
UNIVERSITY OF SEVILLE
Dpt. Computer Science and
Artificial Intelligence

# Membrane Computing, Neural Inspirations, Gene Assembly in Ciliates

Thesis presented by
Tseren-Onolt Ishdorj
to obtain the PhD degree
from the University of Seville

Tseren-Onolt Ishdorj

Approval of the Supervisors of the Thesis

Dr. Gheorghe Păun

Dr. Mario de J. Pérez Jiménez

Seville, 21.12.06

UNIVERSIDAD DE SEVILLA
Dpto. de Ciencias de la Computación
e Inteligencia Artificial

# Membrane Computing, Neural Inspirations, Gene Assembly in Ciliates

Memoria presentada por
Tseren-Onolt Ishdorj
para optar al grado de Doctor
por la Universidad de Sevilla

Tseren-Onolt Ishdorj

V.º B.º Los Directores de la Tesis

Dr. D. Gheorghe Păun

Dr. D. Mario de J. Pérez Jiménez

Sevilla, 21 de diciembre 2006

# Acknowledgments

I am very proud of natural computing community, especially, of P-friends community, for their achievements and for the scientific framework in which I was privileged to work during elaborating this thesis.

My heartfelt thanks to my supervisors Dr. Gheorghe Păun and Prof. Dr. Mario de J. Pérez-Jiménez for their fundamental encouragement to learn science and for their fatherlike support for my social life too.

My warmest thanks to Dr. Ion Petre for his precious support and help.

Thanks to all the members of the Research Group on Natural Computing of the University of Sevilla, Spain, and all the members of the Computational Biomodelling Laboratory, Åbo Akademi University, Finland, who have provided me such wonderful friendly and scientific working environment.

I am much indebted to my co-authors and co-workers for their fruitful collaboration. Special regards to Artiom Alhazov, Matteo Cavaliere, Mihai Ionescu, Linqiang Pan, Vladimir Rogojin, Miguel Angel Gutiérrez-Naranjo, Agustin Riscos-Núñez, Fernando Sancho-Caparrini, Dragoş Sburlan, Francisco Jose Romero-Campero, Alvaro Romero-Jiménez, Badmaanyambuu Dorj, and other friends.

I would like to thank my dad Ishdorj and mom Miishaa. Thanks to my family for their continuous support for making this work possible during these years.

To my loves: wife Baigalmaa
son Tsedev-Ochir
daughter Gangadari

# Contents

# An Introduction to the Thesis

The thesis background belongs to two major branches of natural computing: cellular computing and molecular computing. More precisely, the research is mainly devoted to model (neural) cell biological phenomena in membrane computing area, and to investigate the computability and the complexity issues related to membrane computing models and to certain models of gene assembly in ciliates.

Natural computing is a general term referring to computing going on in nature and to computing inspired by nature, which provides valuable insights into both natural sciences and computer science. Cellular computing is theoretically and practically a significant branch of natural computing as, for instance, artificial neural networks and cellular automata were well developed both mathematically and used in practical applications. In the framework of the continuous development of the research of computing with cells, a flourishing direction was initiated in [79] under the name of membrane computing. The progresses of membrane computing research were impressive (already, in 2003, ISI considered this research area as "fast emerging" in computer science). Membrane systems (also called P systems) are inspired by the compartmental structure and the functioning of living cells. The result is a non-deterministic, distributed and parallel computing model. Very shortly, we have a membrane compartmentalized system, in the regions of which multisets of objects evolve according to some specified rules. We will give more detailed definitions of P systems in Section 2.3.

A natural question in membrane computing concerns the possibility to define computing models inspired by neural processes, hence, to link this area with the neural computing. On the one side, the human brain is an example of a massively parallel system. As neural systems and membrane systems have in common many features of parallel and distributed processes, they could support each other in devising computing models. On the other side, intuitively, incorporating neural computing ingredients is challenging and promising because most of the computing models, for instance, finite automata, which were defined with neural inspiration have known very suc-

cessful applications in computing.

The main part of the thesis is basically focused on elementary neural processes, in particular, on exploring symbolic models of neural cell processes through membrane systems.

Chapter 6 relates membrane computing with another very active research area, that of DNA computing – here we consider the very interesting case of gene assembly in ciliates.

## Content of the Thesis

As any kind of computer, be it mechanical, electrical or biological, needs two basic capabilities in order to function, to store information and to perform operations on it, these capabilities were always considered in our biological computing devices throughout the thesis.

Moreover, after defining a computing model, we always investigate its computational power in comparison with Turing machines and other classic models of computing, and also we investigate its computational efficiency, checking whether it can solve computationally hard problems in feasible time.

The thesis contains an introduction and six chapters. In **Chapter 1**, we briefly mention the historical development of theoretical computer science, starting from G. Leibnitz's (1646–1716) wish to formalize human reasoning in mathematics, A. Turing's (1912–1954) fundamental mathematical model of a computer, and its implementation to the electronic computer by von Neumann (1903–1957). We also give a short informal survey of the exciting computing models of natural computing, especially the two branches considered in the thesis: cellular computing – membrane systems, and molecular computing – DNA computing. Some of the cellular and molecular biology facts are mentioned in the end of **Chapter 1** because those biological motivations are frequently recalled in the following chapters when we formalize bio-inspired operations.

We start in **Chapter 2** with the mathematical definitions, notions, notations of formal languages, computability and computational complexity theory, and at the end of the chapter we give the formal definitions of membrane systems and of basic operations of gene assembly in ciliates, which are the main topics of the thesis. In **Chapter 3**, we formalize some cell interaction operations in P systems, namely membrane merging, membrane separation, membrane release, and chemicals replication and distribution. We investigate the computational power and efficiency of the introduced bio-operations and universality and also efficiency results are proved. This shows that *membrane separation* operation is an efficient tool to achieve an exponential workspace in linear time. Also, we find that the dual operations of membrane merging and membrane separation are useful tools to compute

with cell membranes. Again, both universality and efficiency results are proved both in the maximally and the minimally parallel way of using the rules.

The most essential part of the thesis is focused on "zooming" inside a neural cell in order to find as biologically well motivated, mathematically elegant, and computationally powerful and efficient devices as possible in P systems area. **Chapter 4** and **Chapter 5** are mainly devoted to this direction of research. There is a well known phenomenon in neural biology, the *exciter impulse* and *inhibitor impulse*. We formalized exciting/inhibition rules in P systems as a natural mechanism to control computations. Boolean circuits have been simulated in this framework. Then, in **Chapter 5** we introduce a class of spiking neural P systems called axon P systems. This time we investigate the language generative power of axon P systems in comparison with Chomsky hierarchy. Moreover, a preliminary attempt to consider the computational efficiency of spiking neural P systems is presented here, based on the idea of using a pre-computed arbitrarily large resource. A solution to SAT is proposed in this framework.

We pass to the next major topic of the thesis in **Chapter 6**. Computing in ciliates has been studied since 1998 when Laura Landweber and Lila Kari introduced their intermolecular computing model with ciliates in [58]. There are two computing models in this area, the intermolecular gene rearrangement and the intramolecular gene rearrangement. The contextual intermolecular model has been shown to have the computational power of Turing machines, [55]. The intramolecular gene rearrangement model has waited for clarifying its computational power. We have an answer to this question for a mathematical model based on intramolecular recombination which is shown to be equivalent in power to Turing machines. Similarly, the computational efficiency of both models has not been considered yet so far. At the end of the thesis we present a preliminary approach in this respect, with an example of SAT problem solution.

Each of Chapters 3–6 contains a series of open problems and topics for further research.

The chapters of the thesis are based on the next publications, elaborated by the author with the fruitful collaboration of several co-authors.

**Chapter 3:**

1. A. Alhazov, T.-O. Ishdorj, Membrane Operations in P Systems with Active Membranes. In [85], 37–44.

2. T.-O. Ishdorj, Power and Efficiency of Minimal Parallelism in Polarizationless P Systems. *Journal of Automata, Languages, and Combinatorics*, to appear.

3. T.-O. Ishdorj, Minimal Parallelism for Polarizationless P Systems.

*Proc. 12th Int. Meeting on DNA Computing*, Seoul, June 2006, (C. Mao, T. Yokomori, Eds.), *LNCS 4287*, Springer, Berlin, 2006, 17–32.

4. L. Pan, A. Alhazov, T.-O. Ishdorj, Further Remarks on P Systems with Active Membranes, Separation, Merging and Release Rules. *Soft Computing. A Fusion of Foundations, Methodologies and Applications*, 9 (2005), 686–690.

5. L. Pan, T.-O. Ishdorj, P Systems with Active Membranes and Separation Rules. *Journal of Universal Computer Science*, 10 (5) (2004), 630–649.

**Chapter 4:**

6. M. Cavaliere, M. Ionescu, T.-O. Ishdorj, Inhibiting/De-inhibiting Rules in P Systems. *LNCS 3365*, Springer, Berlin, 2005, 224–238.

7. M. Cavaliere, M. Ionescu, T.-O. Ishdorj, Inhibiting/De-inhibiting Rules in P Systems with Active Membranes. In *Proc. Cellular Computing (Complexity Aspects) Workshop* (M.A. Gutiérrez-Naranjo et. al, Eds.), Sevilla, 2005, 117–130, Fénix Editora, Sevilla, 2005.

8. M. Ionescu, T.-O. Ishdorj, Boolean Circuits and a DNA Algorithm in Membrane Computing. In *Proc. 6th WMC*, Vienna, 18–23 July 2005, and *LNCS 3850*, Springer, Berlin, 2006, 274–293.

9. T.-O. Ishdorj, M. Ionescu, Replicative-Distribution Rules in P Systems with Active Membranes. *Pre-proc. First International Colloquium on Theoretical Aspects of Computing*, Guiyang, China, Sept. 20-24, 2004 - UNU/IIST Report No. 310, 263–278, Zhiming Liu (Ed.) Macau, and *LNCS 3407*, Springer, Berlin, 2005, 69–84.

**Chapter 5:**

10. H. Chen, M. Ionescu, T.-O. Ishdorj, On the Efficiency of Spiking Neural P Systems. *Proc. 8th Int. Conference on Electronics, Information, and Communication*, Ulaanbaatar, Mongolia, June 2006, 49–52.

11. H. Chen, T.-O. Ishdorj, Gh. Păun, Computing Along the Axon. *Preproc. Int. Conference on Bio-inspired Computing: Theory and Applications (BIC-TA)*, September 2006, Wuhan, China, 60–70.

**Chapter 6:**

12. T.-O. Ishdorj, I. Petre, R. Vladimir, Computational Power of Intramolecular Gene Assembly. Manuscript, 2006.

13. T.-O. Ishdorj, I. Petre, An Efficient Computing Paradigm Inspired by Gene Assembly in Ciliates. Manuscript, 2006.

# Chapter 1

# Theoretical Computer Science: A Historical Perspective

The classic notion of computation is firmly rooted in the notion of an algorithm, which, informally speaking, is a set of rules for performing a task. The German mathematician Gottfried W. Leibnitz (1646–1716) wanted to formalize human reasoning in such a way that it could be described as a collection of rules, which then could be executed in a mechanistic way.

This research into the understanding of the notion of an algorithm was carried on by many outstanding scientists and it culminated in the works of Kurt Gödel, Alonzo Church, Alan Turing, and Emil Post approximately in the period 1930–1940.

The work ([103]) for the formalization of the notion of an algorithm by the English mathematician Alan Turing (1912–1954) led to the construction of the first computers, and to the beginnings of computer science. In formalizing the notion of an algorithm, Turing has focused on what a person performing calculations does when following a set of rules, hence following a given algorithm. Thus the beginnings of computer science were rooted in human-designed computing.

John von Neumann (1903–1957) introduced 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.

In the history of computing, electronic computers are only the latest in a long chain of man's attempts to use the best technology available for doing computations. While it is true that their appearance, some decades ago, has revolutionized computing, computing does not start with electronic computers, and there is no reason why it should end with them. Indeed, even electronic computers have their limitations in what concerns the amount of the data they can store and their speed thresholds, both these aspects being determined by physical barriers which will soon be reached.

A promising direction in theoretical computer science for the next generation of computing devices is *natural computing*, a research area more and more active in the last decades. Natural computing is a fast growing field of interdisciplinary research driven by the idea that natural processes can be used for implementing computations, constructing new computing devices, and to get inspirations for new computational paradigms.

The current understanding of this research area includes fields like *quantum Computing*, *evolutionary computing*, *neural networks*, *molecular* and *cellular computing*. Quantum computing uses quantum parallelism to perform computations; evolutionary computing uses the concepts of mutation, recombination, and natural selection from biology to design new algorithms and new ways of performing computations; neural networks construct computational paradigms inspired by the highly interconnected neural structures in the brain and in the nervous system.

Molecular and cellular computing (background topics of this thesis) construct and investigate computational models inspired by molecular processes that can be either artificially created in laboratory or naturally happening in living organisms, like, for instance, the biochemical reactions present in living cells.

## 1.1 Natural Computing: Cellular and Molecular Computing

*Computing with cells* is a considerably old field in computer science theory.

The first cellular computing model inspired in the functioning of the brain, in the way its elements (neurons) are connected by a net in an appropriate way (nervous system), was created in 1943 by Warren S. McCulloch and Walter Pitts, MIT, within the paper "A logical calculus of the ideas immanent in nervous activity" [66]; afterward it led to automata theory, [57]. The model consists of a network of simple processors, connected by communication channels, in such a way that they only operate over local data and data received via such channels. Additionally, these models include a training and learning procedure.

The next fundamental impact into cellular computing is due to John von Neumann who in 1947 introduced the notion of cellular automata (CA)

based on generalizations of multicellular interactions, in an attempt to develop an abstract model of self-reproduction in biology. Cellular automata was designed also to answer the basic question whether 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 (theoretical) answer to this question.

CAs are dynamical systems in which space and time are discrete. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local, identical interaction rule. CAs exhibit three notable features, namely, massive parallelism, locality of cellular interactions, and simplicity of basic components (cells). Thus, they present an excellent point of departure for our forays into parallel cellular machines.

In 1968, the biologist Aristid Lindenmayer (1925–1989), introduced a beautiful mathematical model of development of *multicellular* organisms, [60]. The development happens in parallel everywhere in the organism. A living system may consist of millions of parts, all having their own characteristic behavior. However, although a living system is highly distributed, it is massively parallel. Therefore, *parallelism* is a built-in characteristic of Lindenmayer (L) systems. These basic ideas gave rise to an abundance of language-theoretic problems, both mathematically challenging and interesting from the point of view of diverse applications.

Those biological cell-inspired computing models consider the cell as an atomic unit and only the interactions/communications between cells play an essential role for computing, not also their internal structure.

Three decades after A. Lindemnayer's work, Gheorghe Păun, observing the structure and the behavior of living cells, where chemical compounds are processed in a massively parallel manner inside a compartmental structure of cell membranes that control the substance exchange between regions they delimit, proposed an abstract cellular computing model called *membrane system* in his work "Computing with membranes", [79], in 1998. Membrane systems are based on biological *single living cell* compartmental structure and chemical evolution, communication and interaction functions in compartments. The computing model was subsequently named P system. A key structural notion is that of a *membrane* by which a system is divided into compartments where chemical reactions can take place. These reactions transform multisets of objects present in the compartments into new objects, possibly transferring objects to neighboring compartments, including the environment. P systems are a class of distributed parallel computing devices of a biochemical type.

The idea to perform *computations on a molecular and atomic scale* and to construct "sub-microscopic" computers is credited, by many scientists, to R. P. Feynman. Feynman, in his famous talk *"There is plenty of room*

*at the bottom"* given in 1959 at the annual meeting of American Physical Society, proposed to manipulate directly atoms to construct nano-machines (including nano-computing machines). The content of this talk together with many seminal ideas about (what we generally call now) nanotechnology can be found in [35].

Few years later, in 1973, C. Bennett discussed how to perform computations on a molecular scale (see [9]) proposing to use RNA molecules as a physical medium for implementing computations; in a following work [10] Bennett proposed a theoretical model of a Turing machine storing information using RNA molecules and processing them by using (imaginary) enzymes to catalyze biochemical reactions.

A formal model of the recombination of DNA molecules has been introduced by Tom Head [44], in 1987, as the splicing operation under the influence of restriction enzymes (which cut DNA molecules at specific sites) and ligases (which paste together DNA fragments whose sticky ends match).

As von Neumann moved from Turing's abstract computing model to practice, an important achievement in computer science in general and in natural computing in particular was obtained by computer theorist Leonard M. Adleman in 1994, [1], when he surprised the scientific community by using the tools of molecular biology to solve a hard computational problem. Adleman's seminal experiment, solving an instance of the directed *Hamiltonian Path Problem* by manipulating DNA strands, partially accomplishes Feynman's theory in practice. The experiment provoked an avalanche of computer science/molecular biology/biochemistry/physics research, while generating at the same time a multitude of open problems.

The excitement for DNA computing was mainly caused by its capability of massively parallel searches. This, in turn, showed its potential to yield tremendous advantages from the point of view of *speed, energy consumption and density of stored information.* For example, in Adleman's model, [2], a DNA computer could be 1,200,000 times faster than the fastest supercomputer, and about $10^{10}$ times more energy efficient ([1]). A single DNA memory could hold more words than all the computer memories ever made.

DNA computing is concerned with studying and constructing computational components *in vivo* or *in vitro.* The former is concerned with the theoretical foundations and experimental work on building DNA-based computers in test tubes. The latter is concerned with constructing computational components in living cells, and with studying computational processes taking place in living cells (such as simple switching circuits, [70]).

On the other hand, another general possibility consists in investigating computational processes at molecular level that take place in living cells. For instance, a fruitful line of research consists in analyzing the process of gene assembly in *ciliates* (the reader can consult the recent monograph [32] dedicated to this research area). Two models for gene assembly in ciliates have been proposed and investigated in the last few years. The DNA ma-

nipulations postulated in the two models are very different: one model is intramolecular – a single DNA molecule is involved, folding on itself according to various patterns, while the other is intermolecular – two DNA molecules may be involved, hybridizing with each other. Consequently, the assembly strategies predicted by the two models are completely different. Interestingly however, the final result of the assembly (including the assembled gene) is always the same.

The computational nature of gene assembly in ciliates was first pointed out by Lila Kari and Laura Landweber [55], [58], [59] – they noticed that the process of assembling genes resembles the structure of the "molecular solution" of the directed Hamiltonian path problem reported by Adleman in his paper [1]. Kari and Landweber have proposed a model based on intermolecular interactions. Another model of gene assembly, based on intramolecular interactions (different parts of the same molecule interacting with each other) was introduced by Ehrenfencht, Prescott and Rozenberg in [34] and [91]. This research concentrates on the (formal and biological) properties of the process of gene assembly itself.

As the reader can see, molecular and cellular computing have grown in an extremely fast way and have spread in several different directions; our short historical introduction cannot cover all these aspects but the reader can consult the proceedings of the annual meeting on DNA based computers, currently in its twelfth year, or the monographs dedicated to the area, for instance [11, 87], as well as the proceedings of yearly workshops on membrane computing, currently in its eighth year, or the monographs [81, 25] and the web-page [106].

## 1.2 Cells, Membranes and Molecules: An Informal Understanding

In this section we recall some basic elements concerning living cells: cell structure and functions. The reader can use this section as a short reference for the biological arguments used through all the thesis. We also recall some precise biological facts in the following chapters when we give the formalizations of them.

According to the standard cell biology all living things are composed of one or more cells. But what exactly is a cell, and what does it look like? Francis Harry Compton Crick (1916–2004) (together with James Dewey Watson (1928–) he solved the DNA mystery in 1953, [27]), describes it in his own words: The cell is like a bag, with a strong outside to give it strength, and an inside that is a special filter which divides the inside from the outside. It is a very clever filter; whilst many of the molecules that are produced inside the cell, cannot get through the filter, it still allows other molecules to flow in and out. If we look inside this bag, under a powerful

microscope, we notice that in several parts of the cell there are quite a lot of very large, long molecules, and many rather small, thin ones, which looks rather like a long piece of string. There may be several different kinds of each size. So the cell is like a very complicated factory. In it many sorts of work are going on at the same time, with small molecules being changed one into another, and large molecules being built out of small ones. It is astonishing that the cell can, from every simple beginnings, build up this huge variety of molecules of all shapes and sizes. Imagine, then, every cell acting as a very small, highly efficient factory. (Adapted from Crick's book, *Of Molecules and Men*, [26].)

Cells fall into *prokaryotic* and *eukaryotic* types. Prokaryotic cells are smaller (as a general rule) and lack much of the internal compartmentalization and complexity of eukaryotic cells. No matter which type of cell we are considering, all cells have certain features in common: cell membrane, DNA, cytoplasm, and ribosomes.

The shapes of cells are quite varied with some, such as neurons, being longer than they are wide and others, such as parenchyma (a common type of plant cell) and erythrocytes (red blood cells) being equidimensional. Some cells are encased in a rigid wall, which constrains their shape, while others have a flexible cell membrane (and no rigid cell wall).

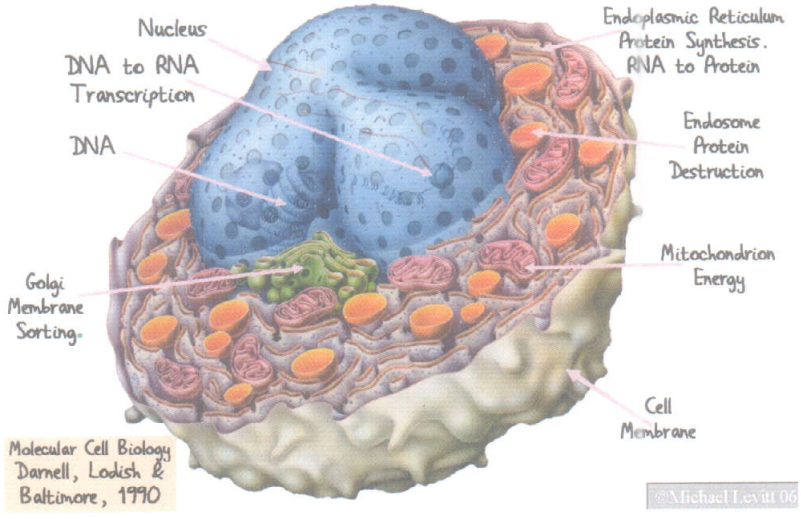



Figure 1.1: Structural architecture of a cell.

The structure and functions of cells are prescribed by genes contained in the DNA molecules of chromosomes. The genetic information in DNA is transcribed into RNA molecules, which in turn are translated into protein molecules. Protein molecules are the principal molecules that prescribe cell structures and functions.

Cells reproduce by cell division. In order to divide, a cell must first increase in size so that two daughter cells of adequate size can be produced. An essential preparation for cell division is the duplication of the chromosomes so that each daughter cell can receive a full diploid set of chromosomes. In other words, each DNA double helix in every chromosome must first be duplicated if a cell is to divide successfully. The duplication of DNA molecules is referred to as DNA replication. The replicated DNA are identical to each other in nucleotide sequences, and therefore contain precisely the same genes, and are identical to the original parental double helix.

Proteins are suspended in the inner layer, although the more hydrophilic areas of these proteins "stick out" into the cells interior and outside of the cell. These proteins function as gateways that, in exchange for a "price", will allow certain molecules to cross into and out of the cell. These integral proteins are sometimes known as gateway proteins. The outer surface of the membrane will tend to be rich in glycolipids, which have their hydrophobic tails embedded in the hydrophobic region of the membrane and their heads exposed outside the cell. The compartments of an animal cell are depicted in Figure 1.1.

We also briefly describe here *the neural cells* structure and functioning which we have chosen, and those features that are of interest from a computational point of view and from which we will abstract the (mathematical) features of our computing devices.

Because this thesis is not deeply connected to biology, but biology is its background, sometimes the reason to introduce models, it might be suitable to describe here the biological phenomenon starting with a mathematical point of view. Several decades ago, John von Neumann considered an approach toward the understanding of the nervous system focusing on the information theory and automata, hence computing machines. He said: "The most immediate observation regarding the nervous system is that its functioning is *prima facie* digital. The basic component of this system is the *nerve cell*, the *neuron*, and the normal function of a neuron is to generate and propagate a *nerve impulse*. This impulse is a rather complex process, which has a variety of aspects – electrical, chemical, and mechanical. It seems, nevertheless, to be a reasonably uniquely defined process, i.e., nearly the same under all conditions; it represents an essentially reproducible, unitary response to a rather wide variety of stimuli. (...) The nerve impulse is a continuous change, propagated – usually at a fixed speed, which may, however, be a function of the nerve cell involved – along the axon. In the area of the axon over which the pulse-potential is passing, the ionic constitution of the intracellular fluid changes, and so do the electrical-chemical properties of the wall of the axon, the *membrane*" (*The Computer and the Brain*, 1958, [69]).

The nervous system consists of about $10^{10}$ nerve cells. Each of these neu-

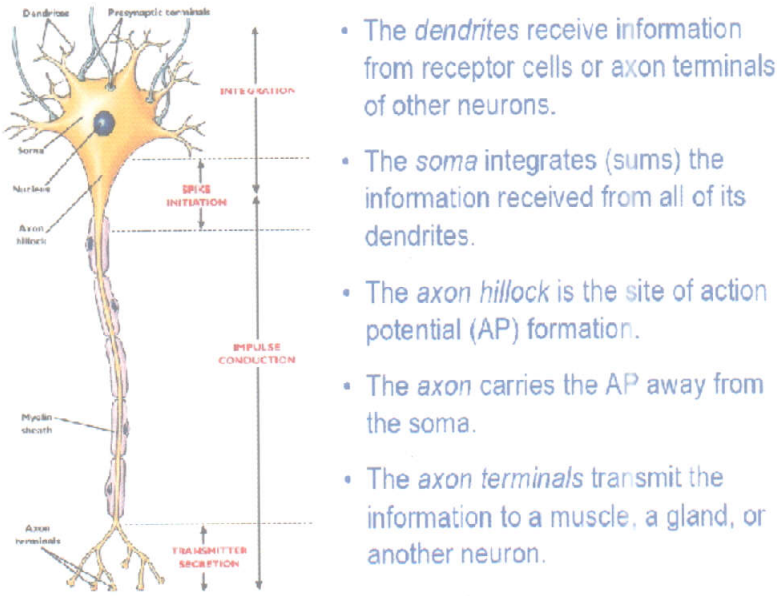



Figure 1.2: A biological neuron structure

rons is connected to about 10000 other neurons via *synapses*. This gigantic network of neurons and synapses is placed in a small portion of the human brain. Neural networks are generally considered as information-processing systems, i.e., as systems which operate transformations of their inputs in order to produce outputs.

Brain cells communicate in a process that begins with an electrical signal and ends with a neurotransmitter binding to a receptor on the receiving neuron. It lasts less than a thousandth of a second, and is repeated billions of times daily in each of the human brain's 100 billion neurons. The main part of the action takes place inside the secreting cell.

The basic unit in the nervous system is the neuron. The neuron is not one homogeneous integrative unit but is (potentially) divided in many sub-integrative units, each one with the ability of mediating a local synaptic output to another cell or local electro-tonic output to another part of the same cell.

