1_{out}t|_1\} \cup$
 $\{Z \rightarrow 0_{out}0_{out}t|_0, U \rightarrow \lambda|_t, Z \rightarrow \lambda|_t, t \rightarrow \lambda\}$;
- $i_0 = 0$.

Here is how the system works. The catalyst c , present in region 2 inhibits the parallelism such that in case objects 0 and 1 enters simultaneously it will delay one of reactions $c0 \rightarrow cZ$ or $c1 \rightarrow cU$ with one time unit – enough time to set up the conditions that will lead to a correct output. Recall that objects Z and U correspond to 0 and 1, respectively.

Next, after objects Z and U were produced, both rules $Z \rightarrow 0_{out}|_U$ and $U \rightarrow 1_{out}|_Z$ are executed simultaneously. As a consequence, objects 0 and 1 are sent synchronously into the outer region.

Now, in case two objects 0 enter in the same time in region 2, then, after rule $c0 \rightarrow cZ$ is executed once, we will have in the region the multiset $\{c, 0, Z\}$. This means that the rules to be applied are $Z \rightarrow 0_{out}0_{out}t|_0$ and $c0 \rightarrow cZ$. In this way both objects 0 are sent synchronously into the outer region. The object t produced is useful to delete the remaining object Z ; it will be deleted as well by the rule $t \rightarrow \lambda$.

One can notice that the system's initial configuration will be restored and so, in principle, it can be used later in other computations.

Let us see now how the system will react in the case when only one symbol 1 enters in region 2. There, only the rule $c1 \rightarrow cU$ will be applied. After this, the system will wait for the next input. When this will become available the system will work as described above. In this way, the input that initially was unsynchronized is synchronized so it can be used further as input for a Boolean gate.

Due to the symmetry of the rules we will have the same behavior for all synchronous and asynchronous pairs of input values. Moreover, one can see that the system is confluent.

We give now an example of how to construct a P system which simulates a Boolean circuit, designed for evaluating a Boolean function, using the basic P systems Π_{AND} , Π_{OR} , and Π_{NOT} constructed in the previous sections. Consider the function $f : \{0, 1\}^4 \rightarrow \{0, 1\}$ given by the formula $f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$.

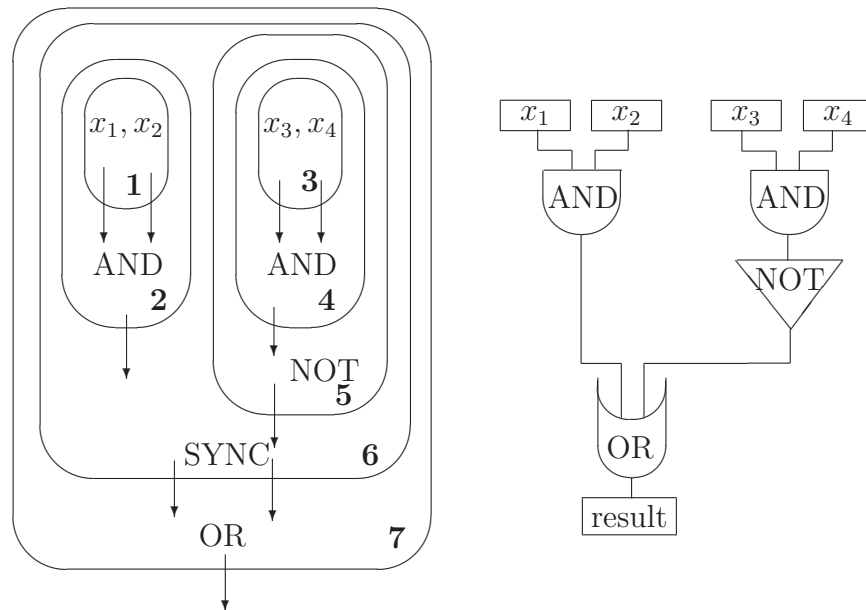


Figure 5.6: The P system which simulates the computation of its associated circuit

A Boolean circuit which evaluates this function is depicted in Figure 5.6. Note that this circuit has a binary tree as its underlying graph, with the leaves as input gates, and the root as output gate.

We simulate this circuit with the P system described (incompletely) in Figure 5.6, and constructed with the following recipe:

1. for every gate of the circuit, with input(s) from input gates (leaves), we have an appropriate P system simulating it, with the innermost membrane containing the input values;
2. for every gate which has at least one input coming as an output of a previous gate, we construct an appropriate P system to simulate it by embedding in

membrane the “environments” of the P systems which compute the gates at the previous level, consider this as an innermost membrane of a “synchronizer”, and the last membrane of the synchronizer as the inner membrane (input membrane) of the new gate.

In our model we omit, for the time being, treating in detail the problem of the input. That is, we make abstraction, for instance, of how input values for the pair (x_1, x_2) arrive in membrane 1.

For the particular formula $(x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$ and the circuit depicted in Figure 5.6 we will have:

- $\Pi_{AND}^{(1)}$ is responsible for computing $(x_1 \wedge x_2)$; it is depicted in Figure 5.6, but only with its inner membrane labeled 1 (input membrane for values of x_1 and x_2), and its outer membrane, labeled 2, from which the result will be send out; recall that between membranes 1 and 2 we have a whole nested membrane structure, with the number of membranes depending on the implementation.
- $\Pi_{AND}^{(2)}$ is responsible for computing $x_3 \wedge x_4$; it is depicted with the extreme membranes only, labeled 3 and 4. These two P systems act in parallel.
- $\Pi_{NOT}^{(3)}$ is responsible for computing $\neg(x_3 \wedge x_4)$; the innermost membrane of the Π_{NOT} will be the former environment of the $\Pi_{AND}^{(2)}$, and so it will contain the result of computing $x_3 \wedge x_4$; the last membrane of $\Pi_{NOT}^{(3)}$ is labeled 5.
- The output of $\Pi_{AND}^{(1)}$ and of $\Pi_{NOT}^{(3)}$ have to be input values for an OR; we let membrane 6 embed $\Pi_{AND}^{(1)}$ and $\Pi_{NOT}^{(3)}$ and this will be the innermost membrane of $\Pi_{SYNC}^{(4)}$. $\Pi_{SYNC}^{(4)}$ has a nested membrane structure not depicted in Figure 5.6.
- $\Pi_{OR}^{(5)}$ has as inner membrane the environment of $\Pi_{SYNC}^{(4)}$. Its external membrane is labeled 7, and the result of its computation is sent in the environment.

Therefore we can conclude, based on the previous facts, that:

Theorem 5.2.1 *Every Boolean circuit α whose underlying graph structure is a rooted binary tree can be simulated by a P system, Π_α . Π_α is constructed from standard P systems of type Π_{AND} , Π_{OR} , and Π_{NOT} , by reproducing in the architecture of the membrane structure, the structure of the tree associated to the circuit. The time complexity for computing a Boolean function depends linearly on the depth of the underlying tree structure of the P system.*

5.2.5 Open Problems and Forthcoming Research

We have proposed a model for simulating Boolean circuits with P systems.

As a first step, we have simulated the gates which compute the elementary Boolean functions, NOT, AND, and OR with P systems with promoters. The depth of the nested membrane structures which simulate the elementary gates decreases when we use more powerful ingredients for our P systems. The time in which such a gate computes the output of the Boolean function is linearly dependent on the depth of the simulated gate.

As a second step, we have given indications on how to use a circuit architecture in order to build a similar architecture for a P system which simulates an entire circuit. The second construction uses as an extra ingredient a “synchronizer”. The time for computing a function with the P system simulation is the depth of the underlying tree structure of the P system, and is proportional to the time required by the Boolean circuit, adding the costs of the individual components implemented with nested membrane structures of a certain length.

The model needs some further developments. The problem of how the input values reach the appropriate input membranes is left open for further work. Also, the mechanism of constructing the simulator of a circuit out of individual components could be further improved, or addressed with different tools, like dynamic P systems.

Other topics of interest in this area are:

- improving the performance of the simulator with respect to the circuit model;
- improving the performance of the circuit model itself, with specific P systems methods (for instance, the problem of re-using gates is not solved yet in circuit theory);
- reducing the number of membranes and/or “sensitive” components for the Boolean gates presented, as well as for the synchronizer.

We mention also the fact that, in the present model, the “parallel computing” feature of P systems is present at two levels: at the level of the individual gates, and at the level of the circuit simulation. At this second level, the parallelism is the same as that present in the circuit model. Maybe these two levels of parallelism could be merged, and the performance improved with P systems techniques.

Further improvements can be obtained by passing to the simulation of k -ary gates, with $k > 2$.

Chapter 6

Two Natural Extensions

In this chapter we will introduce two new models derived from the classical definition of P systems with promoters and inhibitors. *P systems with external promoters/inhibitors* extend the concept of the promoting/inhibiting mechanism by considering computations controlled by the external environment. Recall that membranes in the P systems framework represent frontiers that divide the space into regions. From this perspective, as we will see later, the computation performed by this model can be understood as an interactive and cooperative computation done by more entities, each one evolving not only based on its own rules but also with respect to its neighboring commands. However, here we have considered only the classical tree structure of membranes and not the graph structure of membranes as it is defined, for example, in tissue-like P systems. The model can be furthermore extended in this direction.

In what concerns *string-driven P systems*, they encapsulate in a certain sense, the von Neumann's idea of a stored program. In this design, a mobile initial string of promoters/inhibitors drives the computation by enabling/disabling rules while it is "consumed" each time crosses a membrane.

6.1 P Systems with External Promoters/Inhibitors

Up to now we have considered P systems with promoted/inhibited non-cooperative or cooperative rules where fundamental for the execution of such rules was the fact that the promoting or the inhibiting context was present in the same region as the corresponding rules. In this approach the membrane structure proved not to have an important role, every membrane design being able to be simulated by a P system

with one membrane via an encoding of region labels into symbols.

Here we deal with P systems having the rules from a given region activated by the presence or the absence of certain symbols in the immediate neighboring (inner or outer) region. This model has a biological counterpart and it is inspired by the chemicals that pass through the membranes of the cell, from one region to another, in the sense of polarization gradient. In this case, the electrical charge plays the role of the promoter.

Inspired by these facts we can introduce the following new class of P systems:

Definition 6.1.1 *A communication P system with rewriting-symport rules (called in short, a PRS system) and external context is a construct*

$$\Pi = (V, C, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

where:

- V is the alphabet of objects;
- $C \subseteq V$ is the set of catalysts;
- μ is a membrane structure with m membranes (labeled in a one-to-one manner by $1, \dots, m$);
- w_1, \dots, w_m are the multisets of objects initially present in the regions of the system;
- R_1, \dots, R_m are finite sets of rules associated to membranes, that are of the following types:

◇ simple rules:

$$[A]_i \longrightarrow []_i \alpha \text{ or } A[]_i \longrightarrow [\alpha]_i, \text{ for } A \in V \setminus C, \alpha \in (V \setminus C)^*,$$

◇ inhibited simple rules:

$$[A]_i \neg B \longrightarrow []_i \alpha \text{ or } A[\neg B]_i \longrightarrow [\alpha]_i, \text{ for } A, B \in V \setminus C, \alpha \in (V \setminus C)^*,$$

◇ catalytic rules:

$$\text{pairs of rules } [cA]_i \longrightarrow []_i c\alpha \text{ and } c[]_i \longrightarrow [c]_i, \text{ for } A \in V \setminus C, c \in C, \alpha \in (V \setminus C)^*, \text{ or}$$

$$\text{pairs of rules } cA[]_i \longrightarrow [c\alpha]_i \text{ and } [c]_i \longrightarrow []_i c \text{ for } A \in V \setminus C, c \in C, \alpha \in (V \setminus C)^*,$$

◇ inhibited catalytic rules:

pairs of rules $[cA]_i \neg B \longrightarrow []_i c\alpha$ and $c[]_i \longrightarrow [c]_i$, for $A, B \in V \setminus C$, $c \in C$, $\alpha \in (V \setminus C)^*$, or

pairs of rules $cA[\neg B]_i \longrightarrow [c\alpha]_i$ and $[c]_i \longrightarrow []_i c$, for $A, B \in V \setminus C$, $c \in C$, $\alpha \in (V \setminus C)^*$;

- i_0 is an elementary membrane of μ (the output membrane).

As usual for the P systems, the system evolves from an initial configuration, using the rules in a maximally parallel manner, with their applications controlled by the promoters/inhibitors, and we will consider the result at halting.