Neurons are considered to have 3 main parts: a soma, the main part of the cell where the genetic material is present and life functions take place; a dendrite tree, the branches of the cell from where the impulses come in; an axon, the branch of the neuron over which the impulse (or signal) is propagated. The branches present at the end of the axons are called terminal tree. An axon can be provided by a structure composed by special sheaths. These sheaths are involved in molecular and structural modifications of axons needed to propagate impulse signals rapidly over long distance. The

impulse in effect jumps from node to node, and this form of propagation is therefore called *saltatory conduction.* It is an efficient mechanism that achieves maximum conduction speed with a minimum of active membrane, metabolic machinery, and fiber size. There is a gap between neighboring myelinated regions that is known as the node of Ranvier, which contains a high density of voltage-gated $Na^+$ channels for impulse generation. When the transmitting impulses reach the node of Ranvier or junction nodes of dendrite and terminal trees, or the end bulbs of the trees, it causes the change in polarization of the membrane. The change in potential can be excitatory (moving the potential toward the threshold) or inhibitory (moving the potential away from the threshold). In Figure 1.2, neuron structure is represented.

The impulse transmission through a neuron follows this path: from dendrite to soma to axon to terminal tree, and then to synapse. If different impulses reach at the same time a certain node, then it might happen that the combined effects of the *excitation and inhibition may cancel each other out.* Once the threshold of the membrane potential is reached, an impulse is propagated along the neuron or to the next neuron.

The foregoing account has considered the synapse as a mechanism for unidirectional transmission between cells, as envisaged in the classical studies. The complexity revealed by modern research has widened this view to include several new concepts. *First,* the presynaptic terminal may have receptors for its own released products; thus, the presynaptic terminal may be postsynaptic as well to its own transmitter. *Second,* the postsynaptic terminal may send retrograde signals to the presynaptic terminal. These may be rapid signals. By these actions, the synapse can be viewed as having a *bidirectional nature. Third,* it is obvious that the synapse is not a simple link between two neurons, but rather is a complex organelle in its own right. In most synapses the direct cause of change in potential is not electrical but chemical: the electrical pulse reaches the endbulb and causes the *released* of transmitter molecules from little packets (vesicles) through the synaptic membrane.

The transmitter diffuses across the cleft to act on its postsynaptic receptors. Diffusion is controlled by the cleft extracellular matrix. The concentration of the transmitter is brought back to basal levels by diffusion, and in some cases by hydrolysis or reuptake. Neuropeptides may diffuse over long distances. Some of the previously mentioned ideas will be incorporated (abstracted) in our computing model, investigated in Chapters 4 and 5. More details about neural biology, can be found in the classical book [101], and about information processes taking place along the axon in [100].

*Ciliates* form a diverse group of unicellular organisms – some 8000 species are currently known and many others are likely to exist – that are found practically in all environments containing water. Their diversity can be

appreciated by comparing their genomic sequences: some ciliate types differ genetically more than humans differ from fruit flies!

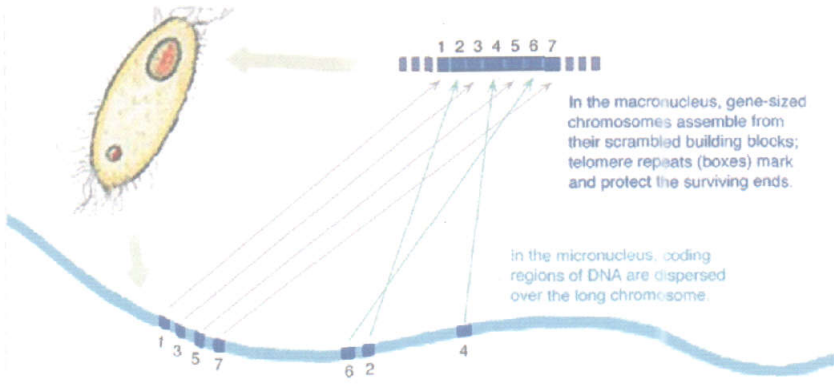



Figure 1.3: Overview of gene unscrambling,
from `http://www.princeton.edu/~ lfl/`

Two characteristics unify ciliates as a single group: the possession of hairlike cilia used for motility and food capture, and the presence of two kinds of functionally different nuclei in the same cell, a micronucleus and a macronucleus; the latter feature is unique to ciliates. The macronucleus is the "household" nucleus – all RNA transcripts are produced in the macronucleus. The micronucleus is a germline nucleus and has no known function in the growth or in the division of the cell. The micronucleus is activated only in the process of sexual reproduction, where at some stage the micronuclear genome gets transformed into the macronuclear genome, while the old macronuclear genome is destroyed. This process is called gene assembly, it is the most involved DNA processing known in living organisms, and it is most spectacular in the Stichotrichs species of ciliates ([43]). An overview of gene unscrambling is presented in Figure 1.3. In Chapter 6, we investigate a computability model based on gene assembly in ciliates.

# Chapter 2

# Preliminaries

## 2.1 Formal Language Preliminaries

This section gives a brief survey of the notions of formal languages, automata theory, and regulated rewriting used in this thesis.

### 2.1.1 Alphabets, Strings, Languages, and Multisets

An *alphabet* is a finite and non-empty set of symbols. Given an alphabet $V$, the free monoid generated by $V$ under the operation of concatenation (defined in the usual sense) is denoted by $V^*$ (it is the set of all strings of symbols from $V$). The empty string is denoted by $\lambda$ (it consists of no symbols); the set of non-empty strings over $V$ is $V^+ = V^* - \{\lambda\}$.

An arbitrary subset of $V^*$ is called a *language* over $V$. A language not containing the empty string is said to be $\lambda - free$.

Given a string $x \in V^*$ such that $x = x_1x_2$, for some $x_1, x_2 \in V^*$, then $x_1, x_2$ are respectively called a *prefix* and a *suffix* of $x$. If $x = x_1x_2x_3$, for some $x_1, x_2, x_3 \in V^*$, then $x_2$ is called a *substring* of $x$. The sets of all prefixes, suffixes, and substrings of a string $x$ are denoted by $Pref(x), Suff(x)$, $Sub(x)$, respectively. The notation $Perm(x)$ indicates the set of all strings that can be obtained as a permutation of the string $x$.

A circular string is a sequence $x = x_1x_2 \cdots x_n$ with the assumption (convention) that $x_1$ follows $x_n$. In other words, $x$ can be represented by any circular permutation of $x_1x_2 \cdots x_n$, for instance, $x_{i+1} \cdots x_nx_1 \cdots x_i$, for any $1 \leq i \leq n-1$. Thus, a circular string $\bullet x$ is an equivalence class of all linear strings associated to the linear string $x$. The set of all circular strings over $V$ is denoted by $V^\bullet$.

The *length* of a string $x \in V^*$ is the number of all occurrences in $x$ of symbols from $V$, and it is denoted by $|x|$, while the number of occurrences in $x$ of a specified symbol $a \in V$ is denoted by $|x|_a$. The empty word has a length 0. For a language $L \subseteq V^*$, the set $length(L) = \{|x| \mid x \in L\}$ is called the length set of $L$.

Given two words $x$ and $y$, the concatenation of $x$ and $y$ (in symbols, $xy$) is defined as the word $z$ consisting of all symbols of $x$ followed by all symbols of $y$; thus, $|z| = |x| + |y|$. In particular, the concatenation of a word $x$ with itself $k$ times will be denoted as $x^k$, and $x^0 = \lambda$.

The set of symbols from $V$ occurring in a string $x$ is denoted by $alph(x)$; given a language $L \subseteq V^*$, we define $alph(L) = \bigcup_{x \in L} alph(x)$.

Given an alphabet $V = \{a_1, a_2, \ldots, a_n\}$, with every string $x \in V^*$ we can associate the *Parikh vector* $\Psi_V(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$. Given a language $L \subseteq V^*$, we can also define the *Parikh image* of $L$ as $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$.

If $FL$ is a family of languages, then $PsFL$ denotes the family of Parikh images of languages in $FL$ and $NFL$ the family of length sets of languages in $FL$.

A set $M$ of vectors in $\mathbb{N}^n$, for some $n \geq 1$, is said to be *linear* if there are some vectors $v_0, \ldots, v_m \in \mathbb{N}^n, m \geq 0$, such that $M = \{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_1, \ldots, \alpha_m \in \mathbb{N}\}$. A finite union of linear sets is a semilinear set. A language $L \subseteq V^*$ is *semilinear* if its Parikh image $\Psi_V(L)$ is a semilinear set.

The usual set operations of union, intersection, difference, and complementation can be naturally extended to languages.

There also are several operations which are specific to languages:

- the *concatenation* of two languages $L_1, L_2 \subseteq V^*$ is the set $L_1 L_2 = \{x_1 x_2 \mid x_1 \in L_1, x_2 \in L_2\}$;
- the *Kleene closure* is $L^* = \bigcup_{i \geq 0} L^i$, where $L^i = L^{i-1} L$ for all $i \geq 1$ and, by convention, $L^0 = \{\lambda\}$. The *+-Kleene closure* is the set $L^+ = \bigcup_{i \geq 1} L^i$;
- the *right quotient* of a language $L_1 \subseteq V^*$ with respect to a language $L_2 \subseteq V^*$ is the set $L_1/L_2 = \{x \in V^* \mid xy \in L_1 \text{ for some } y \in L_2\}$;
- the *left quotient* of a language $L_1 \subseteq V^*$ with respect to a language $L_2 \subseteq V^*$ is the set $L_2 \backslash L_1 = \{x \in V^* \mid yx \in L_1 \text{ for some } y \in L_2\}$;
- the *right derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is the set $\partial_x^r(L) = \{w \in V^* \mid wx \in L\}$;
- the *left derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is the set $\partial_x^l(L) = \{w \in V^* \mid xw \in L\}$.

Let $\mathcal{L}(U^*)$ denote the set of all languages over an alphabet $U$. A *finite substitution* over an alphabet $V$ is a mapping $\sigma : V^* \to \mathcal{L}(U^*)$ such that, for each symbol $a \in V$, $\sigma(a)$ is a finite non-empty language, with $\sigma(\lambda) = \{\lambda\}$ and $\sigma(x_1 x_2) = \sigma(x_1)\sigma(x_2)$ for all strings $x_1, x_2 \in V^*$. If none of the languages $\sigma(a), a \in V$, contains the empty string, then the substitution is

called $\lambda - free$. If each $\sigma(a)$ consists of a single string, then the substitution is said to be a *morphism.*

A morphism $\sigma : V^* \to U^*$ is called a *coding* if $\sigma(a) \in U$ for all $a \in V$ and it is called a *weak coding* if $\sigma(a) \in U \cup \{\lambda\}$ for each $a \in V$.

The mappings defined on strings are extended to languages in the natural way; for instance if $\sigma : V^* \to U^*$ is a morphism and $L \subseteq V^*$, then $\sigma(L) = \{\sigma(x) \mid x \in L\}$.

A family $FL$ of languages is *closed* under an $n$-ary operation $\gamma_n$ if, for all languages $L_1, \ldots, L_n$ in $FL$, the language $\gamma_n(L_1, \ldots, L_n)$ is also in $FL$.

Any language which can be obtained by using finitely many times the operations of union, concatenation, and Kleene closure is called *regular.* Regular languages are defined (among many other possibilities) by means of *regular expressions.* In short, such an expression over a given alphabet $V$ is constructed starting from $\lambda$ and the symbols of $V$ and using the operations of union, concatenation, and Kleene $+$, using parentheses when necessary for specifying the order of operations. Specifically, (i) $\lambda$ and each $a \in V$ are regular expressions, (ii) if $E_1$, $E_2$ are regular expressions over $V$, then also $(E_1) \cup (E_2)$, $(E_1)(E_2)$, $(E_1)^+$ are regular expressions over $V$, and (iii) nothing else is a regular expression over $V$. With each regular expression $E$ we associate a language $L(E)$, defined in the following way: (i) $L(\lambda) = \{\lambda\}$ and $L(a) = \{a\}$, for all $a \in V$, (ii) $L((E_1) \cup (E_2)) = L(E_1) \cup L(E_2)$, $L((E_1)(E_2)) = L(E_1)L(E_2)$, and $L((E_1)^+) = L(E_1)^+$, for all regular expressions $E_1, E_2$ over $V$. Non-necessary parentheses are omitted when writing a regular expression, and $(E)^+ \cup \{\lambda\}$ is written in the form $(E)^*$.

A *multiset* over a set $V$ is a map $M : V \to \mathbb{N}$, where $M(a)$ denotes the multiplicity of the symbol $a \in V$ in the multiset $M$. This fact can also be indicated in the forms $(a, M(a))$ or $a^{M(a)}$, for all $a \in V$. If the set $V$ is finite, e.g., $V = \{a_1, \ldots, a_n\}$, then the multiset $M$ can be explicitly described as $\{(a_1, M(a_1)), (a_2, M(a_2)), \ldots, (a_n, M(a_n))\}$. The *support* of a multiset $M$ is the set $supp(M) = \{a \in V \mid M(a) > 0\}$. A multiset is empty (respectively, finite) when its support is empty (respectively, finite).

Some basic operations may be defined for multisets. Let $M_1, M_2 : V \to \mathbb{N}$ be two multisets. We say that $M_1$ is included in $M_2$, and we denote it by $M_1 \subseteq M_2$, if $M_1(a) \leq M_2(a)$ for all $a \in V$. The inclusion is strict, $M_1 \subset M_2$, if $M_1 \subseteq M_2$ and $M_1 \neq M_2$. The union of $M_1$ and $M_2$ is the multiset $M_1 \cup M_2 : V \to \mathbb{N}$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$ for all $a \in V$. The difference is the multiset $M_1 - M_2 : V \to \mathbb{N}$ defined by $(M_1 - M_2)(a) = M_1(a) - M_2(a)$ for all $a \in V$; obviously, $M_1 - M_2$ is defined only when $M_2$ is included in $M_1$.

If $n \in \mathbb{N}$ and $M : V \to \mathbb{N}$ is a multiset, then the scalar product of $M$ by $n$ is the multiset $n \otimes M : V \to \mathbb{N}$ defined by $(n \otimes M)(a) = n \cdot M(a)$ for all $a \in V$.

A compact notation can be used for finite multisets: if $M = \{(a_1, M(a_1)),$

$(a_2, M(a_2)), \ldots, (a_n, M(a_n))\}$ is a multiset of finite support, then the string $w = a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)}$ (and all its possible permutations) precisely identifies the symbols in $M$ and their multiplicities. Hence, given a string $w \in V^*$, we can assume that it identifies a finite multiset over $V$ defined by $M(w) = \{(a, |w|_a) \mid a \in V\}$. Moreover, given two strings $w_1, w_2$ representing two multisets, the concatenation $w_1 w_2$ denotes the multiset obtained by the union of multisets represented by $w_1$ and $w_2$.

### 2.1.2 Chomsky Grammars

A *Chomsky grammar*, [23], is a quadruple $G = (N, T, S, P)$, where $N, T$ are disjoint alphabets of non-terminal and terminal symbols, $S \in N$ is the axiom and $P$ is a finite set of rewriting rules (or *productions*) of the form $u \to v$, with $u \in (N \cup T)^+, v \in (N \cup T)^*$, with the condition that $u$ contains at least one non-terminal symbol.

Given a string $w = w_1 u w_2$ for $w_1, w_2 \in (N \cup T)^*$ and a production $u \to v$ in $P$, we can rewrite $u$ by means of $v$ and we obtain the string $z = w_1 v w_2$. This operation is called a *direct derivation* in $G$ and it is denoted by $w \Longrightarrow z$. The reflexive and transitive closure of the relation $\Longrightarrow$ is denoted by $\Longrightarrow^*$, meaning that we have $w \Longrightarrow^* z$ if either $w = z$, or $w \Longrightarrow z_1 \Longrightarrow z_2 \Longrightarrow \ldots \Longrightarrow z_k = z$ for some $z_1, \ldots, z_k \in V^*, k \geq 1$. Each string $w \in (N \cup T)^*$ such that $S \Longrightarrow^* w$ is called a *sentential form.*

The *language* $L(G)$ generated by $G$ is the set of sentential forms over the terminal alphabet, that is, $L(G) = \{x \in T^* \mid S \Longrightarrow^* x\}$.

If in the derivation $w \Longrightarrow z$ above we have that $w_1 \in T^*$, then the derivation is *leftmost.* If $w_2 \in T^*$, then the derivation is *rightmost.*

According to the form of the rules in the set $P$, the Chomsky grammars can be classified as follows:

- *Type-0 grammar*: the productions are of the form $u \to v$, where $u \in (N \cup T)^+, v \in (N \cup T)^*$, and $u$ contains at least one non-terminal symbol.

- *Type-1 (context-sensitive) grammar*: the productions are of the form[1] $u \to v$, where $u = u_1 A u_2, v = u_1 z u_2$, with $u_1, u_2 \in (N \cup T)^*, A \in N, z \in (N \cup T)^+$. The form of the productions of a context-sensitive grammar can equivalently be given in terms of the length of the string: for all $u \to v$ in $P$, the condition $|u| \leq |v|$ must hold.

- *Type-2 (context-free) grammar*: the productions are of the form $A \to v$, where $A \in N, v \in (N \cup T)^*$.

[1]The production $S \to \lambda$ is also allowed, providing that the axiom $S$ does not appear in the right-hand member of any rule in $P$.

- *Type-3 (regular) grammar*: the productions are either of the forms $A \to wB$, $A \to w$ (*right regular*) or of the form $A \to Bw, A \to w$ (*left regular*), where $A, B \in N, w \in T^*$.

The families of languages generated by context-sensitive, context-free, and regular grammars are denoted by $CS, CF, REG$, respectively, and are naturally called context-sensitive, context-free, and regular languages. The languages generated by type-0 grammars are called *recursively enumerable languages* and their family is denoted by $RE$.

We can have next informal definitions.

**Definition 2.1** *We call a language $L$* recursively enumerable *iff there is an effective procedure of listing only the words in $L$ with or without repetitions.*

By an effective procedure (equivalently, algorithm or mechanically executable process) we mean a finite set of instructions which describe in an unambiguous and detailed way how the task is performed. There are no restrictions on space or time needed to carry out the procedure, although the amount of information available at each step of the procedure is finite.

**Definition 2.2** *We call a language $L$* recursive *iff there is a terminating algorithm which decides, for any word $w$, whether or not $w$ belongs to $L$.*

We also denote by $FIN$ the family of finite languages. Among these families, the following strict inclusions (describing the *Chomsky hierarchy*) hold:

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

**Theorem 2.1 (Chomsky normal form)** *For each $\lambda$-free context-free grammar $G$ an equivalent context-free grammar $G' = (N, T, S, P)$ can be effectively constructed, where the rules in $P$ have the forms $A \to BC$ or $A \to a$, with $A, B, C \in N$ and $a \in T$.*

**Theorem 2.2 (Greibach normal form)** *For each $\lambda$-free context-free grammar $G$ an equivalent context-free grammar $G' = (N, T, S, P)$ can be effectively constructed, where the rules in $P$ have the forms $A \to aX$ with $A \in N, a \in T$, and $X \in N^*$.*

**Theorem 2.3 (Kuroda normal form)** *For each type-0 grammar $G$ there exists an equivalent type-0 grammar $G' = (N, T, S, P)$ where the rules have the form $A \to BC, A \to a, A \to \lambda, AB \to CD$, with $A, B, C, D \in N$ and $a \in T$.*

### 2.1.3 Regulated Rewriting

The idea of regulated rewriting consists in restricting the application of the rules in a context-free grammar, in order to avoid some derivations and hence obtaining a subset of the context-free language generated in the usual way. The computational power of some context-free grammars with regulated rewriting turns out to be greater than the power of context-free grammars.

We recall here *matrix* grammars.

A *matrix grammar with appearance checking (ac)* is a construction $G = (N, T, S, M, F)$, where $N, T$ are disjoint alphabets of non-terminal and terminal symbols, $S \in N$ is the axiom, $M$ is a finite set of matrices, which are sequences of context-free rules of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, (with $A_i \in N, x_i \in (N \cup T)^*$, in all cases), and $F$ is a set of occurrences of rules in $M$.

For $w, z \in (N \cup T)^*$ we write $w \Longrightarrow z$ if there are a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ in $M$ and strings $w_i \in (N \cup T)^*, 1 \leq i \leq n+1$, such that $w = w_1$, $z = w_{n+1}$, and, for all $1 \leq i \leq n$, either

(*i*) $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$,

or

(*ii*) $w_i = w_{i+1}$, $A_i$ does not appear in $w_i$, and $A_i \to x_i$ appears in $F$.

The rules of a matrix are applied in order, possibly skipping the rules in $F$ if they cannot be applied (one says that these rules are applied in the appearance checking mode). The family of languages generated by matrix grammars with appearance checking is denoted by $MAT_{ac}$. $G$ is called a *matrix grammar without appearance checking* if and only if $F = \emptyset$. In this case, the generated family of languages is denoted by $MAT$.

The following results hold:

**Theorem 2.4**

- $CF \subset MAT \subset MAT_{ac} = RE$.
- $CS - MAT \neq \emptyset$.
- $MAT$ is closed under the operations of union, concatenation, intersection with regular languages.
- Each language $L \in MAT$ over the one-letter alphabet, $L \subseteq \{a\}^*$, is regular.

A matrix grammar $G = (N, T, S, M, F)$ is said to be in the *binary normal form* if $N = N_1 \cup N_2 \cup \{S, \#\}$ with sets $N_1, N_2, \{S, \#\}$ mutually disjoint, and the matrices in $M$ are of one of the following forms:

1. $(S \to XA)$ with $X \in N_1, A \in N_2$;

2. $(X \to Y, A \to x)$ with $X, Y \in N_1, A \in N_2$, $x \in (N_2 \cup T)^*, |x| \leq 2$;
3. $(X \to Y, A \to \#)$ with $X, Y \in N_1, A \in N_2$;
4. $(X \to \lambda, A \to x)$ with $X \in N_1, A \in N_2, x \in T^*, |x| \leq 2$.

Moreover, there exists only one matrix of type 1, $F$ exactly consists of all rules $A \to \#$ appearing in matrices of type 3, and $\#$ is a trap-symbol (once introduced, it can never be removed). Finally, each matrix of type 4 is used only once, at the last step of a derivation.

The corresponding normal form for appearance checking does not use matrices of type 3 and the symbol $\#$.

**Theorem 2.5** *For each matrix grammar (with or without appearance checking) there exists an equivalent one in the binary normal form.*

There exists even a more restricted normal form for matrix grammars with appearance checking. We say that a matrix grammar $G = (N, T, S, M, F)$ is in the *Z-binary normal form* if $N = N_1 \cup N_2 \cup \{S, Z, \#\}$ is the union of mutually disjoint sets, and the matrices in $M$ are of one of the following forms:

1. $(S \to XA)$ with $X \in N_1, A \in N_2$;
2. $(X \to Y, A \to x)$ with $X, Y \in N_1, A \in N_2$, $x \in (N_2 \cup T)^*, |x| \leq 2$;
3. $(X \to Y, A \to \#)$ with $X \in N_1, Y \in N_1 \cup \{Z\}, A \in N_2$;
4. $(Z \to \lambda)$.

Moreover, there is only one matrix of type 1, $F$ consists exactly of all rules $A \to \#$ appearing in matrices of type 3 ($\#$ is a trap-symbol, if it is introduced, then it cannot be removed) and, if a sentential form generated by $G$ contains the symbol $Z$, then it is of the form $Zw$, for some $w \in (T \cup \{\#\})^*$. The matrix of type 4 is used only once, in the last step of a derivation.

**Theorem 2.6** *For each $L \in RE$ there is a matrix grammar with appearance checking in the Z-binary normal form such that $L = L(G)$.*

In several proofs in the P systems area (and also in this thesis) matrix grammars in the binary normal form or in the Z-binary normal form are used. To simplify the use of these grammars we introduce a *standard notation* usually used in the area. A matrix grammar with appearance checking in the binary normal form is always given as $G = (N, T, S, M, F)$ with $N = N_1 \cup N_2 \cup \{S, \#\}$, and with $n+1$ matrices in $M$, injectively labeled with $m_0, m_1, \ldots, m_n$; the matrix $m_0 : (S \to X_{init} A_{init})$ is the initial one, with $X_{init}$ a given symbol from $N_1$ and $A_{init}$ a given symbol from $N_2$; the next

$k$ matrices are without appearance checking rules, $m_i : (X \to \alpha, A \to x)$, $1 \leq i \leq k$, where $X \in N_1, \alpha \in N_1 \cup \{\lambda\}$, and $A \in N_2, x \in (N_2 \cup T)^*, |x| \leq 2$ (if $\alpha = \lambda$ then $x \in T^*$); the last $n-k$ matrices have rules to be applied in the appearance checking mode, $m_i : (X \to Y, A \to \#), k+1 \leq i \leq n$, with $X, Y \in N_1$, $A \in N_2$.

The non-terminal matrices $m_i, 1 \leq i \leq k$, are called *matrices of type 2*, the matrices $m_i, k+1 \leq i \leq n$, are called *matrices of type 3*, and the terminal matrices $m_i : (X \to \lambda, A \to x), 1 \leq i \leq k$, are called *matrices of type 4*.

In the case of grammars in the Z-binary normal form, we have $N = N_1 \cup N_2 \cup \{S, Z, \#\}$, the matrices $m_i, k+1 \leq i \leq n$, can also be of the form $m_i : (X \to Z, A \to \#)$, and the only terminal matrix is $m_{n+1} : (Z \to \lambda)$.

### 2.1.4 Parallel Rewriting

The literature is rich of parallel rewriting devices, where the rewriting of the current sentential form is done in a parallel way (and not in the sequential way like for Chomsky grammars).

Lindenmayer systems (or *L systems*, in short) are the most known parallel rewriting systems. They have been introduced in 1968 to the aim of modeling the development of simple multicellular organisms. The peculiar characteristic of L systems is their *parallelism*: at each step of the rewriting process, all symbols of the current string have to be rewritten, in contrast to the *sequential* rewriting of phrase structure grammars, where only a specific part of the string is rewritten at each step.

A 0L (*zero-interaction Lindenmayer*) *system* is a construction $G = (V, w, h)$, where $V$ is an alphabet, $w \in V^+$ is the axiom, and $h$ is a finite set of rules of the form $a \to v$, with $a \in V, v \in V^*$, such that for each $a \in V$ there is at least one rule $a \to v$ in $h$ (we say that $P$ is *complete*). For $w_1, w_2 \in V^*$ we write $w_1 \Longrightarrow w_2$ if $w_1 = a_1 \ldots a_n$ and $w_2 = v_1 \ldots v_n$, for $a_i \to v_i \in h, 1 \leq i \leq n$. The *language* generated by $G$ is $L(G) = \{x \in V^* \mid w \Longrightarrow^* x\}$.

If for each $a \in V$ there is exactly one rule $a \to v$ in $P$, then $G$ is said to be *deterministic*. If for each rule $a \to v \in h$ it holds $v \neq \lambda$, then the system $G$ is said to be *propagating* (or non-erasing). If we distinguish a subset $T$ of $V$ and we define the generated language as $L(G) = \{x \in T^* \mid w \Longrightarrow^* x\}$, then $G$ is said to be *extended*.

The family of languages generated by 0L systems is denoted by $0L$, and we add the letters $D, P, E$ in front of $0L$ if the systems are also deterministic, propagating or extended, respectively.