We use the notation:

$$PsPCC_m(\alpha, \beta), \alpha \in \{smp\} \cup \{cat_k \mid k \geq 0\}, \beta \in \{proR_i, inhR_i\}$$

to denote the family of sets of vectors of natural numbers generated by P systems with controlled communication having at most m membranes, communication rules that can be simple $\alpha = smp$, or catalytic $\alpha = cat_k$, using at most k catalysts, and external promoters $\beta = proR_i$ or external inhibitors $\beta = inhR_i$ of weight i at the level of rules.

In what follows we will prove that the class of sets of vectors generated by P systems with external inhibitors equals the class of sets of vectors generated by P systems with external inhibitors and only two membranes.

Lemma 6.1.1 $PsPCC_m(smp, inhR_1) = PsPCC_2(smp, inhR_1), m \geq 2$.

Proof. Obviously, $PsPCC_m(inh) \supseteq PsPCC_2(inh)$. For the opposite inclusion we have to show that for any P system with external inhibitors $\bar{\Pi} = (\bar{V}, \bar{C}, \bar{\mu}, \bar{R}, \bar{i}_0)$ generating a set of vectors, there exists an equivalent P system with external inhibitors $\Pi = (V, C, \mu, R, i_0)$ with only 2 membranes.

To this aim, we simulate the computation of $\bar{\Pi}$, with the system Π defined as follows.

Let us denote by $\mathcal{L} = \{1, 2, \dots, m\}$ the set of labels of the regions in $\bar{\Pi}_m$. In addition, assume that $\bar{R} = \{R_1, \dots, R_m\}$, and each $\bar{R}_i \in \bar{R}$, $1 \leq i \leq m$ contains all the rules that cross membrane i . Then, we define:

- $V = \{a_i \mid a \in \bar{V}, i \in \mathcal{L}\}$.
- $C = \bar{C} = \emptyset$;

Let $h : \bar{V}^* \times \mathcal{L} \rightarrow V^*$ be a mapping such that

- 1) $h(a, i) = a_i, a \in \bar{V}, i \in \mathcal{L}$;
- 2) $h(\lambda, j) = \lambda, j \in \mathcal{L}$;
- 3) $h(x_1x_2, j) = h(x_1, j)h(x_2, j), x_1, x_2 \in \bar{V}^*, j \in \mathcal{L}$.

• denote $w = h(\bar{w}_1)h(\bar{w}_2) \dots h(\bar{w}_m)$, where \bar{w}_i is the multiset present in region $i \in \mathcal{L}$ of $\bar{\Pi}_m$ at the beginning of the computation.

• R is defined as follows.

For each rule $A[]_i \longrightarrow [\alpha]_i \in R_i, A \in \bar{V}, \alpha \in \bar{V}^*, i \in \mathcal{L}$, we add to R the rule $h(A, j)[]_1 \longrightarrow [h(\alpha', i)]_1$, providing that j is the label of the outer membrane of membrane i .

For each rule $A[\neg B]_i \longrightarrow [\alpha]_i \in R_i, A, B \in \bar{V}, \alpha \in \bar{V}^*, i \in \mathcal{L}$, we add to R the rule $h(A, j)[\neg h(B, i)]_1 \longrightarrow [h(\alpha', 2)]_1$, providing that j is the label of the outer membrane of membrane i .

For each rule $[A]_i \longrightarrow []_i \alpha \in R_i, A, B \in \bar{V}, \alpha \in \bar{V}^*, i \in \mathcal{L}$, we add to R the rule $[h(A, i)]_1 \longrightarrow []_1 h(\alpha', j)$ providing that j is the outer membrane of membrane i .

For each rule $[A]_i \neg B \longrightarrow []_i \alpha \in R_i, A, B \in \bar{V}, \alpha \in \bar{V}^*, i \in \mathcal{L}$, we add to R the rule $[h(A, i)]_1 \neg h(B, j) \longrightarrow []_1 h(\alpha', j)$ providing that j is the outer membrane of membrane i .

Generally speaking, the purpose of membranes is to keep private the interior rules and objects from the neighboring ones and vice-versa. However, in our case we can express the passage of certain symbol through the membranes by using new symbols that we add to vocabulary and that encode both the crossed membrane label and the symbols from where they derive. In this way we can rewrite the rules, using the new symbols that perfectly describe the passage of objects in the membrane structure; consequently, in our case, we can shrink an arbitrarily membrane structure to only two membranes. The morphism used by the above construction accomplishes the encoding procedure. \square

Here we will prove that the family of sets of vectors of numbers generated by P systems with external inhibitors equals the family of Parikh images of languages generated by ETOL systems.

Theorem 6.1.1 $PsPCC_2(smp, inhR_1) = PsETOL$.

Proof. We will prove the result by showing that communicative P systems with external inhibitors are equivalent with P systems with inhibitors, which at their

turn, generates the same class of sets of vectors as the Parikh image of $ETOL$ as shown in Chapter 3.

The proof of the inclusion $PsP_1(smp, inhR_1) \supseteq PsPCC_2(smp, inhR_1)$ is rather simple and is based on a similar encoding of regions into new objects as was presented above.

For the inclusion $PsP_1(smp, inhR_1) \subseteq PsPCC_2(smp, inhR_1)$ we will simulate the computation of a P system with one region $\Pi_{inh} = (V, C, \mu, w, R, i_0)$. We assume that the set of rules R contains rules of type $A \rightarrow \alpha$ or $A \rightarrow \alpha|_{\neg B}$, $A, B \in V$, $\alpha \in V^*$.

Let us consider the sets $\tilde{V} = \{\tilde{A} \mid A \in V\}$ and $\dot{V} = \{\dot{A} \mid A \in V\}$. In addition, let us define the morphisms:

$$\begin{aligned} h_1 : V^* &\rightarrow \tilde{V}^*, \text{ such that } h_1(A) = \tilde{A} \text{ for all } A \in V; \\ h_2 : V^* &\rightarrow \dot{V}^*, \text{ such that } h_2(A) = \dot{A} \text{ for all } A \in V. \end{aligned}$$

We construct a P system $\Pi_{cc} = (\bar{V}, \bar{C}, \bar{\mu}, \bar{R}, \bar{i}_0)$, simulating Π_{inh} , defined as follows:

$$\begin{aligned} \bar{V} &= V \cup \tilde{V} \cup \dot{V} \cup \{F\}; \\ \bar{C} &= \emptyset; \\ \bar{\mu} &= [[]_2]_1; \\ w_1 &= w; \\ w_2 &= w; \\ \bar{i}_0 &= 1. \end{aligned}$$

The set of rules R is defined as follows¹:

$$\begin{aligned} \text{step } i & \quad A[\neg B] \longrightarrow [\mathbf{h}_1(\alpha)\mathbf{h}_2(\alpha)], \text{ for all rules } A \rightarrow \alpha|_{\neg B} \in R_{inh}, \\ \text{step } i & \quad A[] \longrightarrow [\mathbf{h}_1(\alpha)\mathbf{h}_2(\alpha)], \text{ for all rules } A \rightarrow \alpha \in R_{inh}, \\ \text{step } i & \quad [A] \longrightarrow []F, \text{ if exists } A \rightarrow \alpha \in R_{inh}, \\ \text{step } i & \quad [A]\neg B \longrightarrow []F, \text{ if exists } A \rightarrow \alpha|_{\neg B} \in R_{inh}, \\ \text{step } i + 1 & \quad F[] \longrightarrow [], \\ \text{step } i + 1 & \quad [\mathbf{h}_1(A)] \longrightarrow []\mathbf{h}_1(A), \text{ for all objects } A \in V, \\ \text{step } i + 2 & \quad \mathbf{h}_1(A)[] \longrightarrow [A], \text{ for all objects } A \in V, \end{aligned}$$

¹For the present proof, we will simplify the notation by not including the membrane labels into the syntax of the rules; this is possible here since we have only two membranes and we do not allow the interaction with the environment. In addition, we have specified on their left hand side the moment of their executions during the simulation of one computational step in Π_{inh} .

step $i + 2$ $[\mathbf{h}_2(A)]\neg R \longrightarrow []A$, for all $A \in V$.

Here is how the system Π_{cc} simulates the computation of Π_{inh} . First, remark that in order to correctly simulate the moves of Π_{inh} , we will maintain during the computation in both regions of Π_{cc} a copy of the multiset w – the multiset that represent the current configuration of Π_{inh} . This is especially useful when trying to simulate rules of type $A \rightarrow \alpha|_{\neg B} \in R_{inh}$ because we have to know whether or not the external inhibitor is present.

We assume that the system is in a configuration given by the strings $w_1 = w_2 = w$. The system attempts to execute simultaneously the rules of type

step i $A[\neg B] \longrightarrow [\mathbf{h}_1(\alpha)\mathbf{h}_2(\alpha)]$, for all rules $A \rightarrow \alpha|_{\neg B} \in R_{inh}$,
 step i $A[] \longrightarrow [\mathbf{h}_1(\alpha)\mathbf{h}_2(\alpha)]$, for all rules $A \rightarrow \alpha \in R_{inh}$,
 step i $[A] \longrightarrow []F$, if exists $A \rightarrow \alpha \in R_{inh}$,
 step i $[A]\neg B \longrightarrow []F$, if exists $A \rightarrow \alpha|_{\neg B} \in R_{inh}$.

Remark that the rules of first two types are used to generate inside the inner region, two copies of multiset α (represented by $\mathbf{h}_1(\alpha)$ and $\mathbf{h}_2(\alpha)$). In the same time, the rules of second type delete from region 2 the objects that were within the scope of rules of first type. In addition, remark that there are no other rules that can be applied in this step. Moreover, they produce in region 1 objects R ; these objects will be used later for synchronizing the moments when multiset α appears in both regions.

Next, are executed the rules of type:

step $i + 1$ $F[] \longrightarrow []$,
 step $i + 1$ $[\mathbf{h}_1(A)] \longrightarrow []\mathbf{h}_1(A)$, for all objects $A \in V$.

Observe that the presence of object(s) R in this computational step inhibits the executions of rules of type $[\mathbf{h}_2(A)]\neg F \longrightarrow []A$, for all $A \in V$. Hence, in the third step, the rules of type

$\mathbf{h}_1(A)[] \longrightarrow [A]$, for all objects $A \in V$,
 $[\mathbf{h}_2(A)]\neg F \longrightarrow []A$, for all $A \in V$,

will be executed. The new objects appear at the same time in both regions of the system Π_{cc} and the simulation of the next computational step of Π_{inh} can start.

Finally, if the system Π_{inh} stops because there are no rules to be applied, then also Π_{cc} halts.

Before we conclude, remark that the maximal parallelism as well as the universal clock is fundamental for the construction.

Consequently we have proved that the computation of an arbitrary P system with inhibitors can be simulated by a P system with external inhibitors, hence we have $PsP_1(smp, inhR_1) \subseteq PsPCC_2(smp, inhR_1)$.

Therefore we have that $PsP_1(smp, inhR_1) = PsPCC_2(smp, inhR_1) = PsET0L$. \square

The following theorem shows that catalytic P systems with external inhibitors are computationally complete.

Theorem 6.1.2 $PsPCC_2(cat, inhR_1) = PsRE$.

Proof. The inclusion $PsPCC_2(cat, inhR_1) \subseteq PsRE$ is assumed true by invoking the Turing-Church thesis.