A *tabled* 0L system (T0L) is a construction $G = (V, w, h_1, \ldots, h_n)$ such that each triple $(V, w, h_i), 1 \leq i \leq n$, is a 0L system; each $h_i$ is called a *table*. The language generated by $G$ is $L(G) = \{x \in V^* \mid w \Longrightarrow_{h_{j_1}} w_1 \Longrightarrow_{h_{j_2}} \ldots \Longrightarrow_{h_{j_m}} w_m = x, m \geq 0, 1 \leq j_i \leq n, 1 \leq i \leq m\}$, that is, at each

derivation step only the rules from the same table can be applied. A T0L system is deterministic when each table is deterministic, it is propagating if none of its tables contains an erasing rule, it is extended if there exists an alphabet of terminal symbols. The family of languages generated by T0L systems is denoted by $T0L$, and the letters $D, P, E$ in front of $T0L$ are added if the systems are also deterministic, propagating or extended, respectively.

It is known that:

**Theorem 2.7**

- $CF \subset E0L \subset ET0L \subset CS$;
- $ED0L \subset EDT0L \subset ET0L$;
- $ED0L \subset E0L$.

A very useful normal form for ET0L systems is the following:

**Theorem 2.8** *For each $L \in ET0L$ there exists an $ET0L$ system $G = (V, T, w, h_1, h_2)$ with only two tables, such that $L = L(G)$.*

Moreover, the following stronger normal form for tabled Lindenmayer systems can be obtained.

**Theorem 2.9** *For each $L \in ET0L$ there is an $ET0L$ system $G = (V, T, w, h_1, h_2)$ generating $L$, such that the terminals are only trivially rewritten: for each $a \in T$, if $(a \to \alpha) \in h_1 \cup h_2$, then $\alpha = a$ (these productions are called trivial).*

### 2.1.5 The Splicing Operation

We first introduce the splicing operation and the splicing systems in the formalization considered in [87].

Consider an alphabet $V$ and two special symbols, $\#, \$$, not in $V$. A *splicing rule* (over $V$) is a string of the form

$$r = u_1 \# u_2 \$ u_3 \# u_4,$$

where $u_1, u_2, u_3, u_4 \in V^*$. (For a maximal generality, we place no restriction on the strings $u_1, u_2, u_3, u_4$. The cases when $u_1 u_2 = \lambda$ or $u_3 u_4 = \lambda$ could be ruled out as unrealistic.)

For a splicing rule $r = u_1 \# u_2 \$ u_3 \# u_4$ and strings $x, y, z \in V^*$ we write

$$\begin{aligned} (x, y) \vdash_r z \quad \text{iff} \quad & x = x_1 u_1 u_2 x_2, \\ & y = y_1 u_3 u_4 y_2, \\ & z = x_1 u_1 u_4 y_2, \\ & \text{for some } x_1, x_2, y_1, y_2 \in V^*. \end{aligned}$$

We say that we *splice* $x, y$ at the *sites* $u_1u_2, u_3u_4$, respectively, and the result is $z$.

A *splicing* $(H)$ *scheme* is a pair

$$\sigma = (V, R),$$

where $V$ is an alphabet and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules.

For a given H scheme $\sigma = (V, R)$ and a language $L \subseteq V^*$, we define

$$\sigma_1(L) = \{z \in V^* \mid (x, y) \vdash_r z, \text{ for some } x, y \in L, r \in R\}.$$

Actually, in [44], a different formulation is used. Specifically, in [44], a splicing scheme is a pair $R = (V, \sim)$ where $V$ is an alphabet and $\sim$ a pairing relation, $\sim \subseteq V^3 \times V^3$. Then, having two strings $x$, $y$ and two triples of nonempty words $(\alpha, p, \beta)$, $(\alpha', q, \beta')$ in relation $\sim$ (we write $(\alpha, p, \beta) \sim (\alpha', q, \beta')$), if $x = x'\alpha p\beta x''$ and $y = y'\alpha' q\beta' y''$, and $p = q$, then we allow the recombination operation; the strings obtained by recombination are $z_1 = x'\alpha r\beta' y''$ and $z_2 = y'\alpha' r\beta x''$, where $r = p = q$.

When having a pair $(\alpha, p, \beta) \sim (\alpha', q, \beta')$ and two strings $x$ and $y$ as above, $x = x'\alpha p\beta x''$ and $y = y'\alpha' q\beta' y''$, we can consider only the string $z_1 = x'\alpha r\beta' y''$ as a result of the recombination, because the string $z_2 = y'\alpha' r\beta x''$, is the result of the one-output-recombination with respect to the symmetric pair, $(\alpha', q, \beta') \sim (\alpha, p, \beta)$.

### 2.1.6 Automata Theory

Automata are computing devices which work as accepting devices: they can *decide* whether a string given as input belongs to a specified language.

The basic families of languages of the Chomsky hierarchy are characterized by different kinds of recognizing automata. We present here several types of automata for languages of finite strings.

A (*non-deterministic*) *finite automaton* is defined by the construction $A = (Q, V, q_0, \delta, F)$, where $Q$ is a finite set of states, $V$ is a non-empty finite set of input symbols (such that $Q \cap V = \emptyset$), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta : Q \times V \to \mathcal{P}(Q)$ is the transition function. We denote by $\delta(q, a)$ the set of all states $p \in Q$ such that there is a transition from $q$ to $p$ on input $a$. If $card(\delta(q, a)) \leq 1$ for all $q \in Q, a \in V$, then the automaton is said to be *deterministic*. In one move, if the finite automaton is in the state $q \in Q$ and it is reading the input symbol $a \in V$, then it enters one of the next states in $\delta(q, a)$ and moves its head one symbol to the right.

To formally describe the behavior of the finite automaton on an input string and not only on an input symbol, we must consider an extension of the transition function, namely $\widehat{\delta} : Q \times V^* \to \mathcal{P}(Q)$, such that

1. $\widehat{\delta}(q, \lambda) = q$,

2. $\widehat{\delta}(q, xa) = \bigcup_{q' \in \widehat{\delta}(q,x)} \delta(q', a)$, where $x \in V^*, a \in V$, and $q, q' \in Q$.

The first condition disallows a change in state without any input, the case when the head is reading an empty cell on the tape. The second condition indicates that, starting in state $q$ and reading the string $x$ followed by the input symbol $a$, the finite automaton can be in any of the states which are reachable from the state $q'$ after reading the symbol $a$.

A string $w \in V^*$ is said to be *accepted* (or recognized) by the finite automaton if, starting with the finite control in the initial state $q_0$ and with the head reading the first symbol in $w$, the automaton reaches a final state after the last symbol in $w$ has been scanned. Hence, the language recognized by $A$ is the set of strings $L(A) = \{w \in V^* \mid \widehat{\delta}(q_0, w) \cap F \neq \emptyset\}$.

Non-deterministic and deterministic finite automata are known to be equivalent in the following sense: they both characterize the family of regular languages, *REG*.

A Turing machine is a computational device consisting of a finite state control, an input tape (which is infinite to the right), and a read/write head which can read symbols from the tape and also rewrite the scanned symbols.

Turing machines characterize the family of languages generated by type-0 grammars and, in addition to being language acceptors, they may be viewed as computers of functions from integers to integers. Actually, they are able to compute any *partial recursive function.* The importance of Turing machines is a direct consequence of the *Church-Turing thesis*: the intuitive notion of "computable functions" is identified with the class of partial recursive functions. Thus, the model of Turing machine is computationally complete. Many other formalisms of computation have been defined, but none of them have been shown to be strictly more powerful than Turing machines.

Another relevant characteristic of Turing machines is that there exists a Turing machine $U$ that simulates the computation of an arbitrary Turing machine $M$ on any of its input. Such machine is called a *universal Turing machine.* Thus, we can say that Turing machines are both complete and universal computing devices. A rewriting system $M = (S, \Sigma \cup \{\#\}, P)$ is called a *Turing machine*, [98], iff:

(*i*) $S$ and $\Sigma \cup \{\#\}$ (with $\# \notin \Sigma$ and $\Sigma \neq 0$) are two disjoint alphabets referred to as the *state* and *tape* alphabets.

(*ii*) Elements $s_0$ and $s_f$ of $S$, and $\sqcup$ of $\Sigma$ are the *initial* and *final* state, and the *blank symbol*, respectively. Also a subset $T$ of $\Sigma$ is specified and referred to as the *terminal* alphabet. It is assumed that $T$ is not empty.

(*iii*) The productions (rewriting rules) of $P$ are of the forms

| | | |
|---|---|---|
| (1) | $s_i a \longrightarrow s_j b$ | (overprint) |
| (2) | $s_i ac \longrightarrow as_j c$ | (move right) |
| (3) | $s_i a\# \longrightarrow as_j \sqcup \#$ | (move right and extend workspace) |
| (4) | $cs_i a \longrightarrow s_j ca$ | (move left) |
| (5) | $\#s_i a \longrightarrow \#s_j \sqcup a$ | (move left and extend workspace) |
| (6) | $s_f a \longrightarrow s_f$ | |
| (7) | $as_f \longrightarrow s_f$ | |

where $s_i$ and $s_j$ are states in $S$, $s_i \neq s_f$, $s_j \neq s_f$, and $a, b, c$ are in $\Sigma$. For each pair $(s_i, a)$, where $s_i$ and $a$ are in the appropriate ranges, $P$ either contains no productions (2) and (3) (resp. (4) and (5)) or else contains both (3) and (2) for every $c$ (resp. contains both (5) and (4) for every $c$). There is no pair $(s_i, a)$ such that the word $s_i a$ is a subword of the left side in two productions of the form (1), (3), (5).

A configuration of the machine $M$ is of the form $\#w_1 s_i w_2 \#$, where $w_1 w_2$ represents the contents of the tape, #s are the boundary markers, and the position of the state symbol $s_i$ indicates the position of the read/write head on the tape: if $s_i$ is positioned at the left of a letter $a$, this indicates that the read/write head is placed over the cell containing $a$. The TM changes from one configuration to another according to its rules. The Turing machine $M$ *halts* with a word $w$ iff there exists a derivation that, when started with the read/write head positioned at the beginning of $w$ eventually reaches the final state, i.e., if $\#s_0 w\#$ derives $\#s_f\#$ by successive applications of the rewriting rules (1) – (7). The language accepted by $M$ is defined as follows:

$$L(M) = \{w \in T^* \mid \#s_0 w\# \Longrightarrow_P^* \#s_f\#\}.$$

The machine $M$ is deterministic if at each step of the rewriting process, at most one production can be applied.

There is an equivalent mode of defining the language accepted by a Turing machine: one considers the set of strings $w \in T^*$ such that the machine, starting from the initial state and with $w$ as input, reaches a configuration where no further moves are possible (we say that the machine has reached an *halting state*). Note that it is also possible that a Turing machine never reaches a halting state on some input, as its head is allowed to move both directions on the tape. Hence, given an input string $w$, it may happen that the Turing machine reaches a final state, or it halts in a non-final state, or it never halts (it enters an infinite loop). In the last two cases, the input string $w$ is not accepted by the Turing machine.

Observe that, given a non-deterministic Turing machine $M$ and an input string $w$, usually there are many different computations of $M$ on input $w$. Each computation corresponds to a *path* in the tree associated to $M$, which can be naturally defined as follows: the root corresponds to the initial

configuration of $M$; each node corresponds to a configuration of $M$, and an arc between two nodes corresponds to a direct transition between the configurations of $M$ associated to such nodes; the leaves are (final or non-final) halting configurations of $M$. Note that there also might be infinite paths in the tree, corresponding to the possible non-halting computations of $M$.

It is well-known that the power of deterministic and non-deterministic Turing machine is equivalent, they both characterize the family of recursively enumerable languages ($RE$).

Having presented Turing machines we can recall the definition of *recursive languages.* A language $L$ over the alphabet $V$ is recursive if there exists a Turing machine that, when receives as input a string $w$ over $V$, halts and accepts if the string $w$ is the language $L$, otherwise halts and rejects.

**Theorem 2.10** *Let $L$ be a given language. Then the following conditions are all equivalent:*

- *$L$ is recursively enumerable;*
- *$L$ is accepted by a deterministic Turing machine;*
- *$L$ is accepted by a non-deterministic Turing machine;*
- *there is a grammar $G$ such that $L$=$L(G)$.*

When working on an input string a Turing machine is allowed to use as much tape as it needs. Note that finite state automata use (in the read manner) only the cells where the input string is written. A Turing machine allowed to use only a working space linearly bounded with respect to the length of the input is called *linearly bounded automaton.* These machines characterize the family of context-sensitive languages ($CS$).

We also recall the definition of other Turing universal computing devices largely used in the P systems area: *register machines.*

An *n-register machine* is a construct $M = (n, B, l_0, l_h, I)$, where $n$ is the number of registers, $B$ is a set of labels, and $I$ is a set of labeled instructions of the form $l_i : (op(r), l_j, l_k)$, where $op(r)$ is an operation on register $r$ of $M$, and $l_i, l_j, l_k$ are labels from the set $B$; $l_0$ is the label of the initial instruction, and $l_h$ is the label of the halting instruction. The machine is capable of the following instructions:

1. $l_i : (\mathsf{add}(r), l_j, l_k)$: Add one to the content of register $r$ and proceed, in a non-deterministic way, to instruction with label $l_j$ or to instruction with label $l_k$; in the deterministic variant we demand $l_j = l_k$ and then the instruction is written in the form $l_i : (\mathsf{add}(r), l_j)$.

2. $l_i : (\mathsf{sub}(r), l_j, l_k)$: If register $r$ is not empty, then subtract one from its contents and go to instruction with label $l_j$, otherwise proceed to instruction with label $l_k$.

3. $l_h$ : halt: This instruction stops the machine and can only be assigned to the final label $l_h$.

A deterministic $n$-register machine can analyze an input $m \in \mathbb{N}$, introduced in register 1, which is accepted if and only if the machine finally stops by the halt instruction with all its registers being empty. If the machine does not halt, then the analysis was not successful. We denote by $N(M)$ the set of numbers accepted by the register machine $M$. If $Q$ is a Turing computable set, then there exists a deterministic register machine $M$ with at most three registers, such that $N(M) = Q$, [67].

## 2.2 Computational Complexity

We give only some basic elements of computational complexity.

After defining a class of computing models, two fundamental questions should be answered: (1) *how powerful* these models are, in comparison with Turing machines and other classic models of computing, and (2) *how efficient* these models are. It is not enough that a problem can be solved algorithmically, it is necessary that the solution comes in a reasonable time and using reasonable computing resources. Defining what "reasonable time and resources" means and classifying problems from these points of view are the main tasks of computational complexity.

A complexity class is defined with respect to functions $f : \mathbb{N} \longrightarrow \mathbb{N}$ in the following way: the complexity class **md-res**$_M(f)$ consists of all problems which can be solved in mode $md$ by devices $D$ from class $M$ such that for any input $x$, device $D$ needs at most $f(|x|)$ units from the resource $res$. Let $M$ be a deterministic Turing machine that halts on all inputs. The running time or time complexity of $M$ is the function $t : \mathbb{N} \to \mathbb{N}$, where $t(n)$ is the maximum number of steps that $M$ uses on any input of length $n$. If $t(n)$ is the running time of $M$, we say that $M$ runs in time $t(n)$ and that $M$ is a $t(n)$ time Turing machine. The time complexity class TIME($t(n)$) is defined as TIME($t(n)$)=$\{L \mid L$ *is a language decided by an* $O(t(n))$ *time Turing machine*$\}$.

In time complexity theory polynomial differences in running time are considered to be small, whereas exponential differences are considered to be large. Polynomial time algorithms are fast enough for many purposes, but exponential time algorithms rarely are useful. All reasonable deterministic computational models are polynomially equivalent, that is, any of them can simulate another with only a polynomial increase in running time.

The most important complexity classes are DTIME($t$), DSPACE($f$), which contain the problems which can be solved in a deterministic mode in a time, respectively, a space bounded by the functions $t$ and $f$, and NTIME($t$), NSPACE($f$), which contain the problems which can be solved

in a non-deterministic mode in a time, respectively, a space bounded by the functions $t$ and $f$.

Now, the important step is to chose the function $t$. The most investigated complexity classes are the constant, logarithmic, linear, polynomial, and exponential functions. The union of all classes DTIME($n^k$), $k \geq 1$, is usually denoted by **P**, while the union of all classes NTIME($n^k$), $k \geq 1$, is usually denoted by **NP**. These are the classes of problems which can be solved in a deterministic polynomial time, respectively of problems which can be solved in a non-deterministic polynomial time.

The problems can be *reduced* to other problems. For most natural complexity classes, **P** and **NP** included, there are problems such that *all* problems from that class can be reduced to them. Such problems are called *complete* for the respective classes. Of particular interest are the problems which are **NP**-complete. The **NP**-complete problems are "the most difficult from **NP**". This means, in particular, that they cannot yet be solved in real-time by electronic computers running known algorithms.

## 2.3 Membrane Computing Preliminaries

Membrane systems (referred also as P systems) are a class of distributed parallel computing devices of a biochemical type, which can be seen as a general computing architecture where various types of objects can be processed in parallel by various operations. By *membranes*, a system is divided into compartments where chemical reactions can take place. These reactions transform multisets of objects present in the compartments into new objects, possibly transferring objects to neighboring compartments, including the environment.

Generally, there are two standard ways of investigating the influence of various features of P systems: (1) to consider their computational power/ competence e.g., are P systems using given features computationally universal (hence equivalent to Turing machines)?, and (2) to consider their computational complexity (are P systems able to make use of their intrinsic parallelism and solve hard problems, e.g., **NP**-complete problems, in feasible time?).

In this section we give a general (and short) overview of membrane systems recalling the definitions of two important and well-studied classes of this model of computation, the one with symbol-objects and multiset rewriting rules, the one with computational complexity classes of P systems. We also recall the main results of universality and efficiency of the classes.

### 2.3.1 Membrane Systems with Symbol-Objects

We recall now the classical definition of a membrane system with symbol-objects that can be considered a basic model where only the basic elements

of a P system are present: membranes arranged in an hierarchical structure (as in the cell) and operations (called *evolution rules*) to process multisets of symbol-objects. In the structure of a membrane system with symbol-objects an unique external membrane is present called the *skin membrane*. An *elementary membrane* is a membrane without any membrane inside. The *region* enclosed by a membrane is the space between a membrane and the immediately inner membranes (if any). To each membrane is associated a *label*. Because there is a one-to-one correspondence between membranes and (enclosed) regions we can also say that membrane (with label) $i$ encloses region (with label) $i$. The entire system is *surrounded* by the *environment*. An illustration of these notions appears in Figure 2.1.

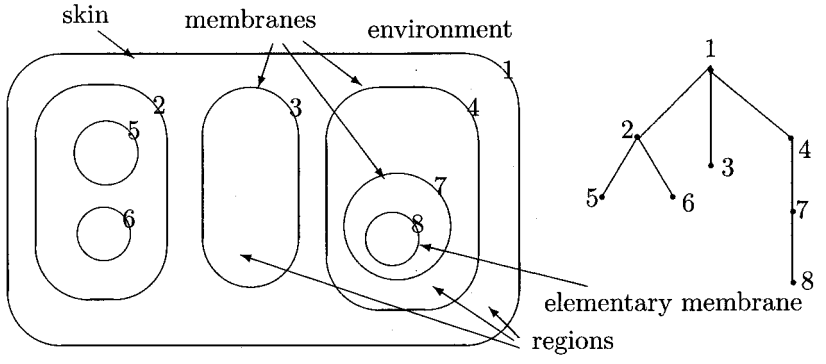



Figure 2.1: A membrane structure and its corresponding tree representation.

**Definition 2.3** *A P system with symbol-objects, of degree $m \geq 1$, is defined as*

$$\Pi = (O, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, i_0),$$

*where:*

- *$O$ is an alphabet and its elements are called* objects*;*
- *$\mu$ is a membrane structure consisting of $m$ membranes arranged in an hierarchical structure; the membranes (and hence the regions that they delimit) are injectively labeled with $1, 2, \cdots, m$;*
- *$w_i$, $1 \leq i \leq m$, are strings that represent multisets over $O$ associated with regions $1, 2, \cdots, m$ of $\mu$;*
- *$R_i$, $1 \leq i \leq m$, are finite sets of* evolution rules *over $O$; $R_i$ is associated with the region $i$ of $\mu$; an evolution rule is of the form $u \rightarrow v$, where $u$ is a string over $O$ and $v$ is a string over $\{a_{here}, a_{out} \mid a \in O\} \cup \{a_{in_j} \mid a \in O, 1 \leq j \leq m\}$.*

- $i_0 \in \{0, 1, 2, \cdots, m\}$; *if* $i_0 \in \{1, \cdots, m\}$, *then it is the label of an* elementary *membrane that encloses the* output region*; if* $i_0 = 0$, *then the output region is the environment.*

For any evolution rule $u \to v$ the length of $u$ is called the *radius* of the rule and the symbols *here, out,* $in_j, 1 \leq j \leq m$, are called *target* indications. To simplify the notation the target indication *here* is omitted. According to the size of the radius of the evolution rules we distinguish between *cooperative* systems (if the radius is greater than one) and *non-cooperative* systems (otherwise).

A special class of *cooperative* systems is that of *catalytic systems*, where a subset $C \subseteq O$ of special symbol-objects (called *catalysts*) is fixed.

In case of *catalytic systems*, the system is of the form

$$\Pi = (O, C, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, i_0).$$

In such systems the evolution rules can be of two different kinds: $a \to v$ and $ca \to cv$ (catalytic rules) where $c \in C$, $a \in O - C$, and $v$ is a string over $\{a_{here}, a_{out} \mid a \in O - C\} \cup \{a_{in_j} \mid a \in O - C, 1 \leq j \leq m\}$.

The *initial configuration* of the system $\Pi$ is constituted by the membrane structure $\mu$ and the multisets represented by the strings $w_i, 1 \leq i \leq m$. In general, we call *configuration* of the system the membrane structure and the tuple of multisets of objects present in the regions of the system.

A transition between configurations is executed using the evolution rules in a *non-deterministic maximally parallel manner* (we suppose that a global clock exists, marking the time for the whole system). This means that the objects are assigned to the rules in such a way that, after this assignation, no other rules can be applied to the objects that have been not assigned and this procedure is applied in parallel in each region of the system, at each step. If an object can be used by several evolution rules, then the choice is made in a non-deterministic way.

The application of an evolution rule $u \to v$ 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 $v$, in the regions specified by the target indications associated to each object in $v$. In particular, if $v$ contains an object $a$ with target indication *here*, then the object $a$ will be placed in the region $i$ where the evolution rule has been applied. If $v$ contains an object $a$ with target indication *out*, then the object $a$ will be moved to the region immediately outside the region $i$ (this can be the environment if the region where the rule has been applied is the *skin* membrane). If $v$ contains an object $a$ with target indication $in_j$ then the object $a$ is moved from the region $i$ and placed into the region $j$ (this can be done only if such region $j$ is immediately inside region $i$; otherwise the evolution rule $u \to v$ cannot be applied).

It is also possible to use target indications of the forms *here, out, in*, with *in* being a weaker version of $in_j$; in such a case, an object $a$ having the

target indication *in* goes to any lower level membrane, non-deterministically chosen (if no inner membrane exists, then the rule cannot be applied).

In other words, by using the evolution rule $u \rightarrow v$ in a region $i$ that contains a multiset identified by $w$, we subtract the multiset identified by $u$ from the multiset identified by $w$, and then we add the objects specified by $v$ to the multiset of the region $i$ and to the multisets of adjacent regions according to the target indications specified. In this way, at each step, all the multisets associated to the regions of the system evolve and the system passes from some configuration to a new one; this *passage* is called *transition.* A sequence of transitions between configurations of a system $\Pi$ is called a *computation*; a computation is *successful* (or *halting*) if and only if it *halts* and this means that there is no rule applicable to the objects present in the last configuration.

The *output* of a successful computation is defined as the number of objects present in the output region in the halting configuration of $\Pi$; the set of numbers computed (or generated) in this way by the system $\Pi$, considering any successful computation, is denoted by $N(\Pi)$.

It is possible to consider as result of a successful computation the vector of numbers representing the multiplicities of objects from the output region in the halting configuration. In this case $Ps(\Pi)$ denotes the set of vectors of numbers generated by $\Pi$, considering all the successful computations.

We denote by $NOP_m(\alpha, tar)$ $[(PsOP_m(\alpha, tar)]$ the family of sets of the form $N(\Pi)$ $[Ps(\Pi)$, resp.] generated by P systems of degree at most $m \geq 1$ (if the degree is not bounded, then the subscript $m$ becomes $*$), using evolution rules of the type $\alpha$.

We can have $\alpha = coo$ and this indicates that the systems considered use cooperative evolution rules, $\alpha = ncoo$ that indicates that the systems use only non-cooperative rules, and $\alpha = cat_k$ that means that the systems use catalytic and non-cooperative rules; the number of catalysts used is at most $k$. Moreover, the symbol $tar$ indicates that the communication between the membranes (and hence the regions) is made using the target indication $in_j$ in the way specified before. If the degree of the system is 1 (only one membrane is present) then the only possible target indications that can be used are *here* and *out* and in such case the notation is $NOP_1(\alpha)$.

We recall now some basic results obtained for P systems with symbol-objects. There results concern the sets of type $N(\Pi)$ but similar results are also true for the sets of type $Ps(\Pi)$. The computational power of symbol-objects P system using only *non-cooperative* evolution rules, is rather low. Such systems can only generate the family of length sets of context-free languages (hence the family of semilinear sets of numbers).

**Theorem 2.11** $NOP_*(ncoo, tar) = NOP_1(ncoo) = NCF.$

On the other hand, as expected, a symbol-object P system with *cooperative* rules is more powerful. In particular, even with only one membrane,

symbol-object P systems can generate all recursively enumerable sets of numbers.

**Theorem 2.12** *$NOP_*(coo, tar) = NOP_m(coo, tar) = NRE$ for all $m \geq 1$.*

This last theorem has been improved: in fact, it is possible to get universality even by using only catalytic evolution rules and two catalysts.

**Theorem 2.13** *$NOP_2(cat_2, tar) = NRE$.*

A special class of objects are the *bi-stable catalysts* that are catalysts having two states $c$ and $\overline{c}$ and the rules involving them may switch between these two states. Let denote by $C$ the set of bi-stable catalysts. The allowed rules are of the forms: $ca \to cv, ca \to \overline{c}v, \overline{c}a \to \overline{c}v$, and $\overline{c}a \to cv$ with $c \in C$, $a \in O-C$, $v$ is a string over $\{a_{here}, a_{out} \mid a \in O-C\} \cup \{a_{in_j} \mid a \in O-C, 1 \leq j \leq m\}$.

Notice that if the two states coincide, then $c$ is a usual catalyst.

We indicate the use of such bi-stable catalysts by writing $2cat$ instead of $cat$. So, by the definition, bi-stable catalysts are at least as powerful as catalysts.

Actually, universality can be obtained by using only one bi-stable catalyst and one membrane.

**Theorem 2.14** *$NOP_1(2cat_1, tar) = NRE$.*

### 2.3.2 Computational Complexity in P Systems

A rigorous framework for dealing with complexity matters in our area is that of *recognizing P systems*, which we introduce here following [88].