For the inclusion $PsPCC_2(cat, inhR_1) \supseteq PsRE$ we will simulate the computation of an arbitrary deterministic register machine $M = (n, \mathcal{P}, l_0, l_h)$ with a P system $\Pi = (V, C, \mu, w_1, w_2, R_1, i_0)$ defined as follows.

$$\begin{aligned} V &= \{a_i, A_i, S_i \mid 1 \leq i \leq n\} \cup \{l, \bar{l}, \bar{\bar{l}}, \tilde{l}, \tilde{\tilde{l}}, L \mid l \in Lab(\mathcal{P})\} \cup \{c\} \\ &\cup \{K, \bar{K}, \bar{\bar{K}}, \bar{\bar{\bar{K}}}, T_0, T_1, X, \bar{X}\}; \\ C &= \{c\}; \\ \mu &= [[]_2]_1; \\ w_1 &= l_0 L_0 a_1^{k_1} \dots a_n^{k_n} c; \\ w_2 &= A_1 \dots A_n S_1 \dots S_n; \\ i_0 &= 1. \end{aligned}$$

The set of rules R is defined as follows:

- for each instruction $(l_1 : ADD(j), l_2) \in \mathcal{P}$, the set R contains the rules:

$$l_1 [] \longrightarrow [A_1 \dots A_{j-1} A_{j+1} \dots A_n S_1 \dots S_n a_j l_2],$$

$$L_1[\neg A_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n],$$

$$[l_2] \longrightarrow [] l_2,$$

$$[a_j] \longrightarrow [] a_j,$$

$$[A_i] \longrightarrow [] \lambda, \quad 1 \leq i \leq n,$$

$$[S_i] \longrightarrow []\lambda, 1 \leq i \leq n;$$

• for each instruction $(l_1 : \text{SUB}(r), l_2, l_3) \in \mathcal{P}$, the set R contains the rules:

$$\bar{l}_1[] \longrightarrow [A_1 \dots A_n S_1 \dots S_{j-1} S_{j+1} \dots S_n \bar{l}_1],$$

$$ca_j[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n X],$$

$$L_1[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n K],$$

$$[\bar{l}_1] \longrightarrow []\bar{\bar{l}}_1,$$

$$[X] \longrightarrow []\bar{X},$$

$$\bar{\bar{l}}_1[] \longrightarrow [\bar{\bar{l}}_1 T_0 A_1 \dots A_n S_1 \dots S_n],$$

$$[K] \longrightarrow []\bar{K},$$

$$[\bar{\bar{l}}_1] \neg X \longrightarrow []\tilde{l}_3,$$

$$\bar{X}[\neg T_0] \longrightarrow [l_2],$$

$$\bar{K}[] \longrightarrow [A_1 \dots A_n S_1 \dots S_n \bar{\bar{K}}],$$

$$[T_0] \longrightarrow []T_1,$$

$$[\bar{\bar{l}}_1] \neg \bar{K} \longrightarrow []\lambda,$$

$$[l_2] \longrightarrow []l_2 L_2,$$

$$T_1[] \longrightarrow [A_1 \dots A_n S_1 \dots S_n],$$

$$\tilde{l}_3[] \longrightarrow [\tilde{l}_3],$$

$$[\tilde{l}_3] \longrightarrow []l_3 L_3,$$

$$[\bar{\bar{K}}] \longrightarrow []\bar{\bar{\bar{K}}},$$

$$\bar{\bar{\bar{K}}}[] \longrightarrow [A_1 \dots A_n S_1 \dots S_n],$$

$$[A_i] \longrightarrow []\lambda, 1 \leq i \leq n,$$

$$[S_i] \longrightarrow []\lambda, 1 \leq i \leq n.$$

Here is how the P system Π simulates the computation of the register machine M . Observe for the beginning that in the P system Π we will represent the number stored into register j of M as the multiplicity of the object a_j . In addition, remark that objects $A_j, S_j, 1 \leq j \leq n$, stand for the addition/subtraction command over register j – both in the simulation of an ADD or SUB instruction, the absence

of symbol A_j or S_j allows the addition or deletion of one occurrence of object a_j . Objects $A_j, S_j, 1 \leq j \leq n$, are produced all the time during the computation except the moment when we actually want to increment or subtract one occurrence of object a_j from the multiset; at that moment we generate all objects $A_i, S_i, 1 \leq i \leq n$, such that $i \neq j$.

Let us see in more details how the simulation of the addition instruction $(l_1 : \text{ADD}(j), l_2) \in \mathcal{P}$ works. Assume that at a certain moment during the computation, the current multisets in regions 1 and 2 are represented by the strings $w_1 = l_1 L_1 a_1^{k_1} \dots a_n^{k_n} c$ and $w_2 = A_1 \dots A_n S_1 \dots S_n$ respectively. Then, the rules that can be executed are:

$$l_1 [] \longrightarrow [A_1 \dots A_{j-1} A_{j+1} \dots A_n S_1 \dots S_n a_j l_2],$$

$$[A_i] \longrightarrow [] \lambda, 1 \leq i \leq n,$$

$$[S_i] \longrightarrow [] \lambda, 1 \leq i \leq n.$$

As a consequence of executing the above rules the next configuration will be represented by $w_1 = L_1 a_1^{k_1} \dots a_n^{k_n} c$ and $w_2 = A_1 \dots A_{j-1} A_{j+1} \dots A_n S_1 \dots S_n a_j l_2$. Now, since in region 2 the object A_j is missing, then the rule

$$L_1 [\neg A_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n]$$

can be executed; its role is to reestablish the initial configuration in region 2. Simultaneously, the system runs the rules

$$[l_2] \longrightarrow [] l_2,$$

$$[a_j] \longrightarrow [] a_j,$$

$$[A_i] \longrightarrow [] \lambda, 1 \leq i \leq n,$$

$$[S_i] \longrightarrow [] \lambda, 1 \leq i \leq n.$$

The rule $[l_2] \longrightarrow [] l_2$ produces in region 1 the object l_2 that corresponds to register machine label l_2 . In addition, by the execution of the rule $[a_j] \longrightarrow [] a_j$, the number of objects a_j in region 1 (that corresponds to the number stored in register j of M) is incremented.

Concerning the simulation of the subtract instruction $(l_1 : \text{SUB}(j), l_2, l_3) \in \mathcal{P}$, the system Π , being in a configuration represented by $w_1 = l_1 L_1 a_1^{k_1} \dots a_n^{k_n} c$ and $w_2 = A_1 \dots A_n S_1 \dots S_n$, executes first the rules:

$$l_1 [] \longrightarrow [A_1 \dots A_n S_1 \dots S_{j-1} S_{j+1} \dots S_n \bar{l}_1],$$

$$[A_i] \longrightarrow [] \lambda, 1 \leq i \leq n,$$

$$[S_i] \longrightarrow []\lambda, 1 \leq i \leq n.$$

In a similar manner as presented in the addition simulation, the rule $l_1[] \longrightarrow [A_1 \dots A_n S_1 \dots S_{j-1} S_{j+1} \dots S_n \bar{l}_1]$ creates the context required for starting the simulation. The absence of object S_j in region 2 allows, in the second step, the (possible) execution of the rules

$$ca_j[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n X],$$

$$L_1[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n K].$$

Observe that in case there exists an object a_j in region 1, both rules are executed, while if there is not, only the rule $L_1[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n K]$ will be executed.

In the same step, the rule $[\bar{l}_1] \longrightarrow []\bar{\bar{l}}_1$ performs. As we will see, the objects derived from object l_1 will be used later to check whether or not the rule $ca_j[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n X]$ was executed. Moreover, they will be also used to introduce in region 2 objects $A_1 \dots A_n S_1 \dots S_n$ that forbids a new addition or subtraction of objects a_j .

Let us consider the first case, i.e. the region 1 contains at least one object a_j . Then, as a consequence of executing the above rules we will have the multisets $w_1 = a_1^{k_1} \dots a_j^{k_j-1} \dots a_n^{k_n} c$ and $w_2 = A_1^2 \dots A_n^2 S_1^2 \dots S_n^2 X K$. The following rules will be further applied:

$$\bar{\bar{l}}_1[] \longrightarrow [\bar{\bar{l}}_1 T_0 A_1 \dots A_n S_1 \dots S_n],$$

$$[K] \longrightarrow []\bar{K},$$

and possibly the rule:

$$[X] \longrightarrow []\bar{X}.$$

Remark that the objects derived from \bar{l}_1 are within the scope of rules that introduce at each odd step objects $A_1 \dots A_n S_1 \dots S_n$ (or $A_1 \dots A_{j-1} A_{j+1} \dots A_n S_1 \dots S_n$ in the first step). In a similar manner the objects derived from K are within the scope of rules that introduce at each even step objects $A_1 \dots A_n S_1 \dots S_n$. Anyway, at each step we delete by rules $A_i \rightarrow \lambda$ and $S_i \rightarrow \lambda$, $1 \leq i \leq n$ all objects A_i and S_i .

Now, since in the third step an object \bar{X} was introduced in region 1 then, in the fourth step, the rule

$$[\bar{\bar{l}}_1] \neg X \longrightarrow []\tilde{l}_3$$

cannot be executed. Moreover, because in region 2 exists an object T_0 also the rule $\bar{X}[\neg T_0] \longrightarrow [l_2]$ cannot be executed. However, in the fourth step the rule $[T_0] \longrightarrow []T_1$ runs and it will allow, in the fifth step, the execution of the rule $\bar{X}[\neg T_0] \longrightarrow [l_2]$. In the same time, rule $[\bar{\bar{l}}_1] \neg \bar{K} \longrightarrow []\lambda$ is executed and so

there will be no way to rewrite $\overline{\overline{l_1}}$ into \tilde{l}_3 and furthermore into l_3 . Finally, by rule $[l_2] \longrightarrow []l_2L_2$ the label of the new register machine instruction to be simulated is generated.

Now let us see what how the simulation is done when the system Π attempts to simulate the instruction $(l_1 : \text{SUB}(j), l_2, l_3) \in \mathcal{P}$ in the case when the register j is empty. Then, the simulation works in a similar manner as in the above presented case with the main difference being that in the fourth step the rule $\overline{\overline{[l_1]}\neg X} \longrightarrow []\tilde{l}_3$ is executed because the object \overline{X} was not produced (the rules $ca_j[\neg S_j] \longrightarrow [A_1 \dots A_n S_1 \dots S_n X]$ and $[X] \longrightarrow []\overline{X}$ were not ran since the object a_j was missing from the initial multiset). So, the following rules are executed in sequence $\tilde{l}_3[] \longrightarrow \tilde{[l_3]}$, $\tilde{[l_3]} \longrightarrow []l_3L_3$. As a consequence, the symbol that corresponds to the next instruction to be simulated is generated.

In this way we simulate the execution of the entire register machine program. Consequently, we have that $PsPCC_2(cat, inhR_1) \supseteq PsRE$.

Therefore, we have proved the equality $PsPCC_2(cat, inhR_1) = PsRE$. \square

6.2 String Driven Computation

In this section we investigate a model of membrane systems using promoted rules, where a string of promoters (called control string) travels through the regions, activating by its left most symbol certain rules of the system. This control string comes from “outside” the system – it is inserted into the skin region at the beginning of the computation and it is consumed symbol by symbol while traveling through the system.

In this way, the inserted string drives the computation of the membrane system by controlling the activation of evolution rules.

When the control string is entirely consumed and no rules can be applied anymore, then the system halts – this corresponds to a successful computation. The multiset of objects present in the output region is the result of such a computation. In this way, a set of control strings (a control program) generates a family of multisets. We also consider another sort of successful computation, which additionally has to satisfy a “clean ending condition” (which requires that an a priori specified “dirty” object is not present in any region upon the completion of the computation).

We will research the influence of the structure of control programs on the generating power. We demonstrate that the different structures that we consider yield

the generative power ranging from finite to recursively enumerable sets.

We consider two different ways (modes) for a control string to travel through the regions of the system: either the string must move at each step (mode (1)), or it can remain in the same region for several consecutive steps until it decides (non-deterministically) to move again (mode (2)). We prove that, under certain conditions, these two modes are equivalent as far as the generative power is concerned.

A string-controlled P system as described above is formally defined as follows.

Definition 6.2.1 *A string-controlled P system (in short, an SC P system), of degree $m \geq 1$, with symbol-objects is a construct*

$$\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0),$$

where:

- V is the alphabet of Π ; its elements are called objects; $\#$ is a special object always present in V ;
- $C \subseteq V$ is the set of catalysts;
- P is the set of promoters; $P \cap V = \emptyset$;
- $L \subseteq P^+$ is the control program (each string in L is a control string);
- μ is a membrane structure consisting of m membranes labeled $1, 2, \dots, m$;
- $w_i, 1 \leq i \leq m$, are strings that represent the multisets over V associated with the regions $1, 2, \dots, m$ of μ ;
- $R_i, 1 \leq i \leq m$, are finite sets of evolution rules associated with the regions i of μ ; an evolution rule can be non-cooperative, $a \rightarrow v, a \rightarrow v|_p$, catalytic $ca \rightarrow cv, ca \rightarrow cv|_p$, or cooperative, $u \rightarrow v, u \rightarrow v|_p$, where $a \in (V - C), c \in C, p \in P, u \in V^+$, and v is a string over V_{tar} , with $V_{tar} = (V - C) \times TAR$, for $TAR = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$; the target is indicated as index of an object and if no target is specified that is intended to be here;
- i_0 is a number between 1 and m that specifies the output region of Π .

Notions like *membrane structure*, *skin membrane*, *non-elementary membrane*, and *system configuration* (initial or not) are defined as usual in the P systems framework.

An SC P system starts the computation from the initial configuration and with one of the strings from the language L chosen non-deterministically, present in the skin membrane of Π (this string is used as *control string*).

The string moves, in a non-deterministic way, across the regions of the system during the computation. We can distinguish two possible modes for moving the string: (1) at each step the string moves passing from one region to an adjacent one; (2) at each step the string can move to an adjacent region or can remain in the same region. In both cases the string cannot be moved into the environment and when it moves from a region to another one, it loses its leftmost symbol.