**Definition 2.4** *A recognizing P system is a P system such that:*

1. *The alphabet $O$ of $\Pi$ contains two distinguished elements,* `YES`, `NO`.
2. *All computations of the system halt.*
3. *If $C$ is a computation of $\Pi$, then either the object* `YES` *or the object* `NO` *(but not both) is sent out to the environment, and only in the last step of the computation.*

We say that $C$ is an accepting (respectively, rejecting) computation if the object `YES` (respectively, `NO`) appears in the environment in the halting configuration of $C$.

In order to assure that a family of recognizing P systems solves a decision problem, two main properties must to be proved: for each instance of the problem,

(a) if *there exists an* accepting computation of the membrane system processing it, answering YES, then the problem also answer YES for that instance (*soundness*);

(b) if the problem answers YES, then *any* computation of the system processing that instance, answer YES (*completeness*).

If we demand that the family of membrane systems is sound and complete, then it satisfies a condition of *confluence*: every computation of a system from the family has the same output.

Next, we formalize these ideas in the following definition.

**Definition 2.5** *Let $X = (I_X, \theta_X)$ be a decision problem. Let $\mathbf{\Pi} = (\Pi(w))_{w \in I_X}$ be a family of recognizing membrane systems* without input.

- *We say that the family $\mathbf{\Pi}$ is sound with regard to $X$ if the following is true: for each instance of the problem $w \in I_X$, if there exists an accepting computation of $\Pi(w)$, then $\theta_X(w) = 1$.*
- *We say that the family $\mathbf{\Pi}$ is complete with regard to $X$ if the following is true: for each instance of the problem $w \in I_X$, if $\theta_X(w) = 1$, then every computation of $\Pi(w)$ is an accepting computation.*

The soundness property means that if we obtain an *acceptance response* of the system (associated with an instance) through some computation, then the answer of the problem (for that instance) is YES. The completeness property means that if we obtain an *affirmative* response to the problem, then any computation of the system must be an accepting one.

These concepts can be extended to families of recognizing P systems *with input membrane* in a natural way, but in this case a P system belonging to the family can process several instances of the problem provided that an appropriate input, depending on the instance, is supplied to the system.

**Definition 2.6** *A P system with input is a tuple $(\Pi, V, i_0)$, where:*

- *$\Pi$ is a usual P system, with the alphabet of objects $O$ and initial multisets $w_1, \ldots, w_m$ (associated with membranes labeled by $1, \ldots, m$, respectively).*
- *$V$ is an alphabet strictly contained in $O$ and such that $w_1, \ldots, w_m$ are multisets over $O - V$; $V$ is called input alphabet.*
- *$i_0 \in \{1, 2, \ldots, m\}$ is the label of a distinguished membrane (of input).*

If $w$ is a multiset over $V$, then the initial configuration of $(\Pi, V, i_0)$ with input $w$ is $(\mu, w'_1, \ldots, w'_m)$, where $w'_i = w_i$ for $i \neq i_0$, and $w'_{i_0} = w_{i_0} \cup w$.

The computations of a P system with input are defined in a natural way, the only change is that the initial configuration is obtained by adding the input multiset $w$ over $V$ to the initial configuration of the system $\Pi$.

**Definition 2.7** *Let $X = (I_X, \theta_X)$ be a decision problem. Let $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$ be a family of recognizing membrane systems* with input. *A* polynomial encoding *of $X$ in $\mathbf{\Pi}$ is a pair $(cod, s)$ of polynomial time computable functions over $I_X$ such that for each instance $w \in I_X$, $s(w)$ is a natural number and $cod(w)$ is an input multiset of the system $\Pi(s(w))$.*

**Definition 2.8** *Let $X = (I_X, \theta_X)$ be a decision problem. Let $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$ be a family of recognizing membrane systems* with input. *Let $(cod, s)$ be a polynomial encoding of $X$ in $\mathbf{\Pi}$.*

- *We say that the family $\mathbf{\Pi}$ is sound with regard to $(X, cod, s)$ if the following is true: for each instance of the problem $w \in I_X$, if there exists an accepting computation of $\Pi(s(w))$ with input $cod(w)$, then $\theta_X(w) = 1$.*
- *We say that the family $\mathbf{\Pi}$ is complete with regard to $(X, cod, s)$ if the following is true: for each instance of the problem $u \in I_X$, if $\theta_X(u) = 1$ then every computation of $\Pi(s(u))$ with input $cod(u)$ is an accepting computation.*

The soundness property means that if given an instance we obtain an *acceptance response* of the system associated with it (and individualized by the appropriate input multiset) through some computation, then the answer of the problem (for that instance) is *yes*. The completeness property means that if we obtain an *affirmative* response to the problem, then any computation of the system associated with it (and individualized by the appropriate input multiset) must be an accepting one.

Next, we consider different polynomial complexity classes in the framework of recognizing membrane systems.

We define polynomial complexity classes in recognizing membrane systems *without input*. In order to solve a decision problem we need, then, to associate with each instance of the problem a system which decides the instance. We impose these systems to be *confluent* in the following sense: an instance of the problem will have a positive answer if and only if *every* (or, equivalently, there exists a) computation of the corresponding system is an accepting computation.

We also demand that *every* computation is bounded, in execution time, by a polynomial function. This is because we do not only want to obtain the same answer, independently of the chosen computation, but that all the computations consume, at most, the same amount of resources (in time).

**Definition 2.9** *Let $\mathcal{R}$ be a class of recognizing P systems without input membrane. A decision problem $X = (I_X, \theta_X)$ is solvable in polynomial time by a family $\mathbf{\Pi} = (\Pi(w))_{w\in I_X}$, of P systems of type $\mathcal{R}$, and we denote it by $X \in \mathbf{PMC}^*_{\mathcal{R}}$, if the following is true:*

- *The family* $\mathbf{\Pi}$ *is polynomially uniform by Turing machines; that is, there exists a deterministic Turing machine working in polynomial time which constructs the system* $\Pi(w)$ *from the instance* $w \in I_X$.

- *The family* $\mathbf{\Pi}$ *is polynomially bounded; that is, there exists a polynomial function* $p(n)$ *such that for each* $w \in I_X$, *all computations of* $\Pi(w)$ *halt in at most* $p(|w|)$ *steps.*

- *The family* $\mathbf{\Pi}$ *is sound and complete with regard to* $X$.

We say that $\mathbf{\Pi}$ provides a *semi–uniform* solution to the problem $X$.

Note that in this complexity class we consider two different tasks: the first one is the construction of the family, which we require to be done in polynomial time (sequential time by deterministic Turing machines). The second one is the execution of the systems of the family, in which we imposed that the total number of steps performed by their computations are bounded by the function $g$ (parallel time by non-deterministic membrane systems).

As a direct consequence of working with recognizing membrane systems is the fact that these complexity classes are closed under complement.

Now we define another polynomial complexity classes in recognizing membrane systems *with input.*

A computation of a Turing machine starts when the machine is in the initial state and we "write" a string in the input tape of the machine. Then, the machine starts to compute according to the transition function. In the definitions of basic P systems that have been initially considered, there is no membrane in which we can "introduce" input objects before allowing the system to begin to work. However, it is easy to consider input membranes in this kind of devices.

In the framework of recognizing membrane systems *with an input membrane*, we propose to solve hard problems in an *uniform* way in the following sense: all instances of a decision problem that have the same *size* (according to a prefixed polynomial time computable criterion) are processed by the same system, to which an appropriate input, that depends on the specific instance, is supplied.

Now, we formalize these ideas in the following definition.

**Definition 2.10** *Let* $X = (I_X, \theta_X)$ *be a decision problem. We say that* $X$ *is solvable in polynomial time by a family of recognizing membrane systems* with input $\mathbf{\Pi} = (\Pi(n))_{n\in\mathbb{N}}$, *and we denote it by* $X \in \mathbf{PMC}_{\mathcal{R}}$, *if the following is true:*

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

- *There exists a polynomial encoding* $(cod, s)$ *of* $X$ *in* $\mathbf{\Pi}$ *such that:*

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

We say that $\mathbf{\Pi}$ provides a *uniform* solution to the problem $X$.

Note that in the above definition and in order to decide about an instance, $w$, of a decision problem, first of all we need to compute the natural number $s(w)$, obtain the input multiset $cod(w)$, and construct the system $\Pi(s(w))$. This is properly a *pre-computation stage*, running in polynomial time expressed by a number of *sequential steps* in the framework of the Turing machines. After that, we execute the system $\Pi(s(w))$ with input $cod(w)$. This is properly the *computation stage*, also running in polynomial time, but now it is described by a number of *parallel steps*, in the framework of membrane computing. As mentioned above, these complexity classes are closed under complement.

### 2.3.3 P Systems with Active Membranes

In membrane computing, P systems with active membranes have a special place, because they provide biologically inspired tools to solve computationally hard problems. Using the possibility to divide or separate membranes, one can create an exponential working space in linear time, which can then be used in a parallel computation for solving, e.g., **NP**-complete problems in polynomial or even linear time. We give below the definition of P systems with active membranes following [80].

**Definition 2.11** *A P system with active membranes (and electrical charges) is a construct*

$$\Pi = (O, H, \mu, w_1, \ldots, w_m, R),$$

*where:*

1. $m \geq 1$ *(the initial degree of the system);*
2. $O$ *is the alphabet of objects;*
3. $H$ *is a finite set of labels for membranes;*
4. $\mu$ *is a membrane structure, consisting of* $m$ *membranes, labeled (not necessarily in a one-to-one manner) with elements of* $H$*;*
5. $w_1, \ldots, w_m$ *are strings over* $O$*, describing the multisets of objects placed in the* $m$ *regions of* $\mu$*;*
6. $R$ *is a finite set of developmental rules, of the following forms:*

(a) $[\ a \to v]^e_h$,
*for* $h \in H, e \in \{+,-,0\}, a \in O, v \in O^*$
*(object evolution rules, associated with membranes and depending on the label and the charge of the membranes, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them);*

(b) $a[\ ]^{e_1}_h \to [\ b]^{e_2}_h$,
*for* $h \in H, e_1, e_2 \in \{+,-,0\}, a, b \in O$
*(communication rules; an object is introduced in the membrane, possibly modified during this process; also the polarization of the membrane can be modified, but not its label);*

(c) $[\ a\ ]^{e_1}_h \to [\ ]^{e_2}_h b$,
*for* $h \in H, e_1, e_2 \in \{+,-,0\}, a, b \in O$
*(communication rules; an object is sent out of the membrane, possibly modified during this process; also the polarization of the membrane can be modified, but not its label);*

(d) $[\ a\ ]^e_h \to b$,
*for* $h \in H, e \in \{+,-,0\}, a, b \in O$
*(dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);*

(e) $[\ a\ ]^{e_1}_h \to [\ b\ ]^{e_2}_h [\ c\ ]^{e_3}_h$,
*for* $h \in H, e_1, e_2, e_3 \in \{+,-,0\}, a, b, c \in O$
*(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label, possibly of different polarizations; the object specified in the rule is replaced in the two new membranes by possibly new objects);*

(f) $[\ [\ ]^{\alpha_1}_{h_1} \cdots [\ ]^{\alpha_1}_{h_k}\ [\ ]^{\alpha_2}_{h_{k+1}} \cdots [\ ]^{\alpha_2}_{h_n}]^{\alpha_0}_{h_0}$
$\to [\ [\ ]^{\alpha_3}_{h_1} \cdots [\ ]^{\alpha_3}_{h_k}]^{\alpha_5}_{h_0}\ [\ [\ ]^{\alpha_4}_{h_{k+1}} \cdots [\ ]^{\alpha_4}_{h_n}]^{\alpha_6}_{h_0}$,
*for* $k \geq 1, n > k, h_i \in H, 0 \leq i \leq n$, *and* $\alpha_0, \ldots, \alpha_6 \in \{+,-,0\}$ *with* $\{\alpha_1, \alpha_2\} = \{+,-\}$*; if the membrane with the label* $h_0$ *contains other membranes than those with the labels* $h_1, \ldots, h_n$ *specified above, then they must have neutral charges in order to make this rule applicable; these membranes are duplicated and then are part of the contents of both new copies of the membrane* $h_0$
*(division of non-elementary membranes; this is possible only if a membrane contains two immediately lower membranes of opposite polarization,* $+$ *and* $-$*; the membranes of opposite polarizations are separated in the two new membranes, but their polarization can change; always, all membranes of opposite polarizations are separated by applying this rule).*

The rules of type $(a)$ are applied in the parallel way (all objects which can evolve by such a rule should do it), while the rules of types $(b), (c), (d), (e), (f)$ are used sequentially, in the sense that one membrane can be used by at most one rule of these types at a time. In total, the rules are used in the non-deterministic maximally parallel manner (all objects and all membranes which can evolve, should evolve) and in a bottom-up manner (for instance, first rules of type $(a)$ are used, and after that the membrane can be divided by rules of types $(e)$, $(f)$; thus, the result of using evolution rules is replicated in the newly produced membranes). The skin membrane can never divide, although, as the rest of membranes, it can be electrically charged. Only halting computations give a result, in this form of the multiset of objects sent into the environment during the computation; non-halting computations give no output.

### 2.3.4 Spiking Neural P Systems

A *spiking neural P system* consists of a set of neurons placed in the nodes of a graph, and communicating through signals (spikes) emitted along the synapses (edges of the graph) and controlled by firing and forgetting rules. Within the system the spikes are moved, created, or deleted (we consider only one type of objects in the system).

**Definition 2.12** *A spiking neural P system of degree $m \geq 1$ is a construct of the form $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_0)$, where:*

1. *$O = \{a\}$ is the singleton alphabet ($a$ is called spike);*
2. *$\sigma_1, \ldots, \sigma_m$ are neurons, of the form $\sigma_i = (n_i, R_i), 1 \leq i \leq m$, where:*
   a) *$n_i \geq 0$ is the initial number of spikes present in the neuron;*
   b) *$R_i$ is a finite set of rules of the following two forms:*
      (1) *$E/a^r \rightarrow a; d$, where $E$ is a regular expression over $O$, $r \geq 1$, and $d \geq 0$;*
      (2) *$a^s \rightarrow \lambda$, for some $s \geq 1$, with the restriction that $a^s \notin L(E)$ for any rule $E/a^r \rightarrow a; d$ of type (1) from $R_i$;*
3. *$syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (synapses among cells);*
4. *$i_0 \in \{1, 2, \ldots, m\}$ indicates the output neuron.*

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron $\sigma_i$ contains $k$ spikes, and $a^k \in L(E), k \geq c$, then the rule $E/a^c \rightarrow a; d$ can be applied. The application of this rule means consuming (removing) $c$ spikes (thus only $k - c$ remain in $\sigma_i$), the neuron is

fired, and it produces a spike after $d$ time units (a global clock is assumed, marking the time for the whole system, hence the functioning of the system is synchronized).

If $d = 0$, then the spike is emitted immediately, if $d = 1$, then the spike is emitted in the next step, etc. In the steps before emitting the spike, the neuron is *closed*, it cannot use any rule and cannot receive spikes from other neurons. The delay in sending the spike along the synapses models the refractory period of the neuron from neurobiology – more details on its formalization can be found in [49].

A spike emitted by a neuron $\sigma_i$ (is replicated and a copy of it) goes to each neuron $\sigma_j$ such that $(i, j) \in syn$.

The rules of type (2) are *forgetting* rules and they are applied as follows: if the neuron $\sigma_i$ contains exactly $s$ spikes, then the rule $a^s \to \lambda$ from $R_i$ can be used, meaning that all $s$ spikes are removed from $\sigma_i$.

If a rule $E/a^c \to a; d$ of type (1) has $E = a^c$, then we will write it in the simplified form $a^c \to a; d$.

In each time unit, if a neuron $\sigma_i$ can use one of its rules, then a rule from $R_i$ *must* be used. Since two firing rules, $E_1/a^{c_1} \to a; d_1$ and $E_2/a^{c_2} \to a; d_2$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a neuron, and in that case only one of them is chosen non-deterministically. By definition, if a firing rule is applicable, then no forgetting rule is applicable, and vice versa.

The initial configuration of the system is described by the numbers $n_1, n_2, \ldots, n_m$, of spikes present in each neuron, with all neurons being open. During the computation, a configuration is described by both the number of spikes present in each neuron and by the state of the neuron, more precisely, by the number of steps from now on until it becomes open (this number is zero if the neuron is already open). Thus, $\langle r_1/t_1, \ldots, r_m/t_m \rangle$ is the configuration where neuron $\sigma_i, i = 1, 2, \ldots, m$ contains $r_i \geq 0$ spikes and it will be open after $t_i \geq 0$ steps; with this notation, the initial configuration is $C_0 = \langle n_1/0, \ldots, n_m/0 \rangle$.

Using the rules as described above, one can define transitions among configurations. A transition between two configurations $C_1, C_2$ is denoted by $C_1 \Longrightarrow C_2$. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where all neurons are open and no rule can be used.

In the spirit of spiking neurons, see, e.g., [62], as the result of a computation, in [49] and [83] one considers the distance between two consecutive spikes which exit the output neuron of the system. Then, in [18] one considers as the result of a computation the so-called spike train of the computation, the sequence of symbols 0 and 1 obtained by associating 1 with a step when a spike exits the system and 0 otherwise. Languages over the binary alphabet are computed in this way.

# Chapter 3

# Using Cell Interaction Operations in P Systems

Cell division is a well-known phenomenon in cell biology. The cell division operation in P systems plays in a crucial role for generating exponential work space in linear time, hence, it is a useful tool to solve computationally hard problems. Tom Head suggested in [46] other membrane handling rules that can be considered in membrane systems, such as merging or separating membranes, etc. Several operation with membranes also appear in Brane calculi, see [13].

Here we consider the following cell membrane interaction operations to formalize in P systems area:

*Membrane merging*: Membrane fusion (merging) is a well-known phenomenon of cell biology. It means that two membranes can be merged into a single membrane, the contents of the former membranes being put together in the new membrane. In order to perform a merging operation two membranes must be adjacent, both of them placed in the same immediately upper membrane.

*Membrane separation*: Membrane fission (budding, separation) is also a well-known phenomenon in cell biology, with interesting applications in bio-technology being developed. To separate the contents of a membrane into two membranes, some of its objects are placed in the first membrane (according to a given property), and the objects which do not have the given property are placed in the other membrane.

*Membrane release*: Chemicals release into cleft of connected neuronal synapses in vesicle formation.

*Replication and distribution*: The neural impulse is replicated in the cleft and distributed into the connected dendrites. Moreover, in the axon of the neuron, chemicals are replicated at the so-called Ranvier

nodes and transmitted to the adjacent nodes in opposite directions through the axon (see Subsection 1.2).

The operations abstracted from the cell interaction processes mentioned above are formalized in the framework of P systems with active membranes. We investigate their computational power and efficiency in both maximally and minimally parallel way of using the rules. The results are as expected: P systems using the combinations of the new operations with old types of operations have computational generative and accepting power of universal Turing machines, and provide efficient algorithms for solving **NP**-complete problems in polynomial time (see [4, 73, 72, 71, 50]). We also see that the membrane separation operation plays an important role in generating an exponential work space in linear time in P systems.

## 3.1 Polarizationless P Systems

We define now polarizationless P systems with active membranes following [5]. A P system *with active membranes* (without electrical charges) is a construct

$$\Pi = (O, C, H, \mu, w_1, \ldots, w_m, R),$$

where:

- $m \geq 1$ is the initial degree of the system;
- $O$ is the alphabet of *objects*;
- $C \subseteq O$ is the set of catalysts;
- $H$ is a finite set of *labels* for membranes;
- $\mu$ is a *membrane structure*, consisting of $m$ membranes, labeled (not necessarily in a one-to-one manner) with elements of $H$;
- $w_1, \ldots, w_m$ are strings over $O$, describing the *multisets of objects* placed in the $m$ regions of $\mu$;
- $R$ is a finite set of *developmental rules*, of the following forms:

  (ev) $[\, a \to v]_h$, for $h \in H, a \in O, v \in O^*$
  (object evolution rules; associated with membranes and depending on the label, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them);

  (*cat*) $[\, ca \to cv]_h$ for $c \in C, a \in O - C, v \in (O - C)^*, h \in H$
  (catalytic rules; the catalyst $c$ is never modified, it only assists the evolution of other objects);

(*pro*) $[\ a \to v|_b]_h$ for $a, b \in O, v \in O^*, h \in H$
(evolution rule with promoter; the rule is applied only if the promoter object $b$ is present in the region);

(cev) $[\ v \to u]_h$ for $u, v \in O^*$
(evolution rules of radius greater than one; a particular case is that of catalytic rules);

(in) $a[\ ]_h \to [\ b]_h$, for $h \in H, a, b \in O$
(communication rules; an object is introduced in the membrane during this process);

(out) $[\ a\ ]_h \to [\ ]_h b$, for $h \in H, a, b \in O$
(communication rules; an object is sent out of the membrane during this process);

(dis) $[\ a\ ]_h \to b$, for $h \in H, a, b \in O$
(dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule is modified);

(ediv) $[\ a]_h \to [\ b]_h[\ c]_h$, for $h \in H, a, b, c \in O$
(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label; the object specified in the rule is replaced in the two new membranes by possibly new objects; and the remaining objects are duplicated);

(ndiv) $[\ a]_h \to [\ b]_h[\ c]_h$, for $h \in H, a, b, c \in O$
(division rules for non-elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label; the object specified in the rule is replaced in the two new membranes by possibly new objects; the remaining objects and membranes contained in this membrane are duplicated, and then are part of the contents of both new copies of the membrane);

The rules of types ev, in, out, dis, ediv, and ndiv are the polarizationless version of the corresponding rules in Section 2.3.3 and [81] with the mentioning that we use now rules for division of non-elementary membranes of the same form as for dividing elementary membranes (note that we no longer use polarizations of membranes).

When the rules of a given type ($\alpha$) are able to change the label(s) of the involved membranes, we denote that type of rules by ($\alpha'$). For instance, a rule of type (in′) is of the form

$$a[\ ]_h \to [\ b]_{h'}, \text{ for } h, h' \in H, a, b \in O.$$

The rules of type ev are applied in the parallel way (all objects which can evolve by such rules have to evolve), while the rules of types in, out, dis, ediv, ndiv are used sequentially, in the sense that one membrane can be

used by at most one rule of these types at a time. In total, the rules are used in the non-deterministic maximally parallel manner (all objects and all membranes which can evolve, should evolve) and in a bottom-up way (first we use the rules of type ev, and then the rules of other types; in this way, in the case of dividing membranes, in the newly obtained membranes we duplicate the result of using first the rules of type ev).

Only halting computations give a result, in the form of the number (or the vector) of objects expelled into the environment during the computation. The set of numbers generated by a system $\Pi$ is denoted by $N_{gen}(\Pi)$ and the family of such sets is denoted by $N_{gen}OP(\textit{types-of-rules})$, with *types-of-rules* indicating the allowed types of rules. A P system can be also used in the accepting mode: we introduce a number in the system in the form of a multiset $a^n$, for some $a \in O$, in a distinguished region $h_0$ and start computing; if the system halts, then the number $n$ is accepted. The set of all numbers accepted in this way by $\Pi$ is denoted by $N_{acc}(\Pi)$, and the family of such sets is denoted by $N_{acc}OP(\textit{types-of-rules})$, with the obvious meaning of the used parameters. A P system is called deterministic if for every input there is a single computation. A P system is called confluent if all of its computations reach the same halting configuration. When using systems with at most $r$ catalysts, we write $cat_r$ for the respective type of rules.

### 3.1.1 Formalization of Membrane Merging, Separation, and Release Operations

We mathematically catch the behavior of the cell interaction operations and formalize them in the framework of P systems with active membranes without polarizations in the following way.

*Membrane merging:*

(mer) $[\,]_h[\,]_h \to [\,]_h$, for $h \in H$
(merging rules for elementary membranes; the two membranes are merged into a single membrane under the control of the label; the objects of the former membranes are put together in the new membrane).

*Membrane separation 1:*

(sep) $[\,Q]_h \to [\,K]_h[\,Q-K]_h$, for $h \in H, Q \subseteq O^*, K \subset Q$, $\forall x \in Supp(K)\ (K(x) = Q(x))$
(separation rules for elementary membranes; the membrane $h$, containing objects from $Q$, is separated into two membranes with the same labels; the objects from $K$ are placed in the first membrane, those from $Q-K$ are placed in the other membrane; we request that both $K$ and $Q-K$ are

not empty and also that both membranes $[\ K]_h$, $[\ Q-K]_h$ are non-empty – the rule is not applied otherwise).

*Membrane separation 2:*

(sep$_2$) $[\ a]_h \to [\ O_1]_h[\ O_2]_h$,
for $h \in H$, $a \in O$, $O_1 \cup O_2 = O$, $O_1 \cap O_2 = \emptyset$
(separation rules for elementary membranes; in reaction with an object, the membrane is separated into two membranes with the same label; at the same time, the object $a$ can evolve by an evolution; the objects from $O_1$ are placed in the first membrane, those from $O_2$ are placed in the second membrane).

*Membrane release:*

(rel) $[\ [\ Q]_h]_h \to [\ ]_h Q$, for $h \in H$, $Q \subseteq O^*$
(release rule; the objects in a membrane are released out of a membrane, surrounding it, while the first membrane disappears).

The label changing versions of merging and separation rules are of the following forms:

(mer′) $[\ ]_{h_1}[\ ]_{h_2} \to [\ ]_{h_3}$, for $h_1, h_2, h_3 \in H$.

(sep′) $[\ Q]_{h_1} \to [\ K]_{h_2}[\ Q-K]_{h_3}$, for $h_1, h_2, h_3 \in H, Q \subseteq O^*, K \subset Q$.

### 3.1.2 Two Universality Results

The following theorem shows that by using membrane separation rules (of type 1) to change the labels of the membranes, the universality can be reached. Here $Ps(\Pi)$ denotes the set of vectors of natural numbers describing the multiplicity of objects expelled into the environment by the various halting computations in system $\Pi$; by $PsOP(types\text{-}of\text{-}rules)$ we denote the family of sets $Ps(\Pi)$ computed by P systems using the types of rules specified by *types-of-rules*.

**Theorem 3.1** $PsOP(cat_1, \mathsf{out}, \mathsf{sep}') = PsRE$.

*Proof.* Consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the binary normal form, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$ and with the matrices of the four forms introduced in Section 2.1.3. Assume that all matrices of forms 2, 3, and 4 are injectively labeled with elements of a set $B$ (without loss of generality, suppose $0 \notin B$), $B = B_1 \cup B_2$, $B_1$ for the

matrices of forms 2 and 4, and $B_2$ for the matrices of form 3. Replace the rule $X \to \lambda$ from matrices of type 4 by $X \to f$, where $f$ is a new symbol.

We construct the P system of degree 2

$$\begin{aligned}
\Pi &= (O, C, H, [\ [\ ]_{X_{init}}]_1, w_1 = \lambda, w_{X_{init}} = cc_0 dA_{init}, R), \\
O &= T \cup N_2 \cup \{c_0, c_1, c_2, c_3, c_4, \#\}, \\
C &= \{c\}, \\
H &= N_1 \cup \{X_m, X'_m \mid X \in N_1, m \in B\} \cup \{0, 1, f\},
\end{aligned}$$

and the set $R$ containing the following rules. We present them in blocks as used for simulating matrices of $G$, thus also having clear the way the system $\Pi$ works.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1, Y \in N_1 \cup \{f\}$, $A \in N_2$, and $x \in (N_2 \cup T)^*$, $|x| \leq 2$, is done in four steps, using the next rules:

1. $[\ c_0 \to c_1 c_2 c_3]_X$.
2. $[\ O]_X \to [\ \{c_1\}]_0 [\ O - \{c_1\}]_{X_m}$.
3. $[\ cA \to cx]_{X_m}$,
   $[\ c_3 \to c_0 c_4]_{X_m}$,
   $[\ cd \to c\#]_{X_m}$,
   $[\ O]_{X_m} \to [\ \{c_2\}]_0 [\ O - \{c_2\}]_{X'_m}$.
4. $[\ O]_{X'_m} \to [\ \{c_4\}]_0 [\ O - \{c_4\}]_Y$.