At each step the *leftmost symbol of the string is used as a promoter* for the rules present in the region where the string is present. A promoted rule is *active* if the necessary promoter is present. The rules without promoters are active in each step. A transition between two configurations of the P system is governed by the application in a maximally parallel manner of the active rules.

The computation *halts* when there are no more rules applicable in any region of the system and the control string was entirely consumed.

An halting computation is considered *successful* if the SC P system *collects the result in a standard way* (we call it standard, since it is the usual way to define successful computations in the P systems area).

If the SC P system *collect the result in the # way*, then an halting computation is *successful* if only if the special object $\#$ is not present in any region of the system in the halting configuration.

Given a successful computation, we consider the number of objects present in the output region, in the halting configuration, as the result of the computation.

Collecting all the numbers obtained, for any possible successful computation, for each string in the language L used as control string, we get the set $N_{\#}^i(\Pi)$, $i \in \{(1), (2)\}$, of natural numbers generated by the system Π working in the mode i and collecting the result in the $\#$ way ($\#$ is removed if the result is collected in the standard way).

We will use the notation $N_{SC}P_{m\#}^{(i)}(\alpha, FL)$, where $i = 1, 2$, $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, and FL is a class of languages, for denoting the family of sets of natural numbers generated by SC P systems, which use at most m membranes, evolution rules (promoted or not) that can be non-cooperative (*ncoo*), cooperative (*coo*) or catalytic using at most k catalysts, the control program is taken in the class FL and the systems work in the mode (i). The prefix (*pro*) is added if only promoted rules are used (the systems then are called *fully-promoted*) and $\#$ is present if the

result is collected in the $\#$ way or is removed if the result is collected in the standard way.

It is clear that, by definition, the following inclusions hold

Lemma 6.2.1

$$\begin{aligned}
& (pro)N_{SC}P_m^{(i)}(\alpha, FL) \subseteq (pro)N_{SC}P_{m\#}^{(i)}(\alpha, FL). \\
& (pro)N_{SC}P_m^{(i)}(\alpha, FL_1) \subseteq (pro)N_{SC}P_{m\#}^{(i)}(\alpha, FL_2), \text{ if } FL_1 \subseteq FL_2. \\
& (pro)N_{SC}P_{m\#}^{(i)}(ncoo, FL) \subseteq (pro)N_{SC}P_{m\#}^{(i)}(cat_j, FL) \\
& \subseteq (pro)N_{SC}P_{m\#}^{(i)}(cat_{j+1}, FL) \subseteq (pro)N_{SC}P_{m\#}^{(i)}(coo, FL), \text{ for } j \geq 1, i \in \{1, 2\}, \\
& \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}, FL, FL_1, FL_2 \text{ families of languages.}
\end{aligned}$$

It is possible to prove that, under certain conditions, a fully-promoted SC P system working in mode (2) [(1)] can be simulated by a fully-promoted SC P system working in mode (1) [(2), resp.] using the same type of rules, the same type of initial language and using a number of membranes that is doubled.

Theorem 6.2.1 *Given the fully-promoted SC P system Π having rules of type α , a control program from a family FL closed under non-erasing regular substitution, m membranes, there exists a fully-promoted SC P system Π' having rules of type α , $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, a control program from FL and $2m$ membranes such that $N_{\#}^{(1)}(\Pi') = \{x + 1 \mid x \in N_{\#}^{(2)}(\Pi)\}$.*

Proof. Given an SC P system $\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$ we construct the SC P system $\Pi' = (V', C, P', L', \mu', w'_1, \dots, w'_{2m}, R'_1, \dots, R'_{2m}, i_0)$ such that $N_{\#}^{(1)}(\Pi') = \{x + 1 \mid x \in N_{\#}^{(2)}(\Pi)\}$.

We take $V' = V \cup \{Z\}$ with $Z \notin V$ and $P' = P \cup \{d\}$ with $d \notin P$. We construct the regular substitution defined as $\phi(p) = p(dp)^*$ for each $p \in P$; we take $L' = \phi(L)$ (notice the substitution is non-erasing and so every class in $\{REG, CF, CS, RE\}$ is closed under this operation). The structure μ' has $2m$ membranes and is obtained from μ in the following way. In each region $i, 1 \leq i \leq m$, of μ add an elementary membrane with label $m + i$. Furthermore we fix $w'_i = w_i Z, 1 \leq i \leq m, w'_i = Z, m + 1 \leq i \leq 2m$.

The evolution rules for Π' are fixed in the following way. $R'_i = R_i \cup \{Z \rightarrow \#|_d\}$ for $1 \leq i \leq m$ and $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$ for $m + 1 \leq i \leq 2m$. We use the “dummy” promoter d and m “dummy” regions labelled as $m + i, \dots, 2m$.

Every computation in Π working in mode (2) can be simulated by a computation in Π' working in mode (1). In fact, take a computation of the system Π . Now, suppose the current control string has the promoter p as leftmost symbol and is in region i of Π . Suppose that the string does not move (Π works in mode (2)) but remains in the same region also for the next step. Then this is simulated in the system Π' , working in mode (1), by moving between region i and the adjacent dummy region $m + i$ and consuming the dummy promoter d present in the control string; this process can be iterated an arbitrary number of times, simulating in this way the non-deterministic staying of the control string in the region i of the system Π .

On the other hand, except the computations that simulate the computations in Π , there are no other successful computations in Π' . In fact, the presence of the rules $Z \rightarrow \#|_d$ in R'_i , for $1 \leq i \leq m$, guarantees that the dummy symbol cannot be used to lose one step by moving into adjacent non dummy regions, otherwise the computation would be not successful. Moreover, if the promoter present immediately to the right of the leftmost symbol of the current control string is non dummy (i.e., the head of the string is of type $pq \dots$, with $p, q \in P$), then the string must move in a non dummy region, because of the rules $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$ for $m+1 \leq i \leq 2m$ that would make the computation unsuccessful, if applied. \square

It is possible to show also the reverse inclusion. A fully-promoted SC P system working in mode (1) can be simulated, under certain conditions, by a fully-promoted SC P system working in mode (2) using a structure having a double number of membranes.

Theorem 6.2.2 *Given a fully-promoted SC P system Π having rules of type α , control program from a family FL closed under non-erasing morphism, m membranes, there exists an SC P system Π' having rules of type α , $\alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 1\}$, control program from FL , $2m$ membranes, such that $N_{\#}^{(2)}(\Pi') = \{x + 2 \mid x \in N_{\#}^{(1)}(\Pi)\}$.*

Proof. Given the fully-promoted SC P system

$$\Pi = (V, C, P, L, \mu, w_1, \dots, w_m, R_1, \dots, R_m, i_0)$$

we construct the fully-promoted SC P system

$$\Pi' = (V', C, P', L', \mu', w'_1, \dots, w'_{2m}, R'_1, \dots, R'_{2m}, i_0)$$

such that $N_{\#}^{(2)}(\Pi') = \{x + 2 \mid x \in N_{\#}^{(1)}(\Pi)\}$. We take $V' = V \cup \{c, c', Z\}$ with $c, c', Z \notin V$. The new set of promoters is $P' = P \cup \{d, d'\}$ with $d, d' \notin P$. We use the non-erasing morphism $\phi(p) = pdd'$ for each $p \in P$; we take $L' = \phi(L)$ (notice that every class in $\{FIN, REG, CF, CS, RE\}$ is closed under this operation). The structure μ' has $2m$ membranes and is obtained from μ in the following way. In each region $i, 1 \leq i \leq m$, of μ add an elementary membrane with label $m + i$.

Furthermore, we fix $w'_i = cZw_i$ for $1 \leq i \leq m$, and $w'_i = Z$ for $m + 1 \leq i \leq 2m$.

The evolution rules of the system Π' are taken in the following way. $R'_i = R_i \cup \{c' \rightarrow c|_{d'}, Z \rightarrow \#|_d\} \cup \{c' \rightarrow \#|_p, c \rightarrow c'_p \mid p \in P\}$ for $1 \leq i \leq m$. $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$ for $m + 1 \leq i \leq 2m$. We use the “dummy” promoters d, d' and m “dummy” regions with labels $m + i, \dots, 2m$.

Every computation in Π working in mode (1) can be simulated by a computation in Π' working in mode (2) and there are no computations of Π' working in mode (2) that are successful and that do not correspond to computations in the system Π working in mode (1).

In fact, take a computation of the system Π . Suppose that, during the computation, the control string $p_{i_1}p_{i_2} \cdots p_{i_j}$ with $p_{i_1}, p_{i_2}, \dots, p_{i_j} \in P$ is present in region i of Π at step k .

Then, the string $p_{i_1}dd'p_{i_2}dd' \cdots p_{i_j}dd'$ will be present in region i of Π' at some step k' (at the beginning the region i is the skin region).

In the system Π , at step $k + 1$, the string must exit (Π works in mode (1)) from region i , entering in one of the adjacent regions, chosen non-deterministically, losing the promoter p_{i_1} and having as leftmost symbol the promoter p_{i_2} .

This movement is simulated in Π' in the following consecutive steps. The rules activated by promoter p_{i_1} present in region i of Π' are executed at step k' , together with the rule $c \rightarrow c'$ present in every region of Π' and activated by any promoter of P . Therefore, at step k' the control string must exit region i otherwise in the next step the rule $c' \rightarrow \#|_{p_{i_1}}$ would be applied and the entire computation would not be successful.

The only region where the string can go is the dummy region $m + i$ present inside region i (otherwise the promoter d that follows p_{i_1} would activate the rule $Z \rightarrow \#|_d$ present in any of the non dummy adjacent regions of region i and the computation would not be successful). Therefore, suppose the string goes to region $m + i$, losing in this way the promoter p_{i_1} ; in region $m + i$ the string can remain an unbounded number of steps (nothing can be applied there); at certain step k'' the string comes back to region i , losing the promoter d and having now as leftmost symbol the

promoter d' ; therefore, in the step $k'' + 1$ the rule $c' \rightarrow c|_{d'}$ is applied. The string having now as leftmost symbol d' can remain an unbounded number of steps in region i (nothing can be applied). Eventually the string exits region i moving in one of the adjacent regions, losing the promoter d' , and having as leftmost symbol the promoter p_{i_2} that is the next non dummy promoter. In the same way, if the promoter present immediately to the right of the leftmost symbol of the current control string is non dummy (i.e., it is from the set P), then the string must move in a non dummy region, because of the rules $R'_i = \{Z \rightarrow \#|_p \mid p \in P\}$ for $m + 1 \leq i \leq 2m$ would make the computation not successful, if applied.

So, the movements of the string in Π have been correctly simulated in the system Π' ; in this way every successful computation of Π can be simulated by Π' and in Π' there are no other successful computations except the ones corresponding to successful computations in Π . \square

As expected, if the control program can be an arbitrary RE language, then the class of fully-promoted SC P systems is universal, even using only non-cooperative rules.

Theorem 6.2.3 $(pro)N_{SC}P_{2\#}^{(1)}(ncoo, RE) = (pro)N_{SC}P_2^{(1)}(ncoo, RE) = NRE$.

Proof. Given an RE language L over the alphabet $\Sigma = \{a\}$, we modify each word in L by inserting at the beginning and between each two symbols the symbol $*$ $\notin \Sigma$. The language obtained in this way is called L' . Now we construct the SC P system

$$\Pi = (V, C, P, L', \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = \{a'\}$;
- $C = \emptyset$;
- $P = \{a\}$;
- $\mu = [[]_2]_1$;
- $w_1 = \lambda; w_2 = a'$;
- $R_1 = \emptyset$;

- $R_2 = \{a' \rightarrow (a')_{out}a'|_a\}$;
- $i_0 = 1$.

At the beginning of the computation one of the strings in L' , chosen in a non-deterministically manner, is inserted in the skin membrane of Π . The string moves between region 1 and region 2 of the system. When the string moves from region 1 to region 2 a symbol $*$ is removed. When the string moves in the opposite direction a symbol from Σ is removed. When the string is in region 2, its leftmost symbol a activates exactly the rule that produces and send out the symbol a' . Therefore when the computation halts (the string is entirely consumed) the output region contains exactly the number of symbols of the string inserted, without counting the symbols $*$. Therefore, collecting all the numbers obtained for all computations we obtain that the SC P system generates exactly the length set of the language L . \square

After seeing the previous result it is rather natural to ask what happens if we increase the power of the evolution rules used by the P system but we decrease the power of the control program.

A first result is obtained when the P system uses cooperative evolution rules but the control program is finite.