In the membrane labeled with the nonterminal $X$ we evolve $c_0$ into the auxiliary objects $c_1, c_2, c_3$, then, using $c_1$ we separate the objects, also guessing the matrix $m$ to simulate. All relevant objects are placed in the membrane with label $X_m$. In step 3, in this membrane we both simulate the second rule of the matrix (if no copy of $A$ is present, hence the rule $[\ cA \to cx]_{X_m}$ cannot be used, then the trap object is introduced by the rule $[\ cd \to c\#]_{X_m}$, hence the simulation of the matrix should be complete) and evolve $c_3$ to $c_0$ and $c_4$. At the same time, the membrane is separated again, with all relevant objects placed in a membrane with label $X'_m$. In the fourth step, this last membrane is separated again, by using the "separator" $c_4$, and thus the membrane with label $Y$ is introduced; it contains again the object $c_0$, hence we return to a configuration as that we have started with.

The simulation of a matrix $m : (X \to Y, A \to \#)$, with $X, Y \in N_1$ and $A \in N_2$, is done in a very similar way, using again four steps and the same rules as above (of course, with $X_m, X'_m$ associated with the new matrix), the only differences being that (i) the rule $[\ cA \to cx]_{X_m}$ used in step 3 is now replaced with the rule $[\ A \to \#]_{X_m}$, and (ii) the rule $[\ cd \to c\#]_{X_m}$ is no longer provided. In this way, if any copy of $A$ appears in membrane $X_m$,

then the trap object is introduced and the computation never halts. If $A$ is not present, then both $c$ and $d$ waits unused, then the simulation of the rule $X \to Y$ of the matrix is completed.

These procedures can be iterated.

We also consider the following rules:

5. $[\ A \to \#]_f$, for all $A \in N_2$.
6. $[\ \# \to \#]_h$, for all $h \in H$.
7. $[\ a]_f \to [\ ]_f a$, for all $a \in T$.
8. $[\ a]_1 \to [\ ]_1 a$, for all $a \in T$.

The equality $\Psi_T(L(G)) = Ps(\Pi)$ easily follows from the above explanations. □

**Remark 3.1** *In the above proof, the rules of type* out *are only used for sending the result of a computation out of the system. Therefore, rules of types* ev *and* sep′ *are sufficient to reach universality for membrane systems with internal output.*

The following theorem shows that the universality can be reached also by using membrane separation rules of type 2 to change the labels of the membranes.

**Theorem 3.2** $PsOP(cat_1, \mathsf{out}, \mathsf{sep}_2') = PsRE$.

*Proof.* Again we consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the binary normal form, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$. Assume that all matrices are injectively labeled with elements of a set $B$. Replace the rule $X \to \lambda$ from matrices of type 4 by $X \to f$, where $f$ is a new symbol.

We construct the P system of degree 2

$$\begin{aligned}
\Pi &= (O, C, H, [\ [\ ]_{X_{init}}]_1, w_1 = \lambda, w_{X_{init}} = c_0 A_{init}, R), \\
O &= O_1 \cup O_2, \\
O_1 &= T \cup N_2 \cup \{c_0, c_1, c_2, c_3 \#\}, \\
O_2 &= \{d\}, \\
C &= \{c\}, \\
H &= N_1 \cup \{X_m \mid X \in N_1, m \in B\} \cup \{0, 1, f\},
\end{aligned}$$

and the set $R$ containing the following rules.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1, Y \in N_1 \cup \{f\}$, is done in three steps, using the next rules:

1. $[\ A]_X \to [\ O_1]_{Y_m}[\ O_2]_0$.
2. $[\ cA \to xc_3d]_{Y_m}$.
3. $[\ c_3]_{Y_m} \to [\ O_1]_Y[\ O_2]_0$,
   $[\ c_3 \to d]_{Y_m}$.
4. $[\ c_1 \to c_2]_{Y_m}$,
   $[\ c_2 \to c_0]_{Y_m}$.

The first rule of the matrix is simulated by the change of the label of the inner membrane (the "dummy" object $d$ and membrane 0 play no further role). Note that if $X \in N_1$ also appears in a matrix of type 3, then when $A$ activates separation, $c_0$ can evolve to $c_1d$ by the second rule of type (5), then to $c_2$ and $c_0$ by the rules of type (4). The correctness of the simulation is obvious, because one cannot simulate one rule of the matrix without simulating the other rule.

The simulation of a matrix $m : (X \to Y, A \to \#)$, with $X, Y \in N_1$ and $A \in N_2$, is done also in three steps, using the next rules:

5. $[\ c_0]_X \to [\ O_1]_{Y_m}[\ O_2]_0$,
   $[\ c_0 \to c_1d]_X$.
6. $[\ c_1 \to c_2]_{Y_m}$,
   $[\ A \to \#]_{Y_m}$.
7. $[\ c_2]_{Y_m} \to [\ O_1]_Y[\ O_2]_0$,
   $[\ c_2 \to c_0d]_{Y_m}$.

While the membrane with label $X$ is used by object $c_0$, no other rule can be used. In the next step, if any copy of $A$ is present, then it introduces the trap-object $\#$ and the computation never stops. If no $A$ is present, then the objects $c_j$ evolve, returning the label of the membrane to $Y$ and introducing the auxiliary object $c_0$, for iterating the procedure.

We also consider the following rules:

8. $[\ A \to \#]_f$, for all $A \in N_2$.
9. $[\ \# \to \#]_h$, for all $h \in H$.
10. $[\ a]_f \to [\ ]_f a$.
11. $[\ a]_1 \to [\ ]_1 a$, for all $a \in T$.

The equality $\Psi_T(L(G)) = Ps(\Pi)$ easily follows from the above explanations. □

### 3.1.3 Two Efficiency Results

**Theorem 3.3** *A uniform family of P systems with rules of types* ev, in, out, sep′ *can solve* SAT *in linear time with respect to the number of variables and the number of clauses in a confluent way.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form:

$$\begin{aligned} \varphi &= C_1 \wedge \cdots \wedge C_m, \\ Var(\varphi) &= \{x_1, \ldots, x_n\}, \\ C_i &= y_{i,1} \vee \cdots \vee y_{i,l_i},\ 1 \leq i \leq m, \text{ where} \\ y_{i,k} &\in \{x_j, \neg x_j \mid 1 \leq j \leq n\},\ 1 \leq i \leq m, 1 \leq k \leq l_i. \end{aligned}$$

The instance $\varphi$ of SAT (to which the size $(m,n)$ is associated) is encoded as a multiset of objects $x_{i,j,j}$ and $\bar{x}_{i,j,j}$, where $x_{i,j,j}$ represents the variable $x_j$ appearing in the clause $C_i$, and where $\bar{x}_{i,j,j}$ represents the literal $\neg x_j$ appearing in the clause $C_i$. Thus, the input multiset is

$$\begin{aligned} cod(w) &= \{x_{i,j,j} \mid x_j \in \{y_{i,k} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{\bar{x}_{i,j,j} \mid \neg x_j \in \{y_{i,k} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\}. \end{aligned}$$

For a given $(n,m) \in \mathbb{N}^2$ being $\langle n,m \rangle = (n+m)(n+m+1)/2$, a recognizing P system $(\Pi(\langle n,m \rangle), \Sigma(\langle n,m \rangle), 1)$ is constructed as follows:

$$\begin{aligned} \Pi(\langle n,m \rangle) &= (O(\langle n,m \rangle), H, \mu, w_0, w_1, w_s, R), \text{ with} \\ O(\langle n,m \rangle) &= \{c_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n+1\} \\ &\cup \{c'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \cup \{t, \texttt{YES}, \texttt{NO}\} \\ &\cup \{d_i \mid 0 \leq i \leq n+m+7\} \cup \{d'_i \mid 1 \leq i \leq n+1\} \\ &\cup \{x_{i,t,j}, \bar{x}_{i,t,j} \mid 1 \leq i \leq m, 1 \leq j \leq n, 0 \leq t \leq j\} \\ &\cup \{x'_{i,t,j}, \bar{x}'_{i,t,j} \mid 1 \leq i \leq m, 1 \leq j \leq n, 0 \leq t \leq j-1\}, \\ \Sigma(\langle n,m \rangle) &= \{x_{i,t,j}, \bar{x}_{i,t,j} \mid 1 \leq i \leq m, 1 \leq j \leq n, 0 \leq t \leq j\}, \\ \mu &= [\,[\,]_0[\,]_1]_s, \\ w_s &= \lambda, w_0 = w_1 = d_0, \\ H &= \{s, 0, 1, \cdots, m+2\}, \end{aligned}$$

The set $R$ of rules of P systems $\Pi(\langle n,m \rangle)$, to which we also give explanations about, are the following:
*Generation phase:*

g1. $[\,O]_1 \rightarrow [\,U]_1[\,O-U]_1$,
$U = \{x'_{i,t,j}, \bar{x}'_{i,t,j} \mid 1 \leq i \leq m, 1 \leq j \leq n, 0 \leq t \leq j-1\} \cup$
$\{c'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \cup \{d'_i \mid 1 \leq i \leq n+1\}$.

g2. $[\ x_{i,t,j} \to x_{i,t-1,j}x'_{i,t-1,j}]_1$,
$[\ \bar{x}_{i,t,j} \to \bar{x}_{i,t-1,j}\bar{x}'_{i,j-1,j}]_1$,
$[\ x'_{i,t,j} \to x_{i,t-1,j}x'_{i,t-1,j}]_1$,
$[\ \bar{x}'_{i,t,j} \to \bar{x}_{i,t-1,j}\bar{x}'_{i,t-1,j}]_1$,
$1 \leq i \leq m,\ 1 \leq t \leq j,\ 1 \leq j \leq n.$

g3. $[\ x_{i,0,j} \to c_{i,j}c'_{i,j}]_1$,
$[\ \bar{x}_{i,0,j} \to \lambda]_1$,
$[\ \bar{x}'_{i,0,j} \to c_{i,j}c'_{i,j}]_1$,
$[\ x'_{i,0,j} \to \lambda]_1,\ 1 \leq i \leq m,\ 1 \leq j \leq n.$

g4. $[\ c_{i,j} \to c_{i,j+1}c'_{i,j+1}]_1$,
$[\ c'_{i,j} \to c_{i,j+1}c'_{i,j+1}]_1$,
$1 \leq i \leq m,\ 1 \leq j \leq n-1.$

g5. $[\ c_{i,n} \to c_{i,n+1}]_1$,
$[\ c'_{i,n} \to c_{i,n+1}]_1,\ 1 \leq i \leq m.$

g6. $[\ d_i \to d_{i+1}d'_{i+1}]_1,\ 0 \leq i \leq n,$
$[\ d'_i \to d_{i+1}d'_{i+1}]_1,\ 1 \leq i \leq n,$
$[\ d_{n+1} \to d_{n+2}]_1$,
$[\ d'_{n+1} \to d_{n+2}]_1$.

In $n+1$ steps, $2^n$ membranes with label 1 are created by rules of types g1–g4. These membranes correspond to all the possible $2^n$ truth assignments of the variables $x_1, x_2, ..., x_n$. During this process, every object $x_{i,j,j}$ of the input evolves to $x_{i,0,j}$ and $x'_{i,0,j}$ in $j$ steps. Then, they evolve to $c_{i,j}$ in membranes where *true* value was chosen for $x_j$ (recall that $x_{i,j,j} = true$ satisfies clause $C_i$) and they are erased in membranes where *false* value was chosen for $x_j$. In the next $n-j$ steps, $c_{i,j}$ evolves to $c_{i,n}$ or $c'_{i,n}$. It takes one more step for $c_{i,n}$ and $c'_{i,n}$ to evolve to $c_{i,n+1}$ by rules of type g5, which means the system is ready for the checking phase. Similarly, $\bar{x}_{i,j,j}$ changes to $c_{i,n+1}$ if $x_j = false$ and it is erased if $x_j = true$. Note that at step $n+2$, each membrane with label 1 contains the object $d_{n+2}$.

*Checking phase:*

c1. $[\ O]_i \to [\ U]_i[\ O-U]_{i+1},\ U = \{c_{i,n+1}\}, 1 \leq i \leq m.$

Starting with $i = 1$, in membranes with label $i$, objects $c_{i,n+1}$ will be separated from the other objects, and the label of the membrane with objects from $O - \{c_{i,n+1}\}$ will become $i+1$. The membranes which do not contain the objects $c_{i+1,n+1}$ will evolve nomore. If all objects $c_{i,n+1}, 1 \leq i \leq m$, are present in a certain membrane, then, after $m$ steps, this membrane will evolve into a membrane with label $m+1$, containing object $d_{n+2}$, by the rules of type $c1$.

c2. $[\ d_{n+2}]_{m+1} \to [\ ]_{m+1}d_{n+2}$.

c3. $[\ d_{n+2} \to tt]_s$.

If $\varphi$ has solutions (that is, truth assignments making true the formula), then at step $n+m+3$, every membrane corresponding to a solution of $\varphi$ ejects $d_{n+2}$ to the skin region, then $d_{n+2}$ will be rewritten into $tt$, by using rules c2 and c3.

c4. $t[\ ]_0 \to [\ t]_0$.

c5. $t[\ ]_{m+1} \to [\ t]_{m+1}$.

c6. $[\ O]_0 \to [\ U']_{m+1}[\ O - U']_{m+2}$, $U' = \{t\}$, $0 \leq i \leq n+m+5$.

c7. $[\ d_{n+m+6} \to d_{n+m+7}]_{m+2}$.

At step $n+m+5$, one copy of $t$ enters the membrane with label 0, and (suppose $\varphi$ has $s$ solutions, $1 \leq s \leq 2^n$) $s$ copies of $t$ enter the $s$ membranes with label $m+1$. At step $n+m+6$, $s-1$ copies of $t$ enter the membranes with label $m+1$, or $s-2$ copies of $t$ enter the $s-2$ membranes with label $m+1$, and 1 copy of $t$ enters the membrane with label 0. Using rule c6, the membrane with label 0 is separated into two membranes, which contain object $t$ and object $d_{n+m+6}$ or $d_{n+m+7}$, respectively. If $\varphi$ has no solution, then no object enters the membrane labeled 0 and rule c6 is not applied.

*Output phase:*

o1. $[\ t \to \lambda]_{m+1}$.

o2. $[\ d_{n+m+7}]_0 \to [\ ]_0$NO.

o3. $[\ d_{n+m+7}]_{m+2} \to [\ ]_{m+2}$YES.

o4. $[\ \mathtt{NO}]_s \to [\ ]_s$NO.

o5. $[\ \mathtt{YES}]_s \to [\ ]_s$YES.

If $\varphi$ has solutions, then at step $n+m+8$, object $d_{n+m+7}$ in the membrane with label $m+2$ ejects YES to skin and then into the environment. If $\varphi$ has no solution, then after $n+m+8$ steps object $d_{n+m+7}$ ejects object NO into skin and then into the environment.

From the previous explanation of the use of rules, one can easily see how the P system designed in the above construction works, and it halts at step $n+m+9$. It is easy to prove that the designed P system is uniform, because it uses $n^2m+4nm+2n+2m+12$ objects, 3 initial membranes containing in total at most $nm+2$ objects, and $6mn+4n+2m+20$ rules. The length of any rule is bounded by $2|O| = 2(4n^2m+4nm+2n+2m+13)$.

But if $\varphi$ has at least two solutions, then the behavior of this system is not deterministic: at step $n+m+6$ either one of the rules of types either c4 or c6 can be applied to the membrane with label 0 (applying c4 in step $2n+2m+6$ results in an extra copy of $t$ in that membrane, and a copy

of $t$ missing in some membranes with label $m+1$). However, the system is confluent: in either case mentioned above, after three further steps, the system produces the output YES and halts in the same configuration (in the membranes with label $m+1$, the objects $t$ are erased). □

## An example

Let us consider the formula $\varphi = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. The initial configuration of the P system $(\Pi(\langle 2,2\rangle), V(\langle 2,2\rangle), 1)$ is:

$$\mu = [\ [\ x_{1,1,1}x_{1,2,2}, \bar{x}_{2,1,1}, \bar{x}_{2,2,2}d_0]_1 [\ d_0]_0]_s.$$

*Generation phase:*

Step 1:

g2. $[\ x_{1,1,1} \to x_{1,0,1}x'_{1,0,1}]_1$,
$[\ x_{1,2,2} \to x_{1,1,2}x'_{1,1,2}]_1$,
$[\ \bar{x}_{2,1,1} \to \bar{x}_{2,0,1}\bar{x}'_{2,0,1}]_1$,
$[\ \bar{x}_{2,2,2} \to \bar{x}_{2,1,2}\bar{x}'_{2,1,2}]_1$,

g6. $[\ d_0 \to d_1 d'_1]_1$,

The objects in membrane 1 evolve by rules from g2 and g6, in a parallel way.

g1. $[\ x_{1,0,1}]_1$, $[\ x'_{1,0,1}]_1$
$[\ x_{1,1,2}]_1$, $[\ x'_{1,1,2}]_1$
$[\ \bar{x}_{2,0,1}]_1$, $[\ \bar{x}'_{2,0,1}]_1$
$[\ \bar{x}_{2,1,2}]_1$, $[\ \bar{x}'_{2,1,2}]_1$
$[\ d_1]_1$, $[\ d'_1]_1$

Membranes with label 1 are separated by rule g1 in the bottom-up manner.

Step 2:

g3. $[\ x_{1,0,1} \to c_{1,1}c'_{1,1}]_1$, $[\ x'_{1,0,1} \to \lambda]_1$
g2. $[\ x_{1,1,2} \to x_{1,0,2}x'_{1,0,2}]_1$, $[\ x'_{1,1,2} \to x_{1,0,2}x'_{1,0,2}]_1$
g3. $[\ \bar{x}_{2,0,1} \to \lambda]_1$, $[\ \bar{x}'_{2,0,1} \to c_{2,1}c'_{2,1}]_1$
g2. $[\ \bar{x}_{2,1,2} \to \bar{x}_{2,0,2}\bar{x}'_{2,0,2}]_1$, $[\ \bar{x}'_{2,1,2} \to \bar{x}_{2,0,2}\bar{x}'_{2,0,2}]_1$
g6. $[\ d_1 \to d_2 d'_2]_1$, $[\ d_1 \to d_2 d'_2]_1$,
g1. $[\ c_{1,1}]_1$, $[\ c'_{1,1}]_1$, $[\ x_{1,0,2}]_1$, $[\ x'_{1,0,2}]_1$
$[\ x_{1,0,2}]_1$, $[\ x'_{1,0,2}]_1$, $[\ c_{2,1}]_1$, $[\ c'_{2,1}]_1$,
$[\ \bar{x}_{2,0,2}]_1$, $[\ \bar{x}'_{2,0,2}]_1$, $[\ \bar{x}_{2,0,2}]_1$, $[\ \bar{x}'_{2,0,2}]_1$
$[\ d_2]_1$, $[\ d'_2]_1$, $[\ d_2]_1$, $[\ d'_2]_1$

Objects evolve according to the rules g2, g3 and g6. Meanwhile, membranes 1 are separated by rules from g1.

Step 3: using rules $g3, g4$, and $g6$.

$$
\begin{array}{ll}
[\, c_{1,1} \to c_{1,2}c'_{1,2}]_1, & [\, c'_{1,1} \to c_{1,2}c'_{1,2}]_1, \\
[\, x_{1,0,2} \to c_{1,2}c'_{1,2}]_1, & [\, x'_{1,0,2} \to c_{1,2}c'_{1,2}]_1, \\
[\, \bar{x}_{2,0,2} \to \lambda]_1, & [\, \bar{x}'_{2,0,2} \to c_{2,2}c'_{2,2}]_1, \\
[\, d_2 \to d_3 d'_3]_1, & [\, d'_2 \to d_3 d'_3]_1, \\
[\, x_{1,0,2} \to c_{1,2}c'_{1,2}]_1, & [\, x'_{1,0,2} \to \lambda]_1, \\
[\, c_{2,1} \to c_{2,2}c'_{2,2}]_1, & [\, c'_{2,1} \to c_{2,2}c'_{2,2}]_1, \\
[\, \bar{x}_{2,0,2} \to \lambda]_1, & [\, \bar{x}'_{2,0,2} \to c_{2,2}c'_{2,2}]_1, \\
[\, d_2 \to d_3 d'_3]_1, & [\, d'_2 \to d_3 d'_3]_1,
\end{array}
$$

The objects in membrane 1 are evolved according to rules g3, g4 and g6; this time no separation happens.

Step 4:

$$
\begin{array}{ll}
\text{g5.}\ [\, c_{1,3}, c_{1,3}]_1, & [\, c_{1,3}, c_{1,3}]_1, \\
[\, c_{1,3}, c_{1,3}]_1, & [\, c_{1,3}, c_{1,3}]_1, \\
[\, \lambda]_1, & [\, c_{2,3}, c_{2,3}]_1, \\
[\, d_4, d_4]_1, & [\, d_4, d'_4]_1, \\
[\, c_{1,3}c_{1,3}]_1, & [\, c_{2,3}, c_{2,3}]_1, \\
[\, c_{2,3}, c_{2,3}]_1, & [\, c_{2,3}, c_{2,3}]_1, \\
[\, d_4, d_4]_1, & [\, d_4, d_4]_1,
\end{array}
$$

Rules g5: $[\, c_{i,2} \to c_{i,3}]_1$, $[\, c'_{i,2} \to c_{i,3}]_1$, and g6 are applied to the membranes with label 1.

Checking phase:

Steps 5,6:

$$
\begin{array}{ll}
\text{c1.}\ [\, c^4_{1,3}]_1, & [\, d^2_4]_2, \\
[\, c^4_{1,3}]_1, & [\, d^2_4 c^2_{2,3}]_2, \\
[\, c_{1,3}c_{1,3}]_1, & [\, d^2_4 c_{2,3}, c_{2,3}]_2, \\
\text{c1.}\ [\, c^2_{2,3}]_2, & [\, d^2_4]_3, \\
[\, c^2_{2,3}]_2, & [\, d^2_4]_3,
\end{array}
$$

At the 5th step, three membranes containing the objects $c_{1,3}$ are separated from the former membrane 1 by rule c1, and rule c1 repeats in the next step separating two membranes with label 2 containing $c_{2,3}$ from the membranes 2.

At the 7th step, two objects $d_4$ are sent out from the membranes 3 by using rule c2. Each one of those objects evolves to $tt$ in the skin membrane by rule c3, in step 8. The counter object $d_j$ was evolving in membrane 0.

The configuration of interest is now $[\,[\,d_8]_0[\;]_3[\;]_3 tttt]_s$. By using rules c4 and c5, the next configuration is reached: $[\,[\,d_9 t]_0[\,t]_3[\,t]_3 t]_s$. Rules c4, c5, and c6 are applicable now. Then, one of the two following configurations is obtained at the end of step 10: $[\,[\,t]_3[\,d_{10}]_4[\,tt]_3[\,t]_3]_s$ by rules c5 and c6, or $[\,[\,d_{10}tt]_0[\,t]_3[\,t]_3]_s$ by rule c4. In the last case, the separation rule c6 is applied to membrane 0 for the next time. In both cases, the YES answer will be sent to the environment after three steps.

Several efficiency results using membrane separation rules of type 2, with polarization or without polarization of membranes, are represented in [73]; we present here only the next result for the case of polarizationless systems.

**Theorem 3.4** *Polarizationless P systems with rules of types* $\mathsf{ev}, \mathsf{out}', \mathsf{sep}_2$ *can solve* SAT *in linear time with respect to the number of variables and the number of clauses in a uniform and deterministic way.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form:

$$\begin{aligned}
\varphi &= C_1 \wedge \cdots \wedge C_m, \\
Var(\varphi) &= \{x_1, \ldots, x_n\}, \\
C_i &= y_{i,1} \vee \cdots \vee y_{i,l_i},\ 1 \leq i \leq m, \text{ where} \\
y_{i,k} &\in \{x_j, \neg x_j \mid 1 \leq j \leq n\},\ 1 \leq i \leq m, 1 \leq k \leq l_i.
\end{aligned}$$

As in the previous proof, the instance $\varphi$ is encoded as a multiset

$$\begin{aligned}
cod(w) &= \{x_{i,j} \mid x_j \in \{y_{i,k} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\} \\
&\cup \{\bar{x}_{i,j} \mid \neg x_j \in \{y_{i,k} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\}.
\end{aligned}$$

For a given $(n, m) \in \mathbb{N}^2$, we construct a recognizing P system $(\Pi(\langle n, m\rangle), \Sigma(\langle n, m\rangle), 2)$, with

$$\begin{aligned}
\Pi(\langle n, m\rangle) &= (O(\langle n, m\rangle), H, \mu, w_1, w_2, R), \\
O(\langle n, m\rangle) &= O_1 \cup O_2, \\
O_1 &= \{x_{i,j}, \bar{x}_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \\
&\cup \{e_i, \bar{e}_i \mid 0 \leq i \leq n-1\} \cup \{c_{i,0}, c_{i,1}, c_{i,2} \mid 0 \leq i \leq m\} \\
&\cup \{d_i \mid 0 \leq i \leq 4n + 2m + 2\} \cup \{a, u, \bar{u}, v, \bar{v}, t, \lambda, \mathtt{YES}, \mathtt{NO}\}, \\
O_2 &= \{x'_{i,j}, \bar{x}'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \\
&\cup \{\bar{e}'_i \mid 0 \leq i \leq n-1\} \cup \{f, \bar{u}', \bar{v}'\}, \\
\Sigma(\langle n, m\rangle) &= \{x_{i,j}, \bar{x}_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}, \\
\mu &= [\,[\,]_2]_1, \\
w_1 &= w_2 = d_0, \\
H &= \{1, 2, 3, 4, 5, 6, 7\},
\end{aligned}$$

and the following rules:

*Generation phase:*

g1. $[\ d_i \to e_i a u]_2$, $0 \le i \le n-2$,
$[\ d_{n-1} \to e_{n-1} a v]_2$.

g2. $[\ a]_2 \to [\ O_1]_2 [\ O_2]_2$,
$[\ a \to tf]_2$,
$[\ e_i \to \bar{e}_i \bar{e}'_i]_2$, $0 \le i \le n-1$,
$[\ u \to \bar{u}\bar{u}']_2$,
$[\ v \to \bar{v}\bar{v}']_2$.

g3. $[\ t]_2 \to [\ ]_3 \lambda$,
$[\ f]_2 \to [\ ]_4 \lambda$,

g4. $[\ \bar{e}_i \to d_{i+1}]_3$, $0 \le i \le n-2$,
$[\ \bar{e}'_i \to d_{i+1}]_4$, $0 \le i \le n-2$,
$[\ \bar{e}_{n-1} \to d_0 u]_3$,
$[\ \bar{e}'_{n-1} \to d_0 u]_4$.

g5. $[\ \bar{u}]_3 \to [\ ]_2 \lambda$,
$[\ \bar{u}']_4 \to [\ ]_2 \lambda$,
$[\ \bar{v}]_3 \to [\ ]_5 \lambda$,
$[\ \bar{v}']_4 \to [\ ]_5 \lambda$.

In $4n$ steps, $2^n$ membranes are created, corresponding to the truth assignments of the variables $x_1, \ldots, x_n$. During this process, object $d_i$ inside the membrane with label 3 corresponds to the *true* value of variable $x_{i+1}$, and object $d_i$ inside the membrane with label 4, corresponds to the *false* value of variable $x_{i+1}$. Object $a$ is used to choose the truth assignment of variables, and objects $u$ and $u'$ are used to change the membrane label back to 2. At step $4n$, labels 3 and 4 of internal membranes are changed to 5 by objects $v$ and $v'$. The membranes with label 5 represent all possible truth assignments of the variables in $\varphi$. Every such membrane will contain $d_0$, $u$, and the objects represent that the clauses are satisfied.

g6. $[\ x_{i,j} \to x_{i,j} x'_{i,j}]_2$, $1 \le i \le m$, $1 \le j \le n$,
$[\ \bar{x}_{i,j} \to \bar{x}_{i,j} \bar{x}'_{i,j}]_2$, $1 \le i \le m$, $1 \le j \le n$.

g7. $[\ x_{i,1} \to c_{i,0}]_3$, $1 \le i \le m$,
$[\ \bar{x}_{i,1} \to \lambda]_3$, $1 \le i \le m$.

g8. $[\ \bar{x}'_{i,1} \to c_{i,0}]_4$, $1 \le i \le m$,
$[\ x'_{i,1} \to \lambda]_4$, $1 \le i \le m$.

g9. $[\ x_{i,j} \to x_{i,j-1}]_3$, $1 \le i \le m$, $2 \le j \le n$,
$[\ \bar{x}_{i,j} \to \bar{x}_{i,j-1}]_3$, $1 \le i \le m$, $2 \le j \le n$,
$[\ x'_{i,j} \to x_{i,j-1}]_4$, $1 \le i \le m$, $2 \le j \le n$,
$[\ \bar{x}'_{i,j} \to \bar{x}_{i,j-1}]_4$, $1 \le i \le m$, $2 \le j \le n$.

The label of the created membranes is 2 and then changes to 3 or 4 at steps $4i+3$, $0 \le i < n$. Every object $x_{i,j}$ of the input evolves to $x_{i,1}$ or $x'_{i,1}$

in $4(i-1)$ steps. Then, it evolves to $c_{i,0}$ in membranes where *true* value was chosen for $x_j$ (recall that $x_{i,j} = \mathit{true}$ satisfies clause $C_i$) and is erased in membranes where *false* value was chosen for $x_j$. Similarly, $\bar{x}_{i,j}$ changes to $c_{i,0}$ if $x_j = \mathit{false}$, and is erased if $x_j = \mathit{true}$.

g10. $[\, c_{i,0} \to c_{i,1}]_2$, $1 \leq i \leq m$.

g11. $[\, c_{i,1} \to c_{i,2} c'_{i,2}]_2$, $1 \leq i \leq m$.

g12. $[\, c_{i,2} \to c_{i,0}]_3$, $1 \leq i \leq m$,
$[\, c'_{i,2} \to c_{i,0}]_4$, $1 \leq i \leq m$.

Rules of types g10, g11, and g12 represent that if the clause $C_i$ is satisfied in an internal membrane, then, it is also satisfied in the new created membranes.

*Checking phase:*

c1. $[\, c_{i,0} \to c_{i-1,0}]_5$, $1 \leq i \leq m$.

c2. $[\, u]_5 \to [\,]_6 \lambda$.

c3. $[\, c_{0,0}]_6 \to [\,]_5 \lambda$.

c4. $[\, d_i \to d_{i+1} u]_6$, $0 \leq i < m-1$.

c5. $[\, d_{m-1} \to d_m]_6$.

By expelling object $u$, the label of the membrane changes from 5 to 6. At the same time, the subscripts of all objects $c_{j,0}$ are decremented by one. A membrane with label 6 where object $c_{0,0}$ appears will change the label back to 5. In addition, the subscript of $d_i$ is incremented by one and $u$ is reproduced (except for $i = m-1$).

If at the beginning of the checking phase $c_{1,0}, \ldots, c_{i,0}$ are present ($1 \leq i < m$), but $c_{i+1,0}$ is absent, then, after $2i+1$ steps, rule c3 will no longer be applicable and the membrane will have label 6, no object $c_{0,0}$, and will never change the label again. After $m+i+1$ steps, the membrane will stop evolving from the beginning of the checking phase. If all objects $c_{i,0}$, $1 \leq i \leq m$, are present at the beginning of the checking phase, then after $2m$ steps they will all be erased, $d_0$ will evolve into $d_m$ and the membrane label will be 5.

*Output phase:*

o1. $[\, d_m]_5 \to [\,]_5 \mathtt{YES}$.

o2. $[\, \mathtt{YES}]_1 \to [\,]_7 \mathtt{YES}$.

o3. $[\, d_i \to d_{i+1}]_1$, $0 \leq i \leq 4n+2m+1$.

o4. $[\, d_{4n+2m+2}]_1 \to [\,]_1 \mathtt{NO}$.

At step $4n + 2m + 1$, every membrane corresponding to a solution of $\varphi$ expels YES in the skin region, and in the next step one copy of YES (if any) is ejected into the environment, changing the label of the skin from 1 to 7. If $\varphi$ has no solutions, then after step $4n + 2m + 2$ the skin membrane remains with label 1 and then rule o4 is applied, ejecting the object NO into the environment.

The P system $\Pi(\langle n, m\rangle)$ can be constructed by a deterministic Turing machine working in polynomial time because it uses at most $4nm + 7n + 5m + 18$ number of objects, 2 initial membranes with at most $nm + 2$ objects and $6mn + 9n + 12m + 17$ rules. The length of any rule is bounded by $4nm + 7n + 5m + 18$. □

We give the following two theorems without proofs; they improve the corresponding theorems in [4], and are not difficult to be proved by using the generation phase in the proof of Theorem 3.4 and in the proofs of Theorems 2 and 3 in [4].

**Theorem 3.5** *A uniform family of P systems with rules of types* ev, out, mer, sep′ *can solve* SAT *in linear time in a confluent way.*

**Theorem 3.6** *A uniform family of P systems with rules of types* ev, mer, sep′, rel *can solve* SAT *in linear time in a confluent way.*

## 3.2 Minimal Parallelism

An interesting problem concerning the application of developmental rules under different constraints is *minimal parallelism*, introduced and investigated in [24], which relaxes the condition of using the rules in a maximally parallel way. More precisely, the rules are used in the *non-deterministic minimally parallel* manner: in each step, from each set of rules $R_i$ (associated with a membrane $i$ of a P system) we use *at least one* rule (without specifying how many) provided that this is possible. The rules to be used, as well as the objects to which they are applied, are non-deterministically chosen. As usual for P systems with active membranes, each membrane and each object can be involved in only one rule, and the choice of rules to use and of objects and membranes to evolve is done in a non-deterministic way.

The set of numbers generated by a system $\Pi$ (see Section 3.1) working in the minimally parallel way is denoted by $N^{min}_{gen}(\Pi)$ and the family of such sets, generated by systems having initially at most $n_1$ membranes and using during the computation configurations with at most $n_2$ membranes is denoted by $N^{min}_{gen}OP_{n_1,n_2}(\textit{types-of-rules})$, with the subscript *"min"* indicating the "minimal parallelism" used in computations, and *types-of-rules* indicating the allowed types of rules. The set of all numbers accepted by a P system $\Pi$ working in the minimally parallel way is denoted by $N^{min}_{acc}(\Pi)$,

and the family of such sets is denoted by $N_{acc}^{min}OP_{n_1,n_2}(types\text{-}of\text{-}rules)$, with the obvious meaning of the used parameters. When the number of membranes does not increase during the computation we use only the subscript $n_1$, denoting the number of membranes initially present in the used systems.

In what follows we give several accepting and generative universality results, as well as efficiency results for polarizationless P systems working in the *minimally parallel mode.*

### 3.2.1 Computational Completeness Results

**The Accepting Case** When we remove polarizations from active membranes, additional features are in general necessary in order to reach the universality. There are several such features, for instance, cooperative rewriting rules, changing membrane labels, using promoter/inhibitor objects, priorities among rules, etc. Some of these tools will be also used in what follows.

In the proofs we will simulate register machines. In all constructions, with each register $r$ of a register machine we associate a membrane with label $r$, and the number stored in register $r$ is represented by the number of copies of object $a$ present in membrane $r$.

**Theorem 3.7** $N_{acc}^{min}OP_n(cat_1, pro, \mathsf{in}, \mathsf{out}) = NRE, n \geq 7.$

*Proof.* Let us consider a deterministic register machine $M = (3, B, l_0, l_h, I)$ accepting an arbitrary set $N(M) \in NRE$. We construct the P system

$$\begin{aligned}
\Pi &= (O, C, H, \mu, (w_h)_{h\in H}, (R_h)_{h\in H}, 1) \text{ with} \\
O &= \{a, c\} \cup \{l, \bar{l}, l', l'', l''', l^{iv}, l^{v}, l^{vi}, l^{vii} \mid l \in B\}, \\
C &= \{c\}, \\
H &= \{0, 1, 1', 2, 2', 3, 3'\}, \\
\mu &= [\,[\,[\,]_{1'}]_1[\,[\,]_{2'}]_2[\,[\,]_{3'}]_3]_0, \\
w_0 &= l_0, w_1 = w_2 = w_3 = c, w_1' = w_2' = w_3' = \lambda,
\end{aligned}$$

and with the following rules in $R_h, h \in H$. (Remember that the number to be analyzed is introduced in region 1 in the form $a^n$.)

An instruction $l_1 : (\mathsf{add}(r), l_2)$ is simulated by means of the following rules:

| *step* | $R_0$ | $R_r$ |
|---|---|---|
| 1. | — | $l_1[\,]_r \to [\,l_1]_r$ |
| 2. | — | $[\,l_1 \to l_2' a]_r$ |
| 3. | — | $[\,l_2']_r \to [\,]_r l_2'$ |
| 4. | $[\,l_2' \to l_2]_0$ | — |

The label-object $l_1$ enters the correct membrane $r$, produces one further copy of $a$ and the label $l_2$, primed, inside membrane $r$, then the label $l_2'$ exits to the skin region, loses its prime, and the process can be iterated.

For the simulation of a SUB instruction $l_1 : (\mathsf{sub}(r), l_2, l_3)$ we use the next rules:

| *step* | $R_0$ | $R_r$ | $R_{r'}$ |
|---|---|---|---|
| 1. | $[l_1 \to l_1' l_1'']_0$ | — | — |
| 2. | $[l_1' \to l_1''']_0$ | $l_1''[\,]_r \to [l_1'']_r$ | — |
| 3. | $[l_1''' \to l_1^{iv}]_0$ | $[ca \to ca'\|_{l_1''}]_r$ | $l_1''[\,]_{r'} \to [l_1'']_{r'}$ |
| 4. | — | $l_1^{iv}[\,]_r \to [l_1^{iv}]_r$ | $[l_1'' \to l_1^{v}]_{r'}$ |
| 5. | — | $[l_1^{iv} \to \bar{l}_2\|_{a'}]_r$ | $[l_1^{v} \to l_1^{vi}]_{r'}$ |
| 6. | — | $[a' \to \lambda\|_{\bar{l}_2}]_r$ | $[l_1^{vi}]_{r'} \to l_1^{vi}[\,]_{r'}$ |
| 7. | — | $[\bar{l}_2 \to l_2\|_{l_1^{vi}}]_r$ or $[l_1^{iv} \to l_3\|_{l_1^{vi}}]_r$ | $l_1^{vi}[\,]_{r'} \to [l_1^{vii}]_{r'}$ |
| 8. | — | $[l_2]_r \to l_2[\,]_r$ or $[l_3]_r \to l_3[\,]_r$ | $[l_1^{vii} \to \lambda]_{r'}$ |

We start the computation by producing a couple of objects $l_1'$ and $l_1''$ in the skin region by rule 1. Then $l_1'$ evolves to $l_1'''$ in the skin region, while object $l_1''$ enters into membrane $r$. In the third step, while $l_1'''$ changes to the next primed version $l_1^{iv}$ in the skin region, object $l_1''$ enters into the inner membrane $r'$ – this happens in both cases irrespective whether object $a$ exists or not in membrane $r$. If an object $a$ was present, it evolves to $a'$ in the presence of promoter object $l_1''$ by means of the catalytic evolution rule $[\, ca \to ca'|_{l_1''}]_r$ and the promoter leaves the membrane $r$. The catalyst $c$ is used to prevent more objects $a$ to evolve in the same step. At the fourth step, object $l_1''$ evolves to $l_1^{v}$ in membrane $r'$ and object $l_1^{iv}$ enters into membrane $r$, respectively. Since no object remains in the skin region, this region will stay idle until the end of the computation. In the step five, object $l_1^{v}$ evolves to $l_1^{vi}$, which will be sent out of the membrane $r'$ in the following step; in membrane $r$, if a promoter object $a'$ is present, then object $l_1^{iv}$ produces the object $\bar{l}_2$. Otherwise, there is no rule to be applied here in this or the next step. In step 6, object $l_1^{vi}$ arrives into membrane $r$, where the object $\bar{l}_2$ promotes the deletion rule $[a' \to \lambda|_{\bar{l}_2}]_r$ and removes the object $a'$ previously produced. In membrane $r$, object $l_1^{vi}$ promotes either object $\bar{l}_2$ or $l_1^{iv}$, to introduce the corresponding label-object $l_2$ or $l_3$, respectively. The correct label-object $l_2$ or $l_3$ is moved to skin region at the 8th step of the computation. The object $l_1^{vi}$ enters membrane $r'$ and is removed.

Note that in the simulation of instructions add and sub of $M$ in each computation step at most one rule from each set $R_h$ has been used, hence the system works both in the minimally and in the maximally parallel mode. The starting configuration of the system is restored after each simulation, hence another instruction can be simulated. If the computation in $M$ halts, hence $l_h$ is reached, this means that the halting label-object $l_h$ is introduced

in the skin region, and also the computation in $\Pi$ halts. Consequently, $N(M) = N_{acc}^{min}(\Pi)$, and this concludes the proof. □

In the next theorem we use context-free evolution rules, move-in and move-out rules, as well as the possibility of changing membrane labels.

**Theorem 3.8** $N_{acc}^{min}OP_n(\mathsf{ev}, \mathsf{in}', \mathsf{out}') = NRE, n \geq 4.$

*Proof.* Let us consider a deterministic register machine $M = (3, B, l_0, l_h, I)$ accepting an arbitrary set $N(M) \in NRE$. We construct the P system

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, w_1, w_2, w_3, R_0, R_1, R_2, R_3, 1) \text{ with} \\
O &= \{a\} \cup \{l, l^i, l^{ii}, l^{iii}, l^{iv}, l^v \mid l \in B\}, \\
H &= \{0\} \cup \{r, r^0, r', r'', r''' \mid r = 1, 2, 3\}, \\
\mu &= [\ [\ ]_1[\ ]_2[\ ]_3]_0, \\
w_0 &= l_0, w_1 = w_2 = w_3 = \lambda,
\end{aligned}$$

and with the following sets of rules.

In the simulation of $M$, membrane labels associated with register $r$ are changed during the computation.

An instruction $l_1 : (\mathsf{add}(r), l_2)$ can be simulated in the same way as in the proof of Theorem 3.7, hence we skip the details.

The simulation of an instruction $l_1 : (\mathsf{sub}(r), l_2, l_3)$ uses the following rules:

| *step* | $R_0$ | $R_r$ |
|---|---|---|
| 1. | $[\ l_1 \to l_1^i l_1^{ii}]_0$ | — |
| 2. | $[\ l_1^{ii} \to l_1^{iii}]_0$ | $l_1^i[\ ]_r \to [\ l_1^i]_{r^0}$ |
| 3. | $[\ l_1^{iii} \to l_1^{iv}]_0$ | $[\ a]_{r^0} \to [\ ]_{r'}a$ |
| 4. | $[\ a \to \lambda]_0$ | $l_1^{iv}[\ ]_{r^0} \to [\ l_3]_{r''}$ or $l_1^{iv}[\ ]_{r'} \to [\ l_2]_{r''}$ |
| 5. | — | $[\ l_1^i]_{r''} \to [\ ]_{r'''}l_1^v$ |
| 6. | $[\ l_1^v \to \lambda]_0$ | $[\ l_2]_{r'''} \to [\ ]_r l_2$ or $[\ l_3]_{r'''} \to [\ ]_r l_3$ |

In the first step of the computation, objects $l_1^i$ and $l_1^{ii}$ are produced by object $l_1$ in the skin region (there are no applicable rules associated with other membranes). In the next step, object $l_1^{ii}$ evolves to $l_1^{iii}$ in the skin region, and in the meantime object $l_1^i$ enters into membrane $r$ changing its label to $r^0$. At the third step, if any copy of object $a$ exists in membrane $r^0$, it leaves the membrane, changing its label to $r'$; this makes sure that no other $a$ can leave the membrane. Simultaneously, object $l_1^{iv}$ is introduced in the skin region. This object enters into membrane with label $r^0$ or $r'$ present in the skin region, introduces the correct object $l_2$ or $l_3$ inside it, and changes the label of the membrane to $r''$. In step 5, object $l_1^i$, which has waited here since step 2, introduces object $l_1^v$ into the skin region, changing

the label of membrane $r''$ to $r'''$. In the meantime no rule is applicable in the skin region. Finally, the correct label-object $l_2$ or $l_3$ is moved in the skin region, and the membrane label is changed from $r'''$ to the initial $r$, in the sixth step. Object $l_1^v$ is erased from the skin region. Thus, the simulation is completed and the system is set up for the next iteration. □

In the next theorem we use the rather strong tool of cooperative evolution rules (with radius at most 2), but the membrane labels will not be changed during the computation.

**Theorem 3.9** $N_{acc}^{min}OP_n(\mathsf{cev}, \mathsf{in}, \mathsf{out}) = NRE, n \geq 4.$

*Proof.* Let us consider a deterministic register machine $M = (3, B, l_0, l_h, I)$ accepting an arbitrary set $N(M) \in NRE$. We construct the P system

$$\begin{aligned}
\Pi &= (O, H, \mu, l_0, \lambda, \lambda, \lambda, R_0, R_1, R_2, R_3, 1) \text{ with} \\
O &= \{a\} \cup \{l^i, l^{ii}, l^{iii}, l^{iv}, l^v \mid l \in B\}, \\
H &= \{0, 1, 2, 3\}, \\
\mu &= [\,[\,]_1[\,]_2[\,]_3]_0,
\end{aligned}$$

and with the following rules.

We again skip the simulation of an instruction $l_1 : (\mathsf{add}(r), l_2)$ since it can be done in the same way as in Theorem 3.7.

The simulation of an instruction $l_1 : (\mathsf{sub}(r), l_2, l_3)$ uses the following rules:

| *step* | $R_0$ | $R_r$ |
|---|---|---|
| 1. | $[\, l_1 \to l_1^i l_1^{ii}]_0$ | — |
| 2. | $[\, l_1^{ii} \to l_1^{iv}]_0$ | $l_1^i[\,]_r \to [\, l_1^i]_r$ |
| 3. | $[\, l_1^{iv} \to l_1^v]_0$ | $[\, l_1^i a \to l_1^{iii}]_r$ |
| 4. | — | $l_1^v[\,]_r \to [\, l_1^v]_r$ |
| 5. | — | $[\, l_1^v l_1^{iii} \to l_2]_r$ or $[\, l_1^v l_1^i \to l_3]_r$ |
| 6. | — | $[\, l_2]_r \to [\,]_r l_2$ or $[\, l_3]_r \to [\,]_r l_3$ |

Again we start by producing two different objects $l_1^i$ and $l_1^{ii}$ in the skin region, while no rule is applied in membrane $r$. In the second step, object $l_1^i$ enters in the membrane $r$, and object $l_1^{iv}$ is produced in the skin region. After that, object $l_1^{iv}$ evolves to $l_1^v$ in the skin region. If there is an object $a$ present in membrane $r$, then it interacts with object $l_1^i$ and introduces object $l_1^{iii}$ reducing the number of $a$s, otherwise object $l_1^i$ remains here. Object $l_1^v$ enters membrane $r$ at step 4 and reacts either with object $l_1^{iii}$ or with $l_1^i$; the corresponding object $l_2$ or $l_3$ is introduced in step 5. Thus, the correct label-object $l_2$ or $l_3$ is introduced into skin membrane at step 6. In steps 4 – 6, the skin membrane is idle. The simulation of instructions from $I$ can

be iterated. Obviously, the system works in both minimally and maximally parallel mode. □

**The Generative Case** We consider now P systems working in the generative mode.

The next universality result is based on the simulation of a matrix grammar. Catalytic evolution rules, the dual pair of operations of membrane merging and separation, also changing membrane labels, are used in the proof.

**Theorem 3.10** $N_{gen}^{min}OP_{2,5}(cat_1, \mathsf{sep}', \mathsf{mer}') = NRE.$

*Proof.* Let us consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the *binary normal form*, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$, $T = \{a\}$, and with the matrices of the forms as mentioned in Subsection 2.1.3.

Assume that all matrices are injectively labeled with elements of a set $B$.

We construct the P system of degree 2

$$\begin{aligned}
\Pi &= (O, C, H, \mu, w_0, w_s, (R_h)_{h\in H}, 0) \text{ with} \\
O &= N_1 \cup \{X' \mid X \in N_1\} \cup N_2 \cup \{c, c_1, c_2, a, f, \#\}, \\
C &= \{c\}, \\
H &= \{m, m_1, m_2, m_1', m_1'', m_2', m_2'', m_2''' \mid m \in B\} \cup \{0, 0', 0'', s\}, \\
\mu &= [\,[\,]_0]_s, \\
w_0 &= XAcc_1c_2, w_s = \lambda,
\end{aligned}$$

and the sets $R_h$ containing the rules below.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1$, $Y \in N_1$, and $A \in N_2, x \in (N_2 \cup T)^*$, $|x| \leq 2$, is done in four steps, using the following rules (the matrix with label $m$ is encoded in the labels of the membranes created by the separation operation; we start from a configuration of the form $[\,[\,Xwcc_1c_2]_0]_s$, for some $w \in (N_2 \cup T)^*$):

| step | $(X \to Y,$ | $A \to x)$ |
|---|---|---|
| 1. | $[\,O]_0 \to [\,\{X\}]_{m_1}[\,O - \{X\}]_{m_2}$ | — |
| 2. | $[\,X \to X']_{m_1}$ | $[\,O]_{m_2} \to [\,\{c, A\}]_{m_2'}[\,O - \{c, A\}]_{m_2''}$, or $[\,c_1 \to \#]_{m_2}$, $[\,\# \to \#]_{m_2}$ |
| 3. | $[\,X' \to Y]_{m_1}$ | $[\,cA \to cx]_{m_2'}$ |
| | $[\,]_{m_1}[\,]_{m_2'} \to [\,]_{m_1'}$ | |
| 4. | $[\,]_{m_1'}[\,]_{m_2''} \to [\,]_0$ | — |

The initial multiset from membrane 0 is $XAcc_1c_2$. We start the simulation of matrix $m$ by separating membrane 0 under the control of object $X$.

After separation, object $X$ takes place in a new membrane $m_1$, and other objects (including $c, c_1, c_2$) are placed in a membrane $m_2$. In the second step, object $X$ evolves to $X'$ while objects $A, c$ makes membrane $m_2$ separate. At the third step, $X'$ introduces $Y$ in membrane $m_1$, one copy of object $A$ evolves to $x$ in the new membrane $m_2'$ (thus, the rules of matrix $m$ have been simulated), and membranes with labels $m_1$ and $m_2'$ are merged into a new membrane labeled $m_1'$. At the fourth step, objects $Y, x, c, c_1, c_2$ returns to membrane 0, obtained by merging membranes $m_1'$ and $m_2''$ (the rule $[\ cA \rightarrow cx]_{m_2'}$ cannot be used again in step 4). If the object $X$ is present, hence membrane 0 is separated, but object $A$ not, then in step 2 one introduces the trap object # by the rule $[\ c_1 \rightarrow \#]_{m_2}$, which must be used, because $[\ O]_{m_2} \rightarrow [\ \{c, A\}]_{m_2'}[\ O - \{c, A\}]_{m_2''}$ cannot be used.

Thus, the matrix $m$ is correctly simulated, and the system can pass to the simulation of another matrix.

The simulation of a matrix with appearance checking $m : (X \rightarrow Y, A \rightarrow \#)$, with $X, Y \in N_1$, and $A \in N_2$, is done in four steps using the following rules:

| step | $(X \rightarrow Y,$ | $A \rightarrow \#)$ |
|---|---|---|
| 1. | $[\ O]_0 \rightarrow [\ \{X, c_1\}]_{m_1}[\ O - \{X, c_1\}]_{m_2}$ | |
| 2. | $[\ O]_{m_1} \rightarrow [\ \{X\}]_{m_1'}[\ O - \{X\}]_{m_1''}$ | $[\ O]_{m_2} \rightarrow [\ \{A\}]_{m_2'}[\ O - \{A\}]_{m_2''}$ |
| 3. | $[\ X \rightarrow Y]_{m_1'}$ | $[\ A \rightarrow \#]_{m_2'}$ |
| | $[\ ]_{m_1''}[\ ]_{m_2} \rightarrow [\ ]_{m_2'''}$ | $[\ \# \rightarrow \#]_{m_2'}$ |
| 4. | $[\ ]_{m_1'}[\ ]_{m_2'''} \rightarrow [\ ]_0$ | |

The computation starts with objects $X, c_1$ making membrane 0 to separate. At the second step, if membrane $m_2$ includes an object $A$, then it will separate into membranes $m_2'$ including $A$ and $m_2''$ including the auxiliary object $c_2$. If $A$ existed, then the computation never stop. At the same time, objects $X$ and $c_1$ take place in the membranes $m_1'$ and $m_1''$, respectively. In the third step, $X$ evolves to $Y$, and if membrane $m_2$ is still present, then membrane $m_1''$ merges with it, creating a new membrane $m_2'''$. Finally, membranes $m_1'$ and $m_2'''$ are merged into membrane 0 including all correct objects and the system returns to a configuration as the starting one.