Theorem 6.2.4 $(pro)N_{SC}P_{* \#}^{(1)}(coo, FIN) = (pro)N_{SC}P_*^{(1)}(coo, FIN) = NFIN$.

Proof. Given an SC P system Π it is sufficient to notice that the number of distinct non-deterministic computations, using only a finite number of steps, is bounded by a constant related to the P system Π . Therefore, considering each possible string in the finite control program, the set of numbers produced is still finite.

On the other hand, the length set of every finite language can be generated by a fully-promoted SC P system, using the construction given in Theorem 6.2.3. \square

In case of fully-promoted SC P systems the situation appears more complicate and universality can be obtained by using a regular control program with one catalyst.

Theorem 6.2.5 $(pro)N_{SC}P_{2 \#}^{(1)}(cat_1, REG) = NRE$.

Proof. One direction of the proof follows from Church-Turing thesis. The other direction can be proved by simulating regularly controlled grammar with appearance checking, known to be universal.

Given an arbitrarily regularly controlled grammar with appearance checking $G = (N, T, S, P, K, F)$, we construct the fully-promoted SC P system Π that simulates the grammar G as follows:

$$\Pi = (V, C, P', L, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

- $V = N \cup T \cup \{c, Z\}$;
- $C = \{c\}$;
- $P' = \text{lab}(P) \cup \{d, d'\}, d, d' \notin \text{lab}(P)$;
- $L = \phi(K)dd'$;
- $\mu = [[]_2]_1$;
- $w_1 = \lambda; w_2 = SZc$;
- $R_1 = \emptyset$;
- $R_2 = \{cA \rightarrow c\alpha|_p \mid p : A \rightarrow \alpha \in P\} \cup \{cZ \rightarrow c\#|_p \mid p \notin F\} \cup \{Z \rightarrow Z_{out}|_{d'}\}$;
- $i_0 = 2$;

with the non-erasing morphism ϕ defined as $\phi(p) = dp$ for each $p \in \text{lab}(P)$.

The SC P system works in the following way. In region 2, that is the output region, the derivations of the grammar G are simulated; these derivations are driven by the strings present in the language L . If in language K there is the string $p_{i_1} \cdots p_{i_k}$, then in L there will be the string $dp_{i_1} \cdots dp_{i_k}dd'$. The promoters d are dummies, they are used only to let the string enter and exit from region 2, passing in this way from a promoter $p_{i_j} \in \text{lab}(P)$, to the next one $p_{i_{j+1}}$, for $1 \leq j \leq k - 1$. Only at most one rule in region 2 is executed when the string comes in that region because of the catalyst c that inhibits the parallelism. If a rule cannot be applied and its label is not in the appearance checking set F , then the computation is trashed ($\#$ is produced by applying the rule $cZ \rightarrow C\#$ that is activated by any promoter $p \in (\text{lab}(P) - F)$) and this is correct since the derivation in the grammar G cannot be continued. If the rule cannot be applied and its label is in F (so the rule has to be used in the appearance checking mode), then no rule is applied in region 2, the string leaves this region and the computation can continue. The last promoter

d' present for any string in L is used to move, at the end of the computation, the symbol Z in region 1. From the above description it is clear that the set of natural numbers generated by the system Π is exactly the length set of the language $L(G)$ and so the theorem is true. \square

6.3 Open Problems and Forthcoming Research

In this chapter we have introduced and investigated two new models of P systems: P systems with external promoters/inhibitors and string driven P systems. For both of them several problems have been left open.

For example, here we did not study the computational capabilities of P systems with external promoters. Nevertheless, we conjecture that they generate the same family of sets of vectors as P systems with external inhibitors (when non-cooperative, or catalytic rules are used).

Moreover, we did not approach the problem from a complexity point of view, for instance, establishing the minimum number of external promoters/inhibitors such that the P systems with catalytic rules and external promoters/inhibitors is still able to generate/accept $PsRE$.

On the other hand, we did not answer to the question regarding non-fully promoted IS P systems. Are they more powerful than fully-promoted IS P Systems? The answer is positive only for IS P systems working in mode (1) and having a finite set of initial strings.

Chapter 7

Promoters and Inhibitors in the P System Framework

P systems with promoters/inhibitors prove to be helpful when attempting to solve other problems within the general framework of P systems. Their ability to express natural interactive processes in an intuitive manner makes such systems a powerful tool when attacking new problems and a base for comparison.

In this chapter we present several results regarding computational capabilities of P systems with strong priorities among rules and we show their equivalence in terms of generated families of sets of vectors with P systems with inhibitors. Consequently, we give an answer to the open problem number 6 from [66].

7.1 P Systems with Strong Priorities

The model of P systems with priorities was initially used to describe the biochemical reactions occurring in the cell. There, priority relations (in the form of a partial order relation) among the rules from each region expressed the following phenomenon: if a biochemical reaction r_1 is more active than a reaction r_2 and it consumes a given resource (energy, for example) from the region, then the reaction r_2 cannot take place despite the availability of all necessary input objects.

In the attempt to make use of these features to design new bio-inspired computational devices, the current trend was to decrease as much as possible the level of cooperation between the objects participating into the rules while maintaining the currently obtained results. The mathematical interest was also the opposite problem, namely to see which is the upper bound of cooperation such that the systems are not anymore universal, knowing that almost for all variants of P systems the

universality results were reached.

This brings us to the topic of the section – it determines the computational power of the classical P system model with strong priorities when only non-cooperative rules are used.

We denote by $PsOP_m(ncoo, \alpha)$, $\alpha \in \{pri, inhR_i\}$, the family of sets of vectors of numbers, computed by P systems of degree at most m , $m \geq 1$, using non-cooperative rules and priorities among rules ($\alpha = pri$) or inhibitors of weight i at the level of rules ($\alpha = inhR_i$).

In Chapter 3 it was proved that P systems with non-cooperative rules and inhibitors at the level of rules generates exactly the $PsET0L$, the family of Parikh images of ET0L languages.

In [66] it was shown in a straightforward manner that $PsOP_1(ncoo, pri) \supseteq PsET0L$ by simulating with a constructed P system with non-cooperative rules and priorities the computation of an arbitrary two table ET0L system H . In addition, catalytic P systems with priorities proved to be universal when only one catalyst is used. The remaining open problem ($Q2$ in [66]) was whether or not non-cooperative systems with priorities are universal. In this section we deal with this problem and the answer is that P systems with priorities in their generative power equal the class $PsET0L$.

The following lemma shows that P systems with non-cooperative rules and priorities, having only one membrane, equal in computational power the ones having the same features, but with $m > 1$ membranes.

Lemma 7.1.1 $PsOP_m(ncoo, pri) = PsOP_1(ncoo, pri), m \geq 1$.

Proof. The inclusion $PsOP_m(ncoo, pri) \supseteq PsOP_1(ncoo, pri)$ is trivial. In what concerns the proof of the inclusion $PsOP_m(ncoo, pri) \subseteq PsOP_1(ncoo, pri)$, we will construct a P system $\Pi_1 = (V, C, \mu, w, R, i_0)$ that simulates the computation of P system $\overline{\Pi}_m = (\overline{V}, \overline{C}, \overline{\mu}, \overline{w}_1, \dots, \overline{w}_m, \overline{R}_1, \dots, \overline{R}_m, \overline{i}_0)$ in the following way.

First, denote by $\mathcal{L} = \{1, 2, \dots, m\}$ the set of labels of the regions in $\overline{\Pi}_m$. Then, we define:

- $V = \{a_i \mid a \in \overline{V}, i \in \mathcal{L}\}$.
- $C = \overline{C} = \emptyset$;

Let $h : \overline{V}^* \times \mathcal{L} \rightarrow V^*$ be a mapping such that

- 1) $h(a, i) = a_i, a \in \overline{V}, i \in \mathcal{L}$;
- 2) $h(\lambda, j) = \lambda, j \in \mathcal{L}$;
- 3) $h(x_1 x_2, j) = h(x_1, j) h(x_2, j), x_1, x_2 \in \overline{V}^*, j \in \mathcal{L}$.

- denote by $w = h(\overline{w_1})h(\overline{w_2}) \dots h(\overline{w_m})$, where $\overline{w_i}$ is the multiset present in region $i \in \mathcal{L}$ of $\overline{\Pi_m}$ at the beginning of the computation.
- R is defined as follows. For each rule $a \rightarrow \alpha \in \overline{R_i}$, $a \in \overline{V}$, α is a string over $\{c, c_{out}, c_{in} \mid c \in \overline{V}\}$, $i \in \mathcal{L}$, we add to R the rule $h(a, i) \rightarrow \alpha'$ where α' is the corresponding string over $\{h(c, i), h(c, j), h(c, k) \mid c \in \overline{V}, i, j, k \in \mathcal{L}\}$, j being the label of the outer region of i , and k being the label of an inner region of i . In addition, we inherit the existing priority relations among the rules.
- $i_0 = 1$.

In other words, for the P system with a single region that simulates a P system with m regions, we have encoded the regions labels into objects (the subscript associated to an object indicates the region where the corresponding object belongs) and we have expressed the rules of regions by the corresponding encoded objects. In this way we ensured that, when simulating $\overline{\Pi_m}$ with Π_1 , both the parallelism at the level of regions and at the level of whole system $\overline{\Pi_m}$ is respected. In addition, one can remark that whenever $\overline{\Pi_m}$ halts, Π_1 halts as well. Moreover, when Π_1 halts, we will have in the output region of Π_1 all the objects corresponding to the multisets present in all regions of $\overline{\Pi_m}$.

However, in the output multiset w_{Π_1} of Π_1 we can distinguish the output multiset $w_{\overline{\Pi_m}}$ of $\overline{\Pi_m}$ because we know which are the objects corresponding to the output region of $\overline{\Pi_m}$ (they are the objects that have as index $\overline{i_0}$). Therefore, we have to delete the unnecessary objects that remain in the output region of Π_1 in a halting configuration since we want to show that Π_1 and $\overline{\Pi_m}$ generate exactly the same set of vectors of numbers. We will modify the rules presented above in the following manner.

We add to the vocabulary V a new symbol D (the object D stands for the “deletion command”) and we replace each rule $a_i \rightarrow \alpha' \in R$ by

$$a_i \rightarrow \alpha' D \in R,$$

of course, maintaining the priority relations among the rules. In addition, we add the following rules (with the corresponding priority relation)

$$\boxed{D \rightarrow \lambda} > \boxed{a_i \rightarrow \lambda}, \text{ for all } a_i \in V, i \neq \overline{i_0}.$$

One can remark that in this way we produce at each computational step at least one object D and also, in the same time, we delete the already existing object(s) D . If there exist rules that can be executed (i.e., there will be objects D) rules of type

$a_i \rightarrow \lambda$ cannot be applied because they are locked according to the priority relations. When the computation halts, objects D are not produced anymore, and so, the deletion rules can start and erase the remaining unnecessary objects. Consequently, we have shown that both systems generate the same family of vectors of natural numbers, hence we have $PsOP_m(ncoo, pri) \subseteq PsOP_1(ncoo, pri)$.

Consequently, we have that $PsOP_m(ncoo, pri) = PsOP_1(ncoo, pri)$. \square

As a consequence of the above result we can state the following result.

Corollary 7.1.1 *For any P system Π with non-cooperative rules, using priorities, there exists an equivalent P system Π' with non-cooperative rules, using priorities and with the same membrane structure such that, for any halting configuration of Π' , all regions of Π' , excepting the output one, are empty.*

Now, we can prove the following result that shows the equality between the classes of sets of vectors generated by P systems using strong priorities among non-cooperative rules and by extended tabled Lindenmayer systems, respectively:

Theorem 7.1.1 $PsOP_m(ncoo, pri) = PsOP_m(ncoo, inhR_1) = PsET0L$.

Proof. In [66] it was shown that $PsOP(ncoo, pri) \supseteq PsET0L$ while in Chapter 3 we have shown that $PsOP(ncoo, inhR_1) = PsET0L$. Now we will show that $PsOP(ncoo, inhR_1) \supseteq PsOP(ncoo, pri)$ and hence, $PsOP(ncoo, pri) = PsET0L$. Here is how we proceed.