The simulation of a matrix $m : (X \rightarrow \lambda, A \rightarrow x)$, with $X \in N_1, A \in N_2$, and $x \in T^*, |x| \leq 2$, is done in six steps. We omit here the detailed explanation for the first 4 steps, because they are the same as in the simulation of matrix $m : (X \rightarrow Y, A \rightarrow x)$. At step 5, membrane $0'$ is separated into membranes with labels 0 and $0''$. The former one includes the terminal objects and the special object $f$. Object $f$ evolves to $\lambda$ at step 6. If there are objects $Z$ from $N_2$ in membrane $0''$, then the rule $[\ Z \rightarrow \#]_{m_1'}$ is applied and the computation will never halt. Thus, the simulation is correctly completed using the following rules:

| step | $(X \to f,$ | $A \to x)$ |
|---|---|---|
| 1. | $[\ O]_0 \to [\ \{X\}]_{m_1} [\ O - \{X\}]_{m_2}$ | — |
| 2. | $[\ X \to X']_{m_1}$ | $[\ O]_{m_2} \to [\ \{c, A\}]_{m_2'} [\ O - \{c, A\}]_{m_2''}$, or $[\ c_1 \to \#]_{m_2}$, $[\ \# \to \#]_{m_2}$ |
| 3. | $[\ X' \to f]_{m_1}$ $[\ ]_{m_1} [\ ]_{m_2'} \to [\ ]_{m_1'}$ | $[\ cA \to cx]_{m_2'}$ |
| 4. | $[\ ]_{m_1'} [\ ]_{m_2''} \to [\ ]_{0'}$ | — |
| 5. | $[\ O]_{0'} \to [\ U]_0 [\ O - U]_{0''}$ for $U = \{f\} \cup T$ | — |
| 6. | $[\ f \to \lambda]_0$ | $[\ Z \to \#]_{0''}, Z \in N_2$ $[\ \# \to \#]_{0''}$ |

□

### 3.2.2 Computational Complexity Results

We present here both *uniform* and *semi-uniform* linear time solutions to SAT based on polarizationless P systems working in the minimally parallel mode. Three results are given, with the following features (types of rules, label changing, type of construction):

| *no.* | *evolution* | *division* | *in/out* | *label change* | *construction* |
|---|---|---|---|---|---|
| 1 | cooper. | elementary | move out | no | semi-uniform |
| 2 | non-coop. | non-element. | move in, out | yes | uniform |
| 3 | cooper. | non-element. | move in, out | no | uniform |

**Theorem 3.11** *P systems working in the minimally parallel mode with rules of types* cev, ediv, out *and constructed in a semi-uniform manner can solve* SAT *in linear time with respect to the number of variables and the number of clauses.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form, $\varphi = C_1 \wedge \cdots \wedge C_m$ with $Var(\varphi) = \{x_1, \ldots, x_n\}$, such that each clause $C_i, 1 \leq i \leq m$, is of the from $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}, k_i \geq 1$, where $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}$. For each $k = 1, 2, \ldots, n$, let us denote

$$t(x_k) = \{e_i \mid \text{there is } 1 \leq j \leq k_i \text{ such that } y_{i,j} = x_k\},$$
$$f(x_k) = \{e_i \mid \text{there is } 1 \leq j \leq k_i \text{ such that } y_{i,j} = \neg x_k\}.$$

These are the sets of clauses which assume the value *true* when $x_i$ is *true*, and *false* when $x_i$ is *false*, respectively.

We construct the P system

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, w_s, R_0, R_1, R_2, R_s), \text{ with} \\
O &= \{a_i, b_i, c_i \mid 1 \leq i \leq n\} \cup \{e_i, e_i' \mid 1 \leq i \leq m\} \\
&\cup \{d_i \mid 0 \leq i \leq 2n+m\} \cup \{e, h, \texttt{YES}, \texttt{NO}\}, \\
H &= \{0, 1, 2, s\}, \\
\mu &= [\,[\,[\,]_0[\,]_2]_1]_s, \\
w_0 &= a_1, w_1 = h, w_2 = d_0, w_s = \lambda,
\end{aligned}$$

The rules of P systems are described below.

The truth assignments are produced by using the rules from the following table:

$$R_0 : \begin{cases} 1.\ [\,a_i]_0 \to [\,b_i]_0[\,c_i]_0, 1 \leq i \leq n, \\ 2.\ [\,e_1e_2 \to e_2']_0, [\,e_i'e_{i+1} \to e_{i+1}']_0, 2 \leq i \leq m-1, \\ 3.\ [\,b_i \to t(x_i)a_{i+1}]_0, [\,c_i \to f(x_i)a_{i+1}]_0, \\ 4.\ [\,b_n \to t(x_n)]_0, [\,c_n \to f(x_n)]_0, 1 \leq i \leq n-1. \end{cases}$$

$$R_2 : 9.\ [\,d_i \to d_{i+1}]_2, 0 \leq i \leq 2n+m.$$

In odd steps, we divide the elementary membrane 0 by using rule 1 (with $b_i, c_i$ corresponding to the truth values *true*, *false*, respectively, for variable $x_i$); in even steps we introduce the clauses $t(x_i)$ and $f(x_i)$ satisfied by $x_i$, and $\neg x_i$, respectively, by using rules 3 and 4. Simultaneously to a division step of membrane 0, when the clauses $C_i$ and $C_{i+1}$ are satisfied by the previously expanded variables, the corresponding objects $e_i'$ and $e_{i+1}$ may interact with each other, and that produces a primed object $e_{i+1}'$ by rule 2. In a single step, if rule 2 can be applied, then it must be applied at least once. For instance, if we have a multiset of objects $\{e_1, e_2, e_2', e_3\}$, then we can have either a multiset of objects $\{e_3', e_2'\}$ or $\{e_3', e_1, e_2\}$ or $\{e_2', e_3, e_2'\}$ in the next step. When we divide a membrane, all inner objects are replicated and presented in the new membranes. The division process lasts $2n-1$ steps. At the end of this phase, all $2^n$ truth assignments for the variables $x_1, \ldots, x_n$ are generated in $2^n$ membranes with label 0.

At the same time, the counter object $d_i, 1 \leq i \leq 2n+m$, is counting the computation steps in membrane 2 by means of rule 9.

In the next phase we check whether the formula is satisfiable, by means of the next rules:

$$\begin{aligned}
R_0 &: 5.\ [\,e_m']_0 \to [\,]_0e_m' \\
R_1 &: \begin{cases} 6.\ [\,he_m' \to e]_1, \\ 7.\ [\,ed_{2n+m} \to \texttt{YES}]_1,\ [\,\texttt{hd}_{\texttt{2n+m}} \to \texttt{NO}]_1, \\ 8.\ [\,\texttt{YES}]_1 \to [\,]_1\texttt{YES},\ [\,\texttt{NO}]_1 \to [\,]_1\texttt{NO} \end{cases} \\
R_2 &: 10.\ [\,d_{2n+m-1}]_2 \to [\,]_2d_{2n+m} \\
R_s &: 11.\ [\,\texttt{YES}]_s \to [\,]_s\texttt{YES},\ [\,\texttt{NO}]_s \to [\,]_s\texttt{NO}.
\end{aligned}$$

If the formula is satisfiable, copies of object $e'_m$ appear in some membranes with label 0 in steps between $2n-1$ and $2n+m-2$ (by rule 5). When object $e'_m$ is produced, it can leave membrane 0. This is done in the first $2n+m-1$ steps of the computation. Then, object $e'_m$ (non-deterministically chosen if there are several) interacts with object $h$ (which initially stayed at membrane 1) and produces object $e$. Object $e$ will wait here for a while. At step $2n+m$, the counter object $d_{2n+m}$ enters into membrane 1 by using rule 10, and interacts here with the object $e$ or $h$ which is available. An object YES or NO is, thus, introduced (rule 7). The correct answer will be sent to the environment in the next two steps by rules 8, 11. The computation stops in at most $2n+m+3$ steps.

Note that the system has a polynomial size in $n$ and $m$: it uses $5n+3m+5$ objects and $5n+m+9$ rules, all rules being of a length linearly bounded with respect to $m$. □

**Theorem 3.12** *P systems constructed in a uniform manner and working in the minimally parallel mode using rules of types* ev, ndiv, in, out′ *can solve* SAT *in linear time.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form, $\varphi = C_1 \wedge \cdots \wedge C_m$ with $Var(\varphi) = \{x_1, \ldots, x_n\}$, such that each clause $C_i, 1 \leq i \leq m$, is of the form $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}, k_i \geq 1$, where $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}$.

The instance $\varphi$ is encoded as a set whose objects $x_{i,j}$ represent the variable $x_j$ appearing in the clause $C_i$ without negation, and objects $x'_{i,j}$ represent the variable $x_j$ appearing in the clause $C_i$ with negation. The input multiset will be

$$\begin{aligned} cod(w) &= \{x_{i,j} \mid x_j \in \{y_{i,j} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{x'_{i,j} \mid \neg x_j \in \{y_{i,j} \mid 1 \leq k \leq l_i\}, 1 \leq i \leq m, 1 \leq j \leq n\}. \end{aligned}$$

For a given $(n,m) \in \mathbb{N}^2$, we construct a recognizing P system $(\Pi(\langle n,m\rangle), \Sigma(\langle n,m\rangle), 0)$ with:

$$\begin{aligned} \Pi(\langle n,m\rangle) &= (O(\langle n,m\rangle), H, \mu, (w_h)_{h\in H}, (R_h)_{h\in H}), \text{where} \\ O(\langle n,m\rangle) &= \{x_{i,j}, x'_{i,j}, t_{j,i}, f_{j,i} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{b_i, c_i, a'_i, a''_i, a'''_i \mid 1 \leq i \leq n\} \cup \{d, \mathtt{YES}, \mathtt{NO}\} \\ &\cup \{l'_i \mid 0 \leq i \leq n\} \cup \{l_i, e_i, \mid 1 \leq i \leq m\} \\ &\cup \{a_i \mid 0 \leq i \leq n+2\} \cup \{d_i \mid 0 \leq i \leq 6n+m+8\}, \\ \Sigma(\langle n,m\rangle) &= \{x_{i,j}, x'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\}, \\ \mu &= [\,[\,[\,[\,]_m \cdots [\,]_2 [\,]_1]_0]_a [\,]_d]_s, \\ w_a &= a_0, w_d = d_0, w_0 = l'_0, w_s = w_i = \lambda, 1 \leq i \leq m, \\ H &= \{s, a, b, d, y, n\} \cup \{i \mid 0 \leq i \leq m\} \cup \{i', i'' \mid 0 \leq i', i'' \leq n\}, \end{aligned}$$

The set $R$ of rules of P systems $\Pi(\langle n, m\rangle)$, to which we also give explanations about, are the following:

1. $[\ d_i \to d_{i+1}]_d,\ 0 \leq i \leq 6n + m + 8.$

Object $d_i$ counts the computation steps in membrane $d$.

*Initialization phase:*

2. $x_{i,j}[\ ]_i \to [\ x_{i,j}]_i,$
   $x'_{i,j}[\ ]_i \to [\ x'_{i,j}]_i,\ 1 \leq i \leq m, 1 \leq j \leq n.$

We re-encode the instance of the problem $\varphi$ into $m$ membranes in $n$ steps.

3. $[\ a_i \to a_{i+1}]_a, 0 \leq i \leq n + 2.$
4. $[\ l'_i \to l'_{i+1}]_0, 0 \leq i \leq n - 1.$

Simultaneously to the use of rule 2, evolution rules 3 and 4 are applied in membranes $a$ and 0, respectively.

5. $[\ l'_n \to l_1 l_2 \ldots l_m]_0.$
6. $l_i[\ ]_j \to [\ l_i]_j,\ 1 \leq i, j \leq m.$
7. $[\ l_i]_j \to [\ ]_{0'} d,\ 1 \leq i, j \leq m.$
8. $a_{n+2}[\ ]_0 \to [\ a_1]_0.$

At step $n + 1$, object $l'_n$ evolves to $l_1, l_2, \ldots, l_m$. Each object $l_j$ $(1 \leq j \leq m)$ enters into an is sent out membrane $j$ in two steps, by means rules 6 and 7, allowing the change of membrane label from $j$ to $0'$. Thus, the initialization phase has been completed in $n + 3$ steps.

*Checking phase:*

9. $[\ a_i]_0 \to [\ b_i]_0[\ c_i]_0, 1 \leq i \leq n.$
10. $[\ b_i \to t_{1,i} t_{2,i} \ldots t_{m,i} a'_i]_0,$
    $[\ c_i \to f_{1,i} f_{2,i} \ldots f_{m,i} a'_i]_0, 1 \leq i \leq n.$

We generate $2^n$ membranes with label 0 by using non-elementary membrane division rule 9. In each step, $b_i$ and $c_i$ correspond to the truth values *true* and *false*, respectively, for variable $x_i$. By rule 10, objects $b_i$ and $c_i$ evolve. Rules 9 and 10 are performed in 2 steps.

11. $t_{i,j}[\ ]_{(j-1)'} \to [\ t_{i,j}]_{(j-1)'}$, or
    $t_{i,j}[\ ]_{(j-1)''} \to [\ t_{i,j}]_{(j-1)''},$
    $f_{i,j}[\ ]_{(j-1)'} \to [\ f_{i,j}]_{(j-1)'}$, or
    $f_{i,j}[\ ]_{(j-1)''} \to [\ f_{i,j}]_{(j-1)''},\ 1 \leq i \leq m, 1 \leq j \leq n.$

12. $[\ a'_j \rightarrow a''_j]_0$.

The groups of objects $t_{1,j}, t_{2,j}, \ldots, t_{m,j}$ and $f_{1,j}, f_{2,j}, \ldots, f_{m,j}$, corresponding to variable $x_j$, are introduced into inner membranes with labels $(j-1)'$ or $(j-1)''$ by rule 11. At the same time, object $a'_j$ evolves to $a''_j$ by rule 12. These two rules are applied simultaneously.

13. $[\ t_{i,j}]_{(j-1)'} \rightarrow [\ ]_{j'}d$, $[\ t_{i,j}]_{(j-1)''} \rightarrow [\ ]_{j'}d$,
$[\ f_{i,j}]_{(j-1)'} \rightarrow [\ ]_{j''}d$, $[\ f_{i,j}]_{(j-1)''} \rightarrow [\ ]_{j''}d$,
$1 \leq i \leq m, 1 \leq j \leq n$.
14. $[\ a''_j \rightarrow a'''_j]_0$.

By using rule 13, objects $t_{i,j}$ and $f_{i,j}$ (corresponding to the truth value *true* and *false* for variable $x_j$ of a clause $C_i$) leave membranes with labels $(j-1)'$ or $(j-1)''$ and change the labels to $j'$ and $j''$, respectively. Object $a'''_j$ is produced in membrane 0 by rule 14. We will check which clauses are satisfied by truth values using rules from 15.

15. $[\ x_{i,j} \rightarrow e_i]_{j'}$,
$[\ x'_{i,j} \rightarrow e_i]_{j''}, 1 \leq i \leq m, 1 \leq j \leq n$.
16. $[\ a'''_j \rightarrow a_{j+1}]_0$.

It is important to bear mind that single and double primes of the labels $j'$ and $j''$ indicate *true* and *false* values respectively. By rule 15, object $x_{i,j}$ in membrane $j'$ and object $x'_{i,j}$ in membrane $j''$ evolve to $e_i$. More precisely, the object without negation has to evolve to object $e_l$, because the membrane label $k'$ has a single prime and this indicates the *true* value. The subscript of $e_l$ must be the same with the first subscript of the object $x_{l,k}$. Similarly, the object with negation has to evolve to $e_l$ because of $k''$ which is double-primed and indicates the value *false*.

In membranes labeled by 0, the objects $a'''_i$ lose their primes and increase the subscript $a_{i+1}$ (rule 16). In this way, the true values corresponding to the next variable $x_{i+1}$ can be assigned (rules 9 and 10). There are five steps between two consecutive membrane divisions. Thus, we need $5n$ division processes in order to obtain all possible $2^n$ truth assignments. Then, the checking process the satisfiability of all clauses ends at the $(6n+3)$th step of the computation. At $(6n+4)$th step, objects $a_{n+1}$ are removed from membrane 0 changing its label to 1, by means of rule 17.

17. $[\ a_{n+1}]_0 \rightarrow [\ ]_1 d$.

*Recognizing phase:*

After the $6n+4$ steps, there are $2^n$ membranes with label 1 and each of them contains $m$ membranes labeled by $n'$ or $n''$.

18. $[\ e_i]_{n'} \to [\ ]_n e_i$,
$[\ e_i]_{n''} \to [\ ]_n e_i, 1 \leq i \leq m.$

By rule 18, objects $e_i, 1 \leq i \leq m$, are introduced into membrane 1 changing the former membrane labels from $n'$ and $n''$ to $n$. The presence of all objects $e_1, e_2, \ldots, e_m$ in a membrane labeled by 1 means that all clauses $C_1, C_2, \ldots, C_m$ are satisfied.

19. $[\ e_i]_i \to [\ ]_{i+1} e_i, 1 \leq i \leq m.$

Objects $e_i, 1 \leq i \leq m$, leave membrane $i$ one by one increasing the label. In $m$ steps, we can get objects $e_m$ to appear in membrane $a$. One of them is non-deterministically chosen and expelled out of membrane $a$, which changes the label to $b$, by means of rule 20.

20. $[\ e_m]_a \to [\ ]_b e_m$.
21. $[\ e_m \to \mathtt{YES}]_{\mathtt{s}}$.
22. $[\ \mathtt{YES}]_{\mathtt{s}} \to [\ ]_{\mathtt{y}} \mathtt{YES}$.

If an object $e_m$ had appeared in the skin membrane, then it will evolve to YES; this object will be sent to the environment, changing the skin membrane label to $y$ by using rules 21, 22, in the $6n+m+8$th step of the computation.

23. $[\ d_{6n+m+7}]_d \to [\ ]_d d_{6n+m+8}$.
24. $[\ d_{6n+m+8}]_s \to [\ ]_n \mathtt{NO}$.

If the formula is not satisfiable, the object NO is sent to environment in the step $6n + m + 9$ by rule 24.

Thus, the problem is solved in a uniform manner, in a linear time, with the system working in both the minimally parallel and the maximally parallel modes.

The system $\Pi(\langle n, m\rangle)$ can be constructed by a deterministic Turing machine working in polynomial time, because it uses $4nm + 13n + 3m + 14$ objects, $m + 4$ initial membranes containing $mn + 3$ initial objects, and $12mn + 14n + 8m + 26$ rules. the length of any rule is bounded by ???. □

## An Example

Let us take the formula $\varphi = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2)$. Then the multiset encoding is $code(\varphi) = x_{11}x'_{21}x_{22}x'_{32}$, and the membrane structure with initial data is: $[\ [\ d_0]_d [\ a_0 [\ x_{11}x'_{2,1}x_{22}x'_{32}l'_0 [\ ]_3 [\ ]_2 [\ ]_1]_0]_a]_s$, see Figure 3.1.

We explain the computation steps of the example following the "program" of Theorem 3.12 simultaneously looking at the diagram representation from Figure 3.1. In the initial data, a counter object $d_0$ is placed in the

membrane with label $d$, a couple of auxiliary objects $a_0$ and $l'_0$ are placed in membranes $a$ and 0, respectively, and the input multiset is introduced in the input membrane 0 – see diagram 0.

The computation steps are the following:

st. 1,2: diagram 1 – rules 1,2, and 3 are simultaneously performed, so that, in two steps, the input multiset is re-encoded into membranes 1,2, and 3, while the counter increases its subscript until 2 in membrane $d$, and the objects $l'_0$ and $a_0$ are evolved to $l'_2$ and $a_2$, respectively.

step 3: diagram 2 – object $l'_2$ evolves to the multiset $l_1 l_2 l_3$ by rule 5 while the counter increases and $a_2$ evolves to $a_3$ by rule 3.

step 4: diagram 3 – besides, using rules 1 and 3, objects $l_1$, $l_2$, and $l_3$ enter into membranes 1, 2, and 3, respectively, by applying rule 6.

step 5: diagram 4 – using rule 7, the objects $l_1$, $l_2$, and $l_3$ are sent out their membranes while the membranes change their labels to 0′; object $a_1$ is introduced into membrane 0 from outside by rule 8 too.

step 6: diagram 5 – the counter indicates that this is the 6th step ($d_6$); object $a_1$ divides its surrounding membrane into two new membranes with labels 0, while the object itself evolves to the objects $b_1$ (true) and $c_1$ (false), which are placed in the new membranes (rule 9).

step 7: diagram 6 – object $b_1$ evolves to $t_{11}t_{21}t_{31}$ and object $c_1$ evolves to $f_{11}f_{21}f_{31}$, respectively, by rule 10.

step 8: diagram 7 – by rule 11, the truth values $t_{i,j}$ and $f_{i,j}$ are introduced into membranes with label 0′, and also $a''_1$ are introduced in membranes 0 by rule 12.

step 9: diagram 8 – while the counter still grows, $a'''_1$ are introduced from $a''_1$s by rule 14; the objects $t_{i,j}$ are sent out the surrounding membranes changing their labels to 1′, resp., $f_{i,j}$ are sent out and change the membrane labels to 1″ by rule 15.

step 10: diagram 9 – object $x_{11}$ evolves to $e_1$ in membrane 1′ while object $x'_{21}$ evolves to $e_2$ in membrane 1″, by rule 15; the next "divider" object $a_2$ is introduced by rule 16.

step 11: diagram 10 – the division operation is repeated by rule 9.

st. 12-15: diagram 11-14 – the operations performed by rules 1,10-16 are repeated.

step 16: diagram 15 – objects $a_3$ from membranes 0 are sent out and the labels of those membranes are changed to 1, by rule 17.

step 17: diagram 16 – in a single step, objects $e_i, 1 \leq i \leq 3$, are introduced into membranes 1, changing their label to 2 by rule 18.

st. 18,19: diagram 17 – objects $e_1$ and $e_2$ leave the surrounding membranes one after the other and increase their label in one; there is no object $e_3$ in membrane $a$, hence, rules 20-22 will not be used any longer.

st. 20-22: diagram 18 – the counter object evolves until $d_{22}$ in three consecutive steps.

step 23: diagram 18 – object $d_{23}$ is introduced into the skin membrane by rule 23.

step 24: diagram 18 – the answer NO is sent out into the environment.

In the following theorem, we use cooperative rules, division for non-elementary membranes, and objects move in and out, but this time we do not use the label changing.

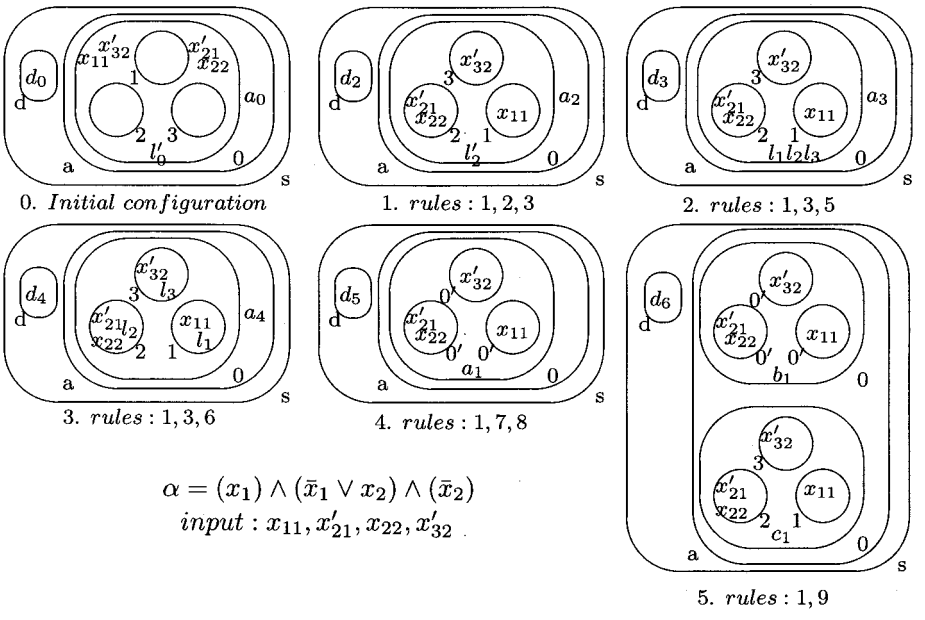
**Theorem 3.13** *P systems constructed in a uniform manner, working in the minimally parallel mode using rules of types* cev, ndiv, in, out *can solve* SAT *in linear time.*

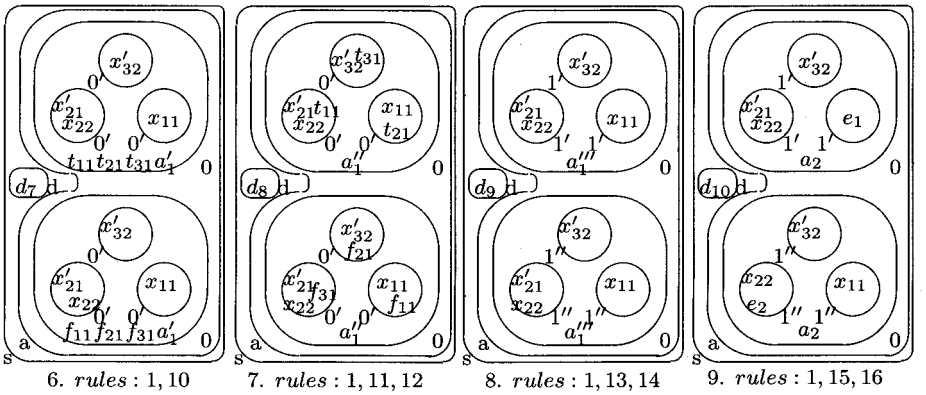
*Proof.* We start from a propositional formula in the conjunctive normal form, $\alpha$, as in the previous proofs. We encode it as in the proof of Theorem 3.12, in form of a multiset $w$, and we construct a recognizing P system $(\Pi(\langle n,m\rangle), \Sigma(\langle n,m\rangle), 0)$ with:

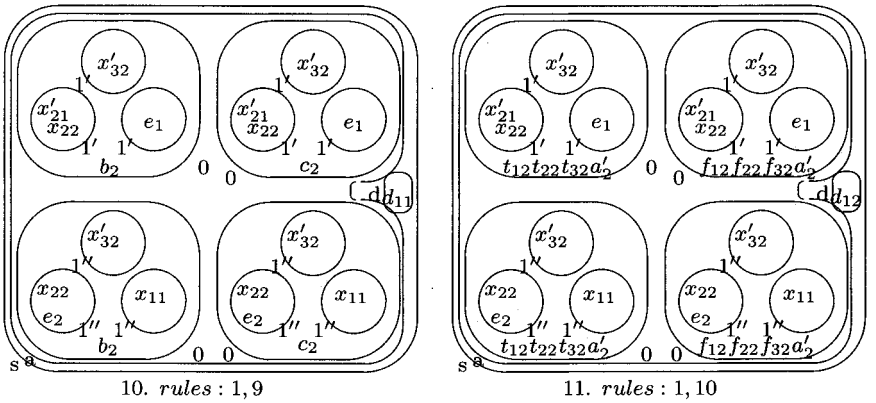
$$\begin{aligned}
\Pi(\langle n,m\rangle) &= (O(\langle n,m\rangle), H, \mu, (w_h)_{h\in H}, (R_h)_{h\in H}),\\
O(\langle n,m\rangle) &= \{x_{i,j}, x'_{i,j}, t_{j,i}, f_{j,i} \mid 1 \leq i \leq m, 1 \leq j \leq n\} \cup \{\mathtt{YES}, \mathtt{NO}\},\\
&\cup \{r_i, a'_i, a''_i \mid 0 \leq i \leq n\} \cup \{d_i \mid 0 \leq i \leq 5n+m+7\}\\
&\cup \{a_i \mid 0 \leq i \leq n+1\} \cup \{e_i, e'_i, \mid 1 \leq i \leq m\},\\
\Sigma(\langle n,m\rangle) &= \{x_{i,j}, x'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\},\\
\mu &= [\,[\,[\,[\,]_m \cdots [\,]_2[\,]_1]_0]_a]_s,\\
w_0 &= r_0, w_s = d_0, w_a = e'_1, w_i = \lambda, 1 \leq i \leq m,\\
H &= \{s, a\} \cup \{i \mid 0 \leq i \leq m\},
\end{aligned}$$