Let us consider an arbitrary P system $\tilde{\Pi}$ with m membranes, non-cooperative rules and with a priority relations among rules. According to Lemma 7.1.1 we know that we can construct an equivalent P system $\bar{\Pi} = (\bar{V}, \bar{C}, \bar{\mu}, \bar{w}, \bar{R}, \bar{i}_0)$ where:

- $\bar{V} = \{X_1, X_2, \dots, X_r\}$;
- $\bar{C} = \emptyset$;
- $\bar{\mu} = []_1$;
- $\bar{w} \in \bar{V}^*$;
- The set \bar{R} is defined by the sequences of rules:

$$\begin{array}{c} \boxed{X_{(1,1)} \rightarrow \alpha_{(1,1)}} > \boxed{X_{(1,2)} \rightarrow \alpha_{(1,2)}} > \dots > \boxed{X_{(1,k_1)} \rightarrow \alpha_{(1,k_1)}} \\ \dots \\ \boxed{X_{(p,1)} \rightarrow \alpha_{(p,1)}} > \boxed{X_{(p,2)} \rightarrow \alpha_{(p,2)}} > \dots > \boxed{X_{(p,k_p)} \rightarrow \alpha_{(p,k_p)}} \end{array}$$

with $X_{(i,j)} \in \bar{V}$, such that $X_{(i,j_1)} \neq X_{(i,j_2)}$, for all $j_1 \neq j_2$, $1 \leq i \leq p$, and $\alpha_{(i,j)} \in \bar{V}^*$, $1 \leq i \leq p$, $1 \leq j \leq k_i$. In addition, without loosing the generality, we will assume that $k_1 \geq k_2 \geq \dots \geq k_p$.

Recall that we assumed that $X_{(i,j_1)} \neq X_{(i,j_2)}$, for all $j_1 \neq j_2$, $1 \leq i \leq p$, because in case $X_{(i,j_1)} = X_{(i,j_2)}$, the rule $X_{(i,j_2)} \rightarrow \alpha_{(i,j_2)}$ will never be applied since the rule $X_{(i,j_1)} \rightarrow \alpha_{(i,j_1)}$, having a grater priority, is applied first (of course, if it fulfills all required conditions).

We construct a P system $\Pi = (V, C, \mu, w, R, i_0)$ with non-cooperative inhibited rules that simulates the moves of $\bar{\Pi}$ and which is defined as follows.

- $V = \bar{V} \cup \{\bar{X} \mid X \in \bar{V}\} \cup \{A_{(i,j)}, U_{(i,j)} \mid 1 \leq i \leq p, 1 \leq j \leq k_i\}$
 $\cup \{S, T, H, \#\} \cup \{W_i \mid 1 \leq i \leq k_1 + 1\};$
- $C = \emptyset;$
- $\mu = []_1;$
- $w = STH\bar{w};$

- The set of rules R is defined as follows:

◊ we add to R the rules:

$$\begin{aligned}
X_i &\rightarrow \bar{X}_i T A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \quad 1 \leq i \leq k, \\
S &\rightarrow U_{(1,0)} U_{(2,0)} \dots U_{(p,0)} W T H A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \\
W &\rightarrow W_1 T H A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \\
W_1 &\rightarrow W_2 T H A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \\
&\dots \\
W_{k_1} &\rightarrow W_{k_1+1} T H A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \\
W_{k_1+1} &\rightarrow S H, \\
T &\rightarrow \lambda, \\
A_{(i,j)} &\rightarrow \lambda, \quad 1 \leq i \leq p, 1 \leq j \leq k_i.
\end{aligned}$$

◊ for each sequence of rules belonging to \bar{R} :

$$\boxed{X_{(i,1)} \rightarrow \alpha_{(i,1)}} > \boxed{X_{(i,2)} \rightarrow \alpha_{(i,2)}} > \dots > \boxed{X_{(i,k_i)} \rightarrow \alpha_{(i,k_i)}}$$

we add to R the rules:

$$\begin{aligned}
U_{(i,0)} &\rightarrow U_{(i,1)} |_{\neg \overline{X_{(1,1)}}}, \\
U_{(i,1)} &\rightarrow U_{(i,2)} |_{\neg \overline{X_{(1,2)}}}, \\
&\dots \\
U_{(i,k_i)} &\rightarrow U_{(i,k_i+1)} |_{\neg \overline{X_{(1,k_1)}}}, \\
\\
U_{(i,0)} &\rightarrow A_{(i,2)} A_{(i,3)} \dots A_{(i,r)} |_{-T}, \\
U_{(i,1)} &\rightarrow A_{(i,1)} A_{(i,3)} A_{(i,4)} \dots A_{(i,r)} |_{-T}, \\
&\dots \\
U_{(i,k_i)} &\rightarrow A_{(i,1)} \dots A_{(i,k_1)} A_{(i,k_1+2)} \dots A_{(i,r)} |_{-T}, \\
U_{(i,k_i+1)} &\rightarrow A_{(i,1)} \dots A_{(i,r)} |_{-T}, \\
\\
\overline{X_{(i,j)}} &\rightarrow \alpha_{(i,j)} |_{-A_{(i,j)}}, \quad 1 \leq j \leq k_i.
\end{aligned}$$

◇ also, we add the rules:

$$\begin{aligned}
S &\rightarrow \lambda, \\
X_i &\rightarrow \# |_{-H} \text{ iff there exists a rule } X_i \rightarrow \alpha_i \in R, \\
\# &\rightarrow \#, \\
H &\rightarrow \lambda, \\
\overline{X_{(i,j)}} &\rightarrow X_{(i,j)} |_{-H}.
\end{aligned}$$

Let us see how the simulation works. First, observe that (as a general technique) when we want to execute a certain non-cooperative rule r at a certain moment during the computation, then we might activate it using an inhibitor; however, this means that all the time during the computation we have to generate the symbol representing the inhibitor, to delete at each step all previously created inhibitors, and only when we actually want to execute r we omit its generation.

We start the computation by executing the rule:

$$X_i \rightarrow \overline{X_i} T A_{(1,1)} A_{(1,2)} \dots A_{(1,k_1)} \dots \dots A_{(p,1)} \dots A_{(p,k_p)}, \quad 1 \leq i \leq k.$$

This rule is responsible for “painting” all objects X_i that correspond to objects in \overline{V} . In the same time we create the objects:

$$A_{(1,1)}, A_{(1,2)}, \dots, A_{(1,k_1)}, \dots \dots, A_{(p,1)}, \dots, A_{(p,k_p)},$$

$1 \leq i \leq k$, that will be used as “flags”, indicating which rules cannot be applied (here the simulation of any rule from $\overline{\Pi}$ is forbidden – all objects are present). In addition, we create the object T that represents as well a flag, its role being to indicate when the selected rules will be effectively applied.

In the same time, the rule

$$S \rightarrow U_{(1,0)}U_{(2,0)} \dots U_{(k,0)}WTHA_{(1,1)}A_{(1,2)} \dots A_{(1,k_1)} \dots A_{(p,1)} \dots A_{(p,k_p)}$$

is executed. All objects $U_{(i,0)}$, $1 \leq i \leq p$, represent the starting points for the sequences of rules of type:

$$\begin{aligned} U_{(i,0)} &\rightarrow U_{(i,1)}|_{\overline{-X_{(1,1)}}}, \\ U_{(i,1)} &\rightarrow U_{(i,2)}|_{\overline{-X_{(1,2)}}}, \\ &\dots \\ U_{(i,k_i)} &\rightarrow U_{(i,k_i+1)}|_{\overline{-X_{(1,k_1)}}}. \end{aligned}$$

Such a sequence (that corresponds to $X_{(i,1)} \rightarrow \alpha_{(i,1)} > \dots > X_{(i,k_i)} \rightarrow \alpha_{(i,k_i)} \in \overline{R}$) is used to check which rules from \overline{R} can be applied. Depending where this sequence stops we will know what rules we have to apply. This information will be stored in the objects $U_{(i,j)}$.

Remark that along with objects $U_{(i,0)}$, $1 \leq i \leq p$, the object W is produced. This object will be used by the cycle (let us call it the “waiting” cycle):

$$\begin{aligned} W &\rightarrow W_1THA_{(1,1)}A_{(1,2)} \dots A_{(1,k_1)} \dots A_{(p,1)} \dots A_{(p,k_p)}, \\ W_1 &\rightarrow W_2THA_{(1,1)}A_{(1,2)} \dots A_{(1,k_1)} \dots A_{(p,1)} \dots A_{(p,k_p)}, \\ &\dots \\ W_{k_1} &\rightarrow W_{k_1+1}THA_{(1,1)}A_{(1,2)} \dots A_{(1,k_1)} \dots A_{(p,1)} \dots A_{(p,k_p)}, \\ W_{k_1+1} &\rightarrow SH, \end{aligned}$$

which produces “enough” time (more than the maximum length of the sequences of rules in \overline{R}) objects $A_{(1,1)}A_{(1,2)} \dots A_{(1,k_1)} \dots A_{(p,1)} \dots A_{(p,k_p)}$ which forbids the application of any rule that corresponds to a rule in \overline{R} . In the last step of the cycle we omit the creation of object T . The absence of object T means that we can apply one of the rules:

$$\begin{aligned} U_{(i,0)} &\rightarrow A_{(i,2)}A_{(i,3)} \dots A_{(i,r)}|_{-T} \\ U_{(i,1)} &\rightarrow A_{(i,1)}A_{(i,3)}A_{(i,4)} \dots A_{(i,r)}|_{-T} \\ &\dots \\ U_{(i,k_i)} &\rightarrow A_{(i,1)} \dots A_{(i,k_1)}A_{(i,k_1+2)} \dots A_{(i,r)}|_{-T} \\ U_{(i,k_i+1)} &\rightarrow A_{(i,1)} \dots A_{(i,r)}|_{-T} \end{aligned}$$

In this way we are able to select which are the rules (that corresponds to rules in \overline{R}) that can be applied, namely:

$$\overline{X_{(i,j)}} \rightarrow \alpha_{(i,j)}|_{-A_{(i,j)}}, \quad 1 \leq j \leq k_i.$$

Now, observe that all the time the “waiting” cycle is active (that is, we intend to make a simulation of a step in $\bar{\Pi}$) the object H is created. Also, the already existing objects H are deleted by the rule $H \rightarrow \lambda$. This object will help us to finish the simulation. Here are the details.

Non-deterministically, object S might also be deleted by the rule $S \rightarrow \lambda$. If this happen, then the object H is not produced anymore and so, the rules $\overline{X_{(i,j)}} \rightarrow X_{(i,j)}|_{-H}$ and $X_{(i,j)} \rightarrow \#|_{-H}$ are executed. So, basically, if symbol $\#$ appears, then the computation will not stop because the rule $\# \rightarrow \#$ will be always executed.

In case the symbol $\#$ is not produced then the computation eventually stops if the computation of $\bar{\Pi}$ stops. This is due to the fact that the cycle involving object S might be always executed. However, the system Π will generate in a non-deterministic manner (if object S is deleted and there is no symbol $\#$) the same language as $\bar{\Pi}$. Consequently, the families of languages generated by these types of systems are equal.

Therefore, we have that $PsOP_m(ncoo, pri) = PsOP_m(ncoo, inhR_1) = PsETOL$.

□

The constructive equality with the class of Parikh images of ETOL languages gives us ”for free” all decidability results known for the family of ETOL languages. For example, it is of a mathematical interest (but not only) to mention that reachability and membership problems for ETOL systems are decidable.

7.2 Open Problems and Forthcoming Research

In this chapter we gave an answer to the open problem regarding the computational power of P systems with strong priorities when non-cooperative rules are used.

We believe that the model of P systems with promoters/inhibitors can be used to tackle problems involving polarizations of membranes, concentrations of objects as well as many other models where the rewriting process is controlled by various conditions. The important gain in showing such results stands on the existence not only of decidability results, but also of the many properties that characterize the family *ETOL*.

Chapter 8

Conclusions

The last twenty years in computer science research were mainly under the auspices of natural influence. Scientist were trying to imitate the way in which nature computes, conceiving new paradigms and computing models from it.

Membrane computing is a recent branch of natural computing, emergent in the field of unconventional models of computation, inspired by the dynamical processes taking place in living cells. Proposed in 1998 by Gheorghe Păun in the paper *Computing with membranes*, the membrane computing paradigm has known immediately a remarkable development. However, several open problems still remained unsolved since the moment when they were brought under the scientific focus.

The main question posed in this thesis was whether we can mimic nature's achievement, creating artificial devices that perform features such as those manifest by their natural counterparts. Obviously, this outstanding goal is yet far off, however, we hope to have taken a small step toward it.

The present thesis covered a range of topics that includes the study of computational capabilities of the classical model of P systems with promoters/inhibitors, the investigation of several bio-motivated generalizations of the model, the design of new computational devices and algorithms. In our endeavor several open problems were solved, and many others arose. In the final sections of previous chapters we have outlined several topics for future research, hence it is not our intention to enter the details again.

Here we will emphasize the main investigations that are presented in this thesis. The achieved results (including the drawbacks) are discussed at an informal level.

We have started our scientific attempt by first briefly introducing the reader to some aspects of theoretical computer science regarding languages and their generative/accepting formal devices. We continue by first presenting the syntax and the

semantics of P systems with promoters/inhibitors and afterward by expressing their computational capabilities when several constraints are imposed.

We were interested here in the computational power of the mentioned P systems with respect to classical families of languages such as regular, context-free, extended tabled Lindemayer languages, recursively enumerable sets and so on. We succeeded to prove universality results (in both accepting/generating cases) for catalytic P systems with promoters/inhibitors of weight one by simulating register machines and regularly controlled grammars with appearing checking, respectively. The same computational power is also obtained for P systems with promoters/inhibitors of weight two, when only non-cooperative rules are used. In addition, such systems have the ultimately confluence property as it was defined in the preliminaries of the thesis.

For the membrane systems only using non-cooperative rules and inhibitors we show their equivalence with the family of Parikh images of languages generated by *ETOL*. We consider this to be very interesting since even if both computational systems share many common features, their halting procedure differs. Moreover, having this outcome, many other results can be inherited from the vast literature regarding *ETOL* systems. Decidability results as well as other properties become available in an easy fashion through this new result.

In what concern membrane systems only using non-cooperative rules and promoters we have sketched a possible proof of their equivalence family of Parikh images of languages generated by *ETOL*. This is of interest especially because for the sequential case there was not found any relation between the classes of languages generated by random context grammars with permitting and with forbidding contexts, respectively.

Customary in the P systems definitions is the existence of a universal clock (rules are executed in one computational step in exactly one time unit) based on which various processes can be synchronized in order to attain a certain goal (compute a function, simulate a classical device, etc). In Chapter 4 we are interested in systems which do their computations independently of the time factors. We proposed timed variants for mentioned P systems by associating a time values to the executions/applications of the rules in a computational step. We succeeded to obtain a subclass of such P systems where each member is able to compute regardless the time values chosen. We have then defined the concepts of time-freeness and clock-freeness according to the way the rules perform the rewriting. Such systems are of interest also because, in a certain sense, they are error tolerant against rules

executions times faulty factors.

Classical P systems theory assumes a maximal parallel rewriting method when defining computations. Here, we were also interested in systems that perform their computation using a variable parallelism for the rewriting. To this aim, we have first introduced and studied the Lindenmayer systems operating in the M-rate mode of derivation. Essentially, we have associated to each rule from any table a multifunction that, based on a given configuration, establishes the number of times a rule is applied (of course with competition on the objects with other rules if it is the case). We have defined two types of variable parallel mode of derivation depending if we impose or not that the rules are applied at least once in one computational step (of course, if possible): the strong and the weak mode. For the strong mode of derivation we succeeded to show that ETOL systems working in the strong mode of derivation generate exactly the family *ETOL*, that is, the maximal parallelism is not fundamental when establishing their generative power. Since P systems share many common features with ETOL systems, we have applied most of the obtained results; hence we have defined M-rate P systems working in the weak and strong mode of derivations. Moreover, we have generalized the definition by considering cooperative rules instead of non-cooperative ones as it was defined for M-rate Lindenmayer systems.

We study the ability of P systems with promoters/inhibitors to solve particular problems. In this respect we deal with P systems solving the static sorting problem. Here, since we have represented the numbers to be sorted as multiplicities of distinct objects and because at the end of computation we wanted a string representing the result of the sorting, we have changed the way we interpret the result of a successful computation – objects are expelled into the environment in the sorted fashion and we take into account the string of their apparition during the whole successful computation. In this respect, we propose several algorithms with better time complexity than the classical sequential ones, all of this despite the disorder among the objects given by the use of multisets.

In addition, by using P systems with promoters/inhibitors we simulate other parallel computing devices namely Boolean circuits. First, we model the Boolean gates and we construct modules that can be assembled into a Boolean circuit. We show that the simulation of a circuit can be done in linear time with respect to the corresponding circuit depth. For Boolean circuits there is a whole literature of algorithms solving various problems, henceforth one can use them, in an intuitive manner, in the P systems framework.

We have considered two new computational models inspired by natural phenomena. *P systems with external promoters/inhibitors* convey the idea of how metabolites pass through a membrane, based on the electrical charge of the destination region. Modeling these phenomena, a natural question arises: what is the computational power of the underlying formal model while preserving the external control feature but decreasing the cooperation of the rules. Here we gave an answer to such problems by showing that when only simple and simple inhibited rules are used we generate exactly *PsETOL*. When using in addition catalytic rules such systems become universal even when just one catalyst is considered. Regarding the computational capabilities of P systems with external promoters, we conjecture that they are able to generate the same family of sets of vectors of numbers as their corresponding (in terms of types of used rules) P systems with external inhibitors.

String controlled P systems embody, in a certain sense, the idea of a stored program. A string of promoters/inhibitors that is placed at the beginning of computation into the skin membrane activates through its left most symbol certain rules from the region; the string travels through membrane structure and at each membrane crossing it loses its left most symbol. We have studied two ways of collecting the result of a successful computation: the string of promoters is exhausted, there are no more rules that can be executed and the result is the multiset of objects in the output region, or, similarly, but considering only the multisets that do not contain a special trap symbol #.

Finally, we show the utility of the P systems with promoters/inhibitors in solving problems in the field of parallel multiset processing. In this respect we give an answer to the problem regarding computational capabilities of P systems with strong priorities.

The potential new directions, continuing and completing the investigated formal models, mentioned at the end of each chapter constitute a basis for further research. We conclude hoping that the present thesis has shed some light on the behavior of P systems and the complex computation they are able to perform.

Computational Results:

1. Theorem 2.2.1: $\lambda RC_{lc} \supset ET0L$.
2. Theorem 3.2.1: $PsOP_m(ncoo, inhR_1) = PsOP_1(ncoo, inhR_1)$.
3. Theorem 3.2.2: $PsOP_m(ncoo, inhR_1) = PsET0L$.
4. Theorem 3.2.3: $PsOP_1(ncoo, proR_1) \supseteq PsET0L$.
5. Theorem 3.2.4: $PsRC_{lc}^\lambda \supseteq PsOP_m(ncoo, proR_1)$.
6. Theorem 3.3.1: $PsOP_m(cat_1, proR_1) = PsRE$.
7. Theorem 3.3.2: $PsOP_2(cat_1, inhR_1) = PsRE$.
8. Theorem 3.3.3: $DPsIOP_2(cat_1, proR_1) = PsRE$.
9. Theorem 3.3.4: $DPsIOP_2(cat_1, inhR_1) = PsRE$.
10. Theorem 3.4.1: $UPsIOP_1(ncoo, proR_2) = PsRE$.
11. Theorem 3.4.2: $UPsIOP_1(ncoo, inhR_2) = PsRE$.
12. Theorem 4.1.1: $PsCOP_m^c(ncoo, \mathbb{N}) = PsCOP_m^c(ncoo, \mathbb{Q}^+)$
 $= PsCOP_m^c(ncoo, \mathbb{R}^+) = PsCF$, for $m \geq 1$.
13. Theorem 4.1.1: $PsTOP_m^t(ncoo) = PsCF$.
14. Theorem 4.1.2: $PsCOP_2^c(cat_1, proR_1, \mathbb{N}) = PsRE$.
15. Theorem 4.1.2: $PsTOP_2^t(cat_1, proR_1) = PsRE$.
16. Theorem 4.1.3: $PsCOP_2(cat_1, proR_1, \mathbb{Q}^+) = PsCOP_2(cat_1, proR_1, \mathbb{R}^+) \supset PsRE$.
17. Theorem 4.1.4: $PsCOP_2^c(cat_1, inhR_1, \mathbb{N}) = PsRE$.
18. Theorem 4.1.3: $PsTOP_2^t(cat_1, inhR_1) = PsRE$.
19. Theorem 4.1.5: $PsCOP_2(cat_1, inhR_1, \mathbb{Q}^+) = PsCOP_2(cat_1, inhR_1, \mathbb{R}^+) \supset PsRE$.
20. Theorem 4.2.3: $ME0L^w = MET0L^w = \mathcal{P}(V^*)$.
21. Theorem 4.2.4: $MET0L^{s,pf} = ET0L$.
22. Theorem 4.2.5: $PsOP_m^{s,pf}(ncoo, inhR_1) = PsET0L$.
23. Theorem 4.2.6: $PsOP_2^{s,pf}(cat_1, inhR_1) = PsOP_2^{s,pf}(cat_1, proR_1) = PsRE$.
24. Theorem 6.1.1: $PsPCC_2(smp, inhR_1) = PsET0L$.
25. Theorem 6.1.2: $PsPCC_2(cat, inhR_1) = PsRE$.
26. Theorem 6.2.3: $(pro)N_{SC}P_{2\#}^{(1)}(ncoo, RE) = (pro)N_{SC}P_2^{(1)}(ncoo, RE) = NRE$.

27. Theorem 6.2.4: $(pro)N_{SC}P_{*\#}^{(1)}(coo, FIN) = (pro)N_{SC}P_*^{(1)}(coo, FIN) = NFIN$.
28. Theorem 6.2.5: $(pro)N_{SC}P_{2\#}^{(1)}(cat_1, REG) = NRE$.
29. Theorem 7.1.1 $PsOP_m(ncoo, pri) = PsOP_m(ncoo, inhR_1) = PsET0L$.

Bibliography

- [1] Adleman L.M., Molecular Computations of Solutions to Combinatorial Problems, *Science*, 226 (1994), 1021-1024.
- [2] Alberts B., Bray D., Lewis J., Raff M., *Molecular Biology of the Cell*, Garland Publishing, New York, 1994.
- [3] Alhazov A., Sburlan D., Static Sorting Algorithms for P Systems, *Applications of P Systems*, (Ciobanu G., Pérez Jiménez M.J., Păun Gh. eds.), Springer-Verlag, Berlin, 2005.
- [4] Alhazov A., Sburlan D., Static Sorting Algorithms for P Systems, *Applications of Membrane Computing* (Ciobanu G., Pérez-Jiménez M., Păun G. Eds.), Springer-Verlag, Berlin, 2005.
- [5] Ardelean I., The Relevance of Biomembranes for P Systems, *Fundamenta Informaticae*, 49, 1-3 (2002), 35–43.
- [6] Ardelean I., Cavaliere M., Sburlan D., Computing Using Signals: From Cells to P Systems, *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9, 9 (2005), 631–639.
- [7] Ardelean I., Besozzi D., Manara C., Aerobic Respiration is a Bio-logic Circuit Containing Molecular Logic Gates, *Pre-proceedings of the Fifth Workshop on Membrane Computing (WMC5)*, Milano, Italy, June 2004, 119–125.
- [8] Arulanandham J.J., Implementing Bead-Sort with P Systems, *Lecture Notes in Computer Science*, 2509, Springer-Verlag, Berlin, 2002, 115-125.
- [9] Atanasiu A., Martín-Vide C., Arithmetic with Membranes, *Romanian Journal of Information Science and Technology*, 4, 1-2 (2001), 5–20.

- [10] Atanasiu A., Martín-Vide C., P Systems and Context-Free Languages, *Actas Primer Congreso Espanol de Algoritmos Evolutivos y Bioinspirados* (Alba E. et al. eds.), Merida, 2002, 341–346.
- [11] Bernardini F., Manca V., Dynamical Aspects of P Systems, *BioSystems*, 70, 2 (2003), 85–93.
- [12] Bernardini F., Gheorghe M., Muniyandi R.C., Pérez Jiménez M.J., Romero Campero F.J., On P Systems as a Modelling Tool for Biological Systems, *Pre-Proceedings of the Sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 193–213.
- [13] Besozzi D., Computational and Modelling Power of P Systems, Ph.D. Thesis, Università degli Studi di Milano, 2003.
- [14] Besozzi D., Ciobanu G., A P System Description of the Sodium-Potassium Pump, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 210–223.
- [15] Besozzi D., Mauri G., Suzuki Y., Tanaka H., Zandron C., Toward a Novel Computational Framework for Molecular Computing: Chemical Reaction as Computation, *Pre-proc DNA10*, Milano, 2004, 455–460.
- [16] Bottoni P., Martín-Vide C., Păun G., Rozenberg G., Membrane Systems with Promoters/Inhibitors, *Acta Informatica*, 38, 10 (2002), 695–720.
- [17] Calude C.S., Păun G., Computing with Cells and Atoms in a Nutshell, *Complexity*, 6, 1 (2000), 38–48.
- [18] Calude C.S., Păun G., *Computing with Cells and Atoms*, Taylor and Francis, London, 2001.
- [19] Cavaliere M., Deufemia V., Further Results on Time-Free P systems, *Proceedings of the ESF Exploratory Workshop on Cellular Computing (Complexity Aspects)*, Seville, Spain, 2005, 95–116.
- [20] Cavaliere M., Ionescu M., Ishdorj T.O., Inhibiting/de-inhibiting Rules in P Systems, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 224–238.
- [21] Cavaliere M., Sburlan D., Time-Independent P Systems, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 239–258.

- [22] Cavaliere M., Sburlan D., Time and Synchronization in Membrane Systems, *Fundamenta Informaticae*, 64, 1-4 (2005), 65–77.
- [23] Cavaliere M., Riscos-Núñez A., Rozenberg G., Sburlan D., String Driven Computation in Membrane Systems, manuscript.
- [24] Cavaliere M., Riscos-Núñez A., Rozenberg G., Sburlan D., P Systems with External Control, manuscript.
- [25] Ceterchi R., Martín-Vide C., P Systems with Communication for Static Sorting, *GRLMC Report*, (Cavaliere M., Martín-Vide C., Păun G., eds.), Rovira i Virgili University, 26, 2003.
- [26] Ceterchi R., Sburlan D., Simulating Boolean Gates with P Systems, *Lecture Notes in Computer Science*, 2933, Springer-Verlag, Berlin, 2003, 104–122.
- [27] Ciobanu G., Păun G., The minimal Parallelism is Still Universal, submitted.
- [28] Csuhaj-Varjú E., Kelemenova A., Kelemen J., Păun G., Eco-Grammar Systems – A Grammatical Framework for Life-Like Interactions. *Artificial Life*, 3 (1997), 1–28.
- [29] Csuhaj-Varjú E., P Automata, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 19–35.
- [30] Dang Z., Ibarra O.H., Yen H.C., On Various Notions of Parallelism in P Systems, *International Journal of Foundations of Computer Science*, 16, 4 (2005), 693–705.
- [31] Dassow J., Păun G., *Regulated Rewriting in Formal Language Theory*, Springer-Verlag, Berlin, 1989.
- [32] Dassow J., Păun G., On the Power of Membrane Computing, *Journal of Universal Computer Science*, 5, 2 (1999), 33–49.
- [33] Ewert S., van der Walt A.P.J., Generating Pictures Using Random Permitting Context, *IJPRAI*, 13, 3 (1999), 339–355.
- [34] Ewert S., van der Walt A.P.J., A Shrinking Lemma for Random Forbidding Context Languages, *Theoretical Computer Science*, 237, 1-2 (2000), 149–158.
- [35] Ewert S., van der Walt A.P.J., A Pumping Lemma for Random Permitting Context Languages, *Theoretical Computer Science*, 270, 1-2 (2002), 959–967.

- [36] Fernau H., Parallel Grammars: A Phenomenology, *Grammars*, 6, 1 (2003), 25–87.
- [37] Ferretti C., Mauri G, Zandron C., Using Membrane Features in P Systems, *Romanian J. of Information Science and Technology*, 4, 1-2 (2001), 241–257.
- [38] Feynman R.P., There's Plenty of Room at the Bottom, *Miniaturization*, (Gilbert D.H. ed.) , Reinhold, New York, 1961, 282–296.
- [39] Freund R., Păun G., On the Number of Non-terminals in Graph-Controlled, Programmed, and Matrix Grammars, *Lecture Notes in Computer Science*, 2055, Springer-Verlag, Berlin, 2001, 214–225.
- [40] Freund R., Kari L., Oswald M., Sosik P., Computationally Universal P Systems without Priorities: Two Catalysts Are Sufficient, *Theoretical Computer Science*, 330, 2 (2005), 251–266.
- [41] Freund R., Asynchronous P Systems and P Systems Working in the Sequential Mode, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 36–62.
- [42] Frisco P., *Theory of Molecular Computing. Splicing and Membrane Computing*. Ph.D. Thesis, IPA (Institute voor Programaturkunde en Algorithmiek), Leiden, 2004.
- [43] Greenlaw R., Hoover H. J., Parallel Computation, in *Algorithms and Theory of Computation Handbook* (M. J. Atallah ed.), CRC Press, 1999.
- [44] Harrison M., *Introduction to Formal Language Theory*, Addison-Wesley, Reading, MA, 1978.
- [45] Head T., Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors, *Bulletin of Mathematical Biology*, 49 (1987), 737–759.
- [46] Hopcroft J.E., Ullman J.D., *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.
- [47] Ionescu M., Sburlan D., On P Systems with Promoters/Inhibitors, *Journal of Universal Computer Science*, 10, 5 (2004), 581–599.

- [48] Ionescu M., Ishdorj T.O., Boolean Circuits and a DNA Algorithm in Membrane Computing, *Pre-Proc. of the sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 410–438.
- [49] Krishna S.N., *Languages of P Systems: Computability and Complexity*, Ph.D. Thesis, Indian Institute of Technology Madras, 2001.
- [50] Krishna S.N., Păun A., Three Universality Results on P Systems, *GRLMC Report*, (Cavaliere M., Martín-Vide C., Păun G., eds.), Rovira i Virgili University, 26, 2003, 198–206.
- [51] Krithivasan K., Prakash V.J., Simulating Boolean Circuits with Tissue P Systems, *Pre-proceedings of the Fifth Workshop on Membrane Computing (WMC5)*, Milano, Italy, June 2004, 343–359
- [52] Leporati A., Mauri G., Zandron C., Simulating the Fredkin Gate with Energy-Based P Systems, *Journal of Universal Computer Science*, 10, 5, (2004), 600–619.
- [53] Lindenmayer A., Mathematical Models for Cellular Interaction in Development, Parts I and II. *Journal of Theoretical Biology*, 18 (1968), 280–315.
- [54] Marcus S., Membranes versus DNA, *Fundamenta Informaticae*, 49, 1–3 (2002), 223–227.
- [55] Minsky M.L., Recursive Unsolvability of Post’s Problem of ‘Tag’ and Other Topics in Theory of Turing Machines., *Ann. Math.*, 74 (1961), 437–455.
- [56] Minsky M.L., *Finite and Infinite Machines*, Prentice Hall, EngleWood Cliffs, New Jersey, 1967.
- [57] Mitrana V., *Bioinformatica*, L&S Infomat, Bucuresti, 2001.
- [58] Obtulowicz A., Păun G., (In Search of) Probabilistic P Systems, *BioSystems*, 70, 2 (2003), 107–121.
- [59] Ogihara M., Ray A., Simulating Boolean Circuits on a DNA Computer, *Technical Report*, 631, 1996.
- [60] Păun G., *Probleme Actuale in Teoria Limbajelor Formale*, Ed. Științifică și Enciclopedică, Bucharest, 1984.

- [61] Păun G., Rozenberg G., Salomaa A., *DNA Computing - New Computing Paradigms*, Springer-Verlag, Berlin, 1998.
- [62] Păun G., Computing with Membranes. An Introduction, *Bulletin of the EATCS*, 67 (1999), 139–152.
- [63] Păun G., Yu S., On Synchronization in P Systems, *Fundamenta Informaticae*, 38, 4 (1999), 397–410.
- [64] Păun G., Computing with Membranes, *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143.
- [65] Păun G., Computing with Membranes: Attacking NP-Complete Problems. *Unconventional Models of Computation* (Antonioni I., Calude C.S., Dinneen M.J., eds), Springer-Verlag, Berlin, 2000, 94–115.
- [66] Păun G., *Membrane Computing. An Introduction*, Springer-Verlag, Berlin, 2002.
- [67] Păun G., Rozenberg G., A Guide to Membrane Computing, *Theoretical Computer Science*, 287, 1 (2002), 73–100.
- [68] Pelmont J., *Catalyseurs du monde vivant*, Press Universitaires de Grenoble, Grenoble, 1995.
- [69] Pérez Jiménez M.J., Sancho Caparrini F., A Formalization of Transition P Systems, *Fundamenta Informaticae*, 49, 1-3 (2002), 261-272.
- [70] Pérez Jiménez M.J., Romero-Jiménez A., Sancho Caparrini F., Complexity Classes in Models of Cellular Computing with Membranes, *Natural Computing*, 2, 3 (2003), 265-285.
- [71] Pérez Jiménez M.J., Riscos-Núñez A., A Linear Solution to the Knapsack Problem Using Active Membranes, *Lecture Notes in Computer Science*, 2933, Springer-Verlag, Berlin, 2004, 250–268.
- [72] Pérez Jiménez M.J., An Approach to Computational Complexity in Membrane Computing, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 85–109.
- [73] Riscos-Núñez A., *Cellular Programming: Efficient Resolution of NP-complete Numerical Problems*, Ph.D. Thesis, Universidad de Sevilla, 2005.

- [74] Rozenberg G., T0L Systems and Languages, *Information and Control*, 23 (1973), 357–381.
- [75] Rozenberg G., Extension of Tabled E0L Systems, and Languages, *Internat. J. Comput. Inform. Sci.*, 2 (1973), 311–334.
- [76] Rozenberg G., L Systems, Sequences and Languages, *GI JJahrestagung*, 1975, 71–84.
- [77] Rozenberg G., More on ET0L systems versus random context grammars, *Inform. Process. Lett.*, 5, 4 (1976), 102–106.
- [78] Rozenberg G., Salomaa A., *The Mathematical Theory of L Systems*, Academic Press, New York, 1980.
- [79] Rozenberg G., Salomaa A. eds, *Handbook of Formal Languages* (3 volumes), Springer-Verlag, Berlin, 1997.
- [80] Salomaa A., *Formal Languages*, Academic Press, New York, 1973.
- [81] Sburlan D., A Formal Approach to the Minimization of States of UMLs State-chart Diagram, *Bull. PAMM*, 2039 (2002), 65–73.
- [82] Sburlan D., A Static Sorting Algorithm for P Systems with Mobile Catalysts, *Analele Stiintifice Univ. Ovidius Constanța*, seria Matematica, 11, 1 (2003), 195–205.
- [83] Sburlan D., New Results on P Systems with Multiset Promoted/Inhibited Rules, *Bull. PAMM*, 2164 (2004), 45–54.
- [84] Sburlan D., From Cells to Digital Signal Processing, *Proceedings of International Workshop on Mathematical Modelling of Environmental and Life Science Problems*, Romanian Academy of Sciences, Romania, 2004, 269–278.
- [85] Sburlan D., Clock-free P Systems, *Pre-Proceedings of the Fifth International Workshop on Membrane Computing (WMC5)*, Milan, Italy, 2004, 372–383.
- [86] Sburlan D., Modeling The Dynamical Parallelism of Bio-Systems. *M-Rate Systems*, *Pre-Proceedings of the Sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 517–529.

- [87] Sburlan D., Non-cooperative P systems with Priorities Characterize PsETOL, *Pre-Proceedings of the Sixth Workshop on Membrane Computing (WMC6)*, Vienna, Austria, 2005, 530–539.
- [88] Sburlan D., Further Results On P Systems with Promoters/Inhibitors, accepted *International Journal of Foundations of Computer Science*.
- [89] Shepherdson J. C., Sturgis H. E., Computability of Recursive Functions, *J. Assoc. Comput. Mach.* 10 (1963), 217–255.
- [90] Sipper M., Studying Artificial Life Using a Simple, General Cellular Model, *Artificial Life Journal*, 2, 1 (1995), 1–35.
- [91] Stamatopoulou I., Gheorghe M., Kefalas P., Modelling Dynamic Organization of Biology-Inspired Multi-agent Systems with Communicating X-Machines and Population P Systems, *Lecture Notes in Computer Science*, 3365, Springer-Verlag, Berlin, 2005, 389–403.
- [92] Suzuki Y., Tanaka H., Modeling p53 Signaling Pathways by Using Multi-set Processing, *Applications of Membrane Computing*, Springer-Verlag, Berlin, 2005, 201–214.
- [93] Șerbanai L.D., *Programming Languages and Compilers*, Romanian Academy Press, 1987.
- [94] van der Walt A.P.J., Random Context Languages, *IFIP Congress*, 1971, 66–68.
- [95] Wang L., Sun Y.P., Chen W.L., Li J.H., Zhang C.C., *FEMS Microbiol. Lett.*, 213 (2002), 155–165.
- [96] White D., *The Physiology and Biochemistry of Prokaryotes*, Oxford University Press, 2nd edition, 2000.
- [97] Zandron C., *A Model for Molecular Computing: Membrane Systems*, Ph.D. Thesis, Università degli Studi di Milano, 2001.
- [98] The P Systems Web Page: <http://psystems.disco.unimib.it/>