The set $(R_h)_{h\in H}$ of rules of P system $\Pi(\langle n,m\rangle)$, to which we also give explanations about, are the following:

1. $[\, d_i \to d_{i+1}]_s, 0 \leq i \leq 5n+m+5.$


0. Initial configuration
1. rules : 1, 2, 3
2. rules : 1, 3, 5
3. rules : 1, 3, 6
4. rules : 1, 7, 8
5. rules : 1, 9
$\alpha = (x_1) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_2)$
input : $x_{11}, x'_{21}, x_{22}, x'_{32}$


6. rules : 1, 10
7. rules : 1, 11, 12
8. rules : 1, 13, 14
9. rules : 1, 15, 16


10. rules : 1, 9
11. rules : 1, 10

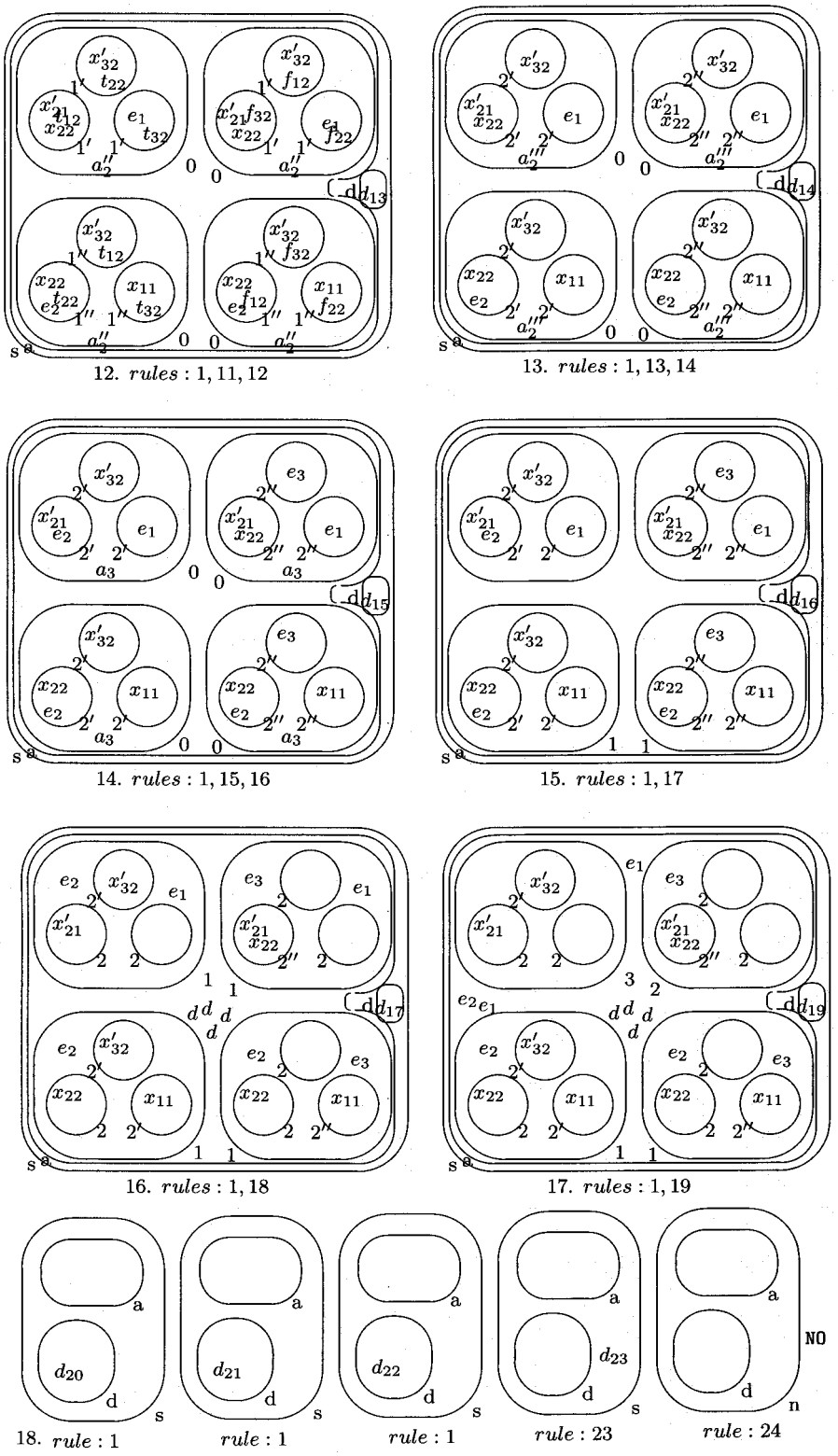



Figure 3.1: An example of solving SAT in linear time. The pictures are read left to right.

This time counter objects take place in the skin region.

*Initialization phase:*

2. $x_{i,j}[\ ]_i \to [\ x_{i,j}]_i$,
   $x'_{i,j}[\ ]_i \to [\ x'_{i,j}]_i$, $1 \leq i \leq m, 1 \leq j \leq n$,
   $[\ r_i \to r_{i+1}]_0, 0 \leq i \leq n-1$,
   $[\ r_n \to a_0]_0$.

As above, the input set of $\alpha$ is re-encoded in $m$ membranes in $n$ steps.

3. $[\ a_i]_0 \to [\ t_{i+1}]_0[\ f_{i+1}]_0, 0 \leq i \leq n-1$.

We generate $2^n$ membranes with label 0 by using the division of non-elementary membrane (rule 3). In each step, $t_i$ and $f_i$ correspond to the truth values *true* and *false*, respectively, for variable $x_i$.

4. $[\ t_i \to t_{i,1}t_{i,2}\ldots t_{i,m}a'_i]_0$,
   $[\ f_i \to f_{i,1}f_{i,2}\ldots f_{i,m}a'_i]_0, 1 \leq i \leq n$.

Objects $t_i$ and $f_i$ produce $m$ copies of objects $t$ and $f$, as well as an object $a'_i$, in each membrane 0.

5. $t_{i,j}[\ ]_j \to [\ t_{i,j}]_j$,
   $f_{i,j}[\ ]_j \to [\ f_{i,j}]_j$, $1 \leq j \leq m$,
   $[\ a'_i \to a''_i]_0$.

Newly produced objects $t_{i,j}$ and $f_{i,j}$ $(1 \leq j \leq m)$, associated with variable $x_i$, are introduced into the corresponding membranes $j$ $(1 \leq j \leq m)$ by rules 5 in a single step. In each membrane with label 0, the rule $[\ a'_i \to a''_i]_0$ is simultaneously performed.

6. $[\ x_{j,i}t_{i,j} \to e'_j]_j$,
   $[\ x'_{j,i}f_{i,j} \to e'_j]_j, 1 \leq j \leq m, 1 \leq i \leq n$,
   $[\ a''_i \to a_{i+1}]_0$.

If some clauses $C_j, 1 \leq j \leq m$, are satisfied by a truth value for variable $x_i$ then, the corresponding objects $x_{j,i}$ and $t_{i,j}$, ($x'_{j,i}$ and $f_{i,j}$, respectively), interact in the corresponding membranes $j, 1 \leq j \leq m$, and introduce the objects $e_j, 1 \leq j \leq m$, by rules 6, indicating clauses which are satisfied. In this step, the object $a''_i$ evolves to $a_{i+1}$ in each membrane 0. Four steps elapse between two division operations, hence $4n$ steps are needed to exhaust variables $x_1, \ldots, x_n$.

*Checking phase:*

7. $[\ a_{n+1} \rightarrow r_1 \ldots r_m]_0$.
8. $r_i[\ ]_i \rightarrow [\ r_i]_i, 1 \leq i \leq m$.
9. $[\ r_i e_i' \rightarrow e_i]_i, 1 \leq i \leq m$.

In each membrane 0, $m$ copies of object $r$ are produced by rule 7, and then, they are introduced into each membrane $i, 1 \leq i \leq m$, by rule 8. Using rule 9, objects $e_i, 1 \leq i \leq m$, are produced by means of an interaction of objects $e_i'$ and $r_i$, in each membrane $i, 1 \leq i \leq m$, in one step.

10. $[\ e_i]_i \rightarrow [\ ]_i e_i, 1 \leq i \leq m$.

Then, in a single step, each membrane $i, 1 \leq i \leq m$, expels the corresponding object $e_i$, indicating that clause $C_i$ is satisfied in membrane $i$.

11. $[\ e_1 e_2 \rightarrow e_2']_0$,
$[\ e_i' e_{i+1} \rightarrow e_{i+1}']_0, 2 \leq i \leq m-1$.

By the application of rule 11, in $m-1$ steps, we check whether all clauses are satisfied in a membrane 0. After these operations, if the formula $\alpha$ is satisfied, we get an object $e_m'$ in some membranes with label 0. Up to now, we have made $5n+m+4$ computation steps.

12. $[\ e_m']_0 \rightarrow [\ ]_0 e_m'$.

Membranes 0 expel object $e_m'$, if such an object exists, by rule 12. Then, one of $e_m'$s, non-deterministically chosen if there are several, interacts with object $e_1'$ (which has been waiting here from the beginning) in membrane $a$ (rule 13), and produces the object $a_0'$. If no object $e_m'$ is presented here, no rule is applied in this step, and object $e_1'$ remains unchanged.

13. $[\ e_m' e_1' \rightarrow a_0']_a$.
14. $d_{5n+m+6}[\ ]_a \rightarrow [\ d_{5n+m+7}]_a$.
15. $[\ a_0' d_{5n+m+7} \rightarrow \texttt{YES}]_a$ *or*
$[\ e_1' d_{5n+m+7} \rightarrow \texttt{NO}]_a$.

In the next step, the counter object $d_{5n+m+7}$ enters into membrane $a$, where it interacts with the objects $a_0'$ or $e_1'$ present here producing thus, one of the objects `YES` or `NO`, by means of rules 14 and 15.

16. $[\ \texttt{YES}]_a \rightarrow [\ \ ]_a \texttt{YES}$ *or* $[\ \texttt{NO}]_a \rightarrow [\ \ ]_a \texttt{NO}$.
17. $[\ \texttt{YES}]_s \rightarrow [\ \ ]_s \texttt{YES}$ *or* $[\ \texttt{NO}]_s \rightarrow [\ \ ]_s \texttt{NO}$.

Either YES or NO will be introduced into the skin region, and from here sent to environment. Thus, the problem is solved in $5n + m + 10$ steps. The observation that the system is of polynomial size completes the proof. □

The main contribution of this section is the use of the minimal parallelism in the framework of P systems with active membranes, without using membrane polarizations. Both universality and efficiency results were proved in this framework, for various combinations of types of rules.

Besides possible improvements of the previous theorems, it should also be investigated the possibility to obtain similar results concerning universality and efficiency results for other classes of P systems working in the minimally parallel mode, especially, when rules of the types ev, out, sep are available.

## 3.3 Replicative-Distribution Rules

The biological motivations of *replicative-distribution* operations are mentioned in Section 1.2. Mathematically, we capture the idea of replicative-distribution rules as following:

(rds) $a[\,]_{h_1}[\,]_{h_2} \to [\,u]_{h_1}[\,v]_{h_2}$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$
(replicative-distribution rule (for sibling membranes); an object is replicated and distributed into two adjacent membranes);

(rdn) $[\,a[\,]_{h_1}]_{h_2} \to [\,[\,u]_{h_1}]_{h_2} v$, for $h_1, h_2 \in H, a \in O, u, v \in O^*$
(replicative-distribution rule (for nested membranes); an object is replicated and distributed into a directly inner membrane and outside the directly surrounding membrane).

The rules are applied non-deterministically, in the maximally parallel manner. Note that the multisets $u$ and $v$ might be empty.

The label changing versions of replicative-distribution rules are of the following form:

(rds′) $a[\,]_{h_1}[\,]_{h_2} \to [\,u]_{h_3}[\,v]_{h_4}$ for $h_i \in H, 1 \leq i \leq 4$
(the label of both or only one membrane can be changed);

(rdn′) $[\,a[\,]_{h_1}]_{h_2} \to [\,[\,u]_{h_3}]_{h_4} v$, for $h_i \in H, 1 \leq i \leq 4$
(the label of both or only one membrane can be changed).

### 3.3.1 Computational Universality

In the previous sections we have seen that P systems with active membranes and with particular combinations of several types of rules can reach universality. Here, we show that P systems with only one type of rules, namely rdn′, are Turing complete. The proof is based on the simulation of matrix grammars with appearance checking.

**Theorem 3.14** $PsOP_4(\mathsf{rdn}') = PsRE$.

*Proof.* It is enough to prove that any recursively enumerable set of vectors of non-negative integers can be generated by a P system with active membranes using rules of type $\mathsf{rdn}'$ and four membranes.

Consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the binary normal form, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$ and with the matrices of the four forms introduced in Section 2.1.3. Assume that all matrices are injectively labeled with elements of a set $B$. Replace the rule $X \to \lambda$ from matrices of type 4 by $X \to f$, where $f$ is a new symbol.

We construct the polarizationless P system of degree 4

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, w_1, w_2, w_{X_{init}}, R),\\
O &= T \cup N_2 \cup \{A_m \mid A \in N_2, m \in B\} \cup \{c, c', c'', c''', \lambda, \#\},\\
H &= N_1 \cup \{X_m \mid X \in N_1, m \in B\} \cup \{0, 1, 2, f\},\\
\mu &= [\,[\,[\,[\,]_2]_1]_{X_{init}}]_0,\\
w_0 &= cA_{init}, w_{X_{init}} = w_1 = w_2 = \lambda.
\end{aligned}$$

and the set $R$ containing the rules below.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1, Y \in N_1 \cup \{f\}$, and $A \in N_2$, is done in two steps, using the following replicative-distribution rules:

1. $[\,A[\,]_X]_0 \to [\,[\,A_m]_{Y_m}]_0\lambda$.
2. $[\,A_m[\,]_1]_{Y_m} \to [\,[\,\lambda]_1]_Y x$.

The first rule of the matrix is simulated by the change of the label of membrane $X$, and the correctness of this operation is obvious (one cannot simulate one rule of the matrix without simulating at the same time also the other rule).

The simulation of a matrix $m : (X \to Y, A \to \#)$, with $X, Y \in N_1$, and $A \in N_2$, is done in four steps, using the rules:

3. $[\,c[\,]_X]_0 \to [\,[\,c']_{Y_m}]_0\lambda$.
4. $[\,c'[\,]_1]_{Y_m} \to [\,[\,c'']_1]_{Y'_m}\lambda$.
5. $[\,A[\,]_{Y'_m}]_0 \to [\,[\,\#]_f]_0\lambda$,
   $[\,c''[\,]_2]_1 \to [\,[\,\lambda]_2]_1 c'''$.
6. $[\,c'''[\,]_1]_{Y'_m} \to [\,[\,\lambda]_1]_Y c$.

By using rule 3, object $c$ replicates to $c'$ and $\lambda$ which are distributed, in the same time, as follows: $c'$ enters membrane $X$ changing its label to $Y_m$ and $\lambda$ is send out of the skin membrane. The second step (rule 4) makes $c'$ to evolve to $\lambda$ and $c''$; $c''$ will be sent to membrane 1 and $\lambda$ gets out of membrane $Y_m$ changing it to $Y'_m$. In the next step, if any copy of $A$ is present, then, it

introduces the trap-object # and the computation never stops. Otherwise, $c''$ following the same replicative-distribution rule transforms into $\lambda$ and $c'''$, which enter membranes 1 and $Y'_m$, respectively. The last computational step produces the result we were looking for by replicating $c'''$ to $\lambda$ and $c$ and distributing $\lambda$ to membrane 1 and $c$ to the skin membrane, changing label $Y_m$ to $Y$. Now, the process can be iterated having $c$ in the skin membrane as in its initial configuration.

We also consider the following rules (applicable in the case $A$ is present in the skin membrane):

7. $[\ \#[\ ]_1]_f \rightarrow [\ [\ \lambda]_1]_f\#$.
8. $[\ \#[\ \ ]_f]_0 \rightarrow [\ [\ \#]_f]_0\lambda$.

Finally, we consider the rules

9. $[\ A[\ \ ]_f]_0 \rightarrow [\ [\ \#]_f]_0\lambda$ for all $A \in N_2$.

which checks that when using the matrix of type 4 of $G$, the derivation is terminal.

The equality $\Psi_T(L(G)) = Ps(\Pi)$ easily follows from the above explanations. □

### 3.3.2 Computational Efficiency

We use again the most investigated way to obtain exponential work space–membrane division. The following theorem shows that SAT can be solved in linear time by P systems with polarizationless active membranes using the rules of types ndiv and rdn′. We recall here the propositional formula $\varphi$ from the proof of Theorem 3.11.

**Theorem 3.15** *P systems with rules of types* ndiv *and* rdn′*, constructed in a semi-uniform manner, can deterministically solve* SAT *in linear time with respect to the number of variables and the number of clauses.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form:

$$\begin{aligned} \varphi &= C_1 \wedge \cdots \wedge C_m, \\ Var(\varphi) &= \{x_1, \ldots, x_n\}, \\ C_i &= y_{i,1} \vee \cdots \vee y_{i,l_i},\ 1 \leq i \leq m, \text{ where} \\ y_{i,k} &\in \{x_j, \neg x_j \mid 1 \leq j \leq n\},\ 1 \leq i \leq m, 1 \leq k \leq l_i. \end{aligned}$$

We construct the following P system which is associated to the formula $\varphi$:

$$\begin{aligned}
\Pi &= (O, H, \mu, w_0, \cdots, w_7, R), \text{ with} \\
O &= \{d_i \mid 1 \leq i \leq m\} \cup \{a_i \mid 1 \leq i \leq n\} \\
&\cup \{c_i \mid 1 \leq i \leq m\} \cup \{b_i \mid 0 \leq i \leq n\} \\
&\cup \{e_i \mid 0 \leq i \leq 2n+m+4\} \cup \{\texttt{YES}, \texttt{NO}\} \\
&\cup \{t_i, f_i \mid 1 \leq i \leq n\}, \\
\mu &= [\,[\,[\,[\,]_3]_4]_2[\,[\,[\,]_6]_7]_5]_0, \\
w_2 &= a_1 \cdots a_n b_0, w_5 = e_0, w_0 = w_2 = w_3 = w_4 = w_6 = w_7 = \lambda, \\
H &= \{i \mid 0 \leq i \leq 9\},
\end{aligned}$$

The rules of the P system $\Pi$, to which we also give explanations about, are the following:

*Global control:*

E1. $[\, e_i[\,]_7]_5 \rightarrow [\,[\, e_{i+1}]_7]_5\lambda$.

E2. $[\, e_i[\,]_6]_7 \rightarrow [\,[\,\lambda]_6]_7 e_{i+1}, 0 \leq i \leq 2n+m+1$.

The "nested" membranes with label 5,7, and 6 are used only for the global control of the computation. Rules E1 and E2 are used to count the computation steps.

*Generation phase:*

G1. $[\, a_i]_2 \rightarrow [\, t_i]_2[\, f_i]_2$, $1 \leq i \leq n$.

Using rule G1, with $a_i$ non-deterministically chosen, we produce the truth values $true$ and $false$ assigned to variable $x_i$, which are placed in two separate copies of membrane 2. In this way, when we assign truth values to all variables in $n$ steps, we get all $2^n$ truth assignments, placed in $2^n$ separate copies of membrane 2.

G2. $[\, b_i[\,]_4]_2 \rightarrow [\,[\, b_{i+1}]_4]_2\lambda$.

$[\, b_i[\,]_3]_4 \rightarrow [\,[\,\lambda]_3]_4 b_{i+1}$, for all $0 \leq i \leq n-1$.

G3. $[\, b_n[\,]_3]_4 \rightarrow [\,[\,\lambda]_3]_1\lambda$.

G4. $[\, b_n[\,]_4]_2 \rightarrow [\,[\,\lambda]_1]_2\lambda$.

Initially, object $b_0$ is placed in membrane 2. Rule G2 works simultaneously to division and increases the subscript of $b_i$ in one in each step. If in the $n$th step of the computation, object $b_n$ takes place in membrane 2, then $n$ was an odd number. If it was an even number, then object $b_n$ takes place in membrane 4. At the next step, rules G3 or G4 can be applied, and they change the label of membrane 4 to 1, while object $b_n$ disappears. This ensures that rule G5 can be used.

G5. $[\ t_i[\ ]_1]_2 \to [\ [\ v_i]_1]_2\lambda$,
$[\ f_i[\ ]_1]_2 \to [\ [\ v_i']_1]_2\lambda$, $1 \leq i \leq n$.

*Checking phase:*

C1. $[\ c_i[\ ]_3]_i \to [\ [\ c_i]_3]_{i+1}d_i, 1 \leq i \leq m$.

In the checking phase, by using rule C1, object $c_i, 1 \leq i \leq n$, is placed in membranes labeled by $4n+5-m$ of level $n+1$, and it is replicated into object $c_i$ and counter object $d_i$. Object $c_i$ is sent into the direct inner elementary membrane with label $d$, which is on the deepest level $(n+2)$ of our membrane structure, and object $d_i$ is sent out the surrounding membrane on $n$th level. Meanwhile, the label of the surrounding membrane is increased in one. If at the beginning of the checking phase $c_1, \ldots, c_i$ are present $(1 \leq i < m)$, and $c_{i+1}$ is absent in the membrane, rule C1 will no longer be applicable after $i+1$ steps, and the membrane will never change the label again. If all objects $c_i, 1 \leq i \leq m$, are present in some membranes, then after $m$ steps, objects $d_m$ are produced into the membranes $2n+1$ and $2n+2$ of level $n$.

*Output phase:*

O1. $[\ d_m[\ ]_{m+1}]_2 \to [\ [\ \lambda]_{m+1}]_8 d_m$.
O2. $[\ d_m[\ ]_8]_0 \to [\ [\ \lambda]_8]_9$YES.

If $\varphi$ has solutions, after $2n+m+2$ steps, objects $d_m$ appear in the skin membrane using rules O1, and again one object $d_m$, non-deterministically chosen, object YES is sent to the environment. At the same time, the skin label changes to 9 using rule O2 in order to prevent further output. Thus, the formula is satisfiable and the computation stops. That was the $(2n+m+3)$th step of the whole computation.

E3. $[\ e_{2n+m+2(3)}[\ ]_7]_5 \to [\ [\ \lambda]_7]_5 e_{2n+m+3(4)}$.
E4. $[\ e_{2n+m+3(4)}[\ ]_5]_0 \to [\ [\ \lambda]_5]_0$NO.

If $\varphi$ has no solution and if $2n+m+2$ is an odd step, after two more steps the counter object $e_{2n+m+4}$ will expel the correct answer NO to the environment. Otherwise, after one more step object $e_{2n+m+3}$ will perform this operation. Since rule O2 did not apply (the case in which $\varphi$ has no solution), the label of the skin membrane is still 0, so rule E4 is applicable. □

A computation in a P systems with active membranes always starts from a given initial configuration, and we usually create an exponential workspace in linear time by membrane division, membrane creation, string replication, or membrane separation. One can also use a different strategy (which will be also used in Section 5.5): starting from an arbitrarily large initial membrane

structure, without objects placed in its regions, we trigger a computation by introducing objects related to a given problem in a specified membrane. We use object replicative-distribution rules, as discussed in above. In this way, the number of objects can increase exponentially. Two results of this type were given in [52], but we skip them here.

The considered replicative-distribution operations rds and rdn are motivated from biological cell functioning and they are proved to be computationally powerful and efficient tools in P systems. In the next chapter, we will reconsider these operations.

# Chapter 4

# Membrane Systems with Neural-Like Operations

In the present chapter, we mainly focus on the behavior of a single neural cell, considering both its structure and some functions in the aim of modeling them in the area of membrane systems.

We have explored some biochemical operations of the neural cells (neural-operations): impulse transmission and propagation, and excitation/inhibition impulses.

### Using Neural-Operations in P Systems

Although we have mentioned the biological backgrounds of the our neural operations in Section 1.2, let us be a bit more precise in order to make the connection to P systems.

A neuron has a *body*, the *dendrites*, which form a very fine filamentary bush around the body of the neuron, and the *axon*, a unique, long filament, which in turn also ends with a fine filamentous bush; each of the filaments from the end of the axon is terminated with a small bulb. It is by means of these end-bulbs and the dendrites that the neurons are linked to each other: the impulses are sent through the axon, from the body of the neuron to the end-bulbs, and the end-bulbs transmit the impulses to the neurons whose dendrites they touch. Such a contact junction between an end-bulb of an axon and dendrites of another neuron is called cleft.

In what concerns the operations that can be performed in a neuron, the proposed models of computation in P systems are based on various combinations of the primitive *neural-operations* of *excitation/inhibition.* There are two basic types of impulses: excitatory impulses increase the membrane potential, whereas inhibitory impulses decrease the membrane potential. If different impulses reach at the same time a certain node, then it might happen that the combined effects of the excitation and inhibition may cancel each other or excite new impulses. Once the threshold of the membrane

potential is reached, an impulse is propagated along the neuron or to the next neuron.

## 4.1 Exciting/Inhibiting (On/Off) Operation

It is possible to introduce the impulse excitation and inhibition mechanism in the P systems area by using evolution rules equipped with the ability to send excitatory/inhibitory signals. An inhibited rule is formally written as $r : \neg(u \rightarrow v)$, and the meaning is that the rule cannot be applied. An evolution rule can de-inhibit an inhibited rule allowing it to be applied. To this aim, we also consider rules of the form $r : (u \rightarrow v)\{r_1, \ldots, r_k\}$, which say that, when the rule is applied, $u$ evolves into $v$ and the rules $r_1, \ldots, r_k$ are inhibited or de-inhibited, changing their previous states.

We now pass to introduce a new class of P systems with an inhibiting/de-inhibiting mechanism which controls computations and we explore the computational power of the class considering catalytic and non-cooperative inhibiting /de-inhibiting rules. In particular, we prove that universality can be obtained (in generative and accepting cases) by using one catalyst. If we use only non-cooperative rules, then the systems can generate at least the Parikh sets of the languages generated by ET0L systems, but we omit this result to present here.

A P system *with inhibiting/de-inhibiting rules* (in short, an ID P system), of degree $m \geq 1$, is a construct

$$\Pi = (O, C, H, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0),$$

where:

- $m \geq 1$ is the degree of the system;
- $O$ is the alphabet of *objects*;
- $C \subseteq O$ is the set of catalysts;
- To each rule in $R = R_1 \cup R_2 \cup \cdots \cup R_m$, a unique label is associated. The set of all rule labels is $H = \{r_1, \cdots, r_k\}$. We denote $\neg H = \{\neg r_i \mid r_i \in H\}$. For a set $Q$ of rules we indicate with $lab(Q)$ the set of labels of the rules that compose $Q$.
- $\mu$ is a *membrane structure*, consisting of $m$ membranes, labeled $1, 2, \ldots, m$;
- $w_1, \ldots, w_m$ are strings over $O$, describing the *multisets of objects* placed in the $m$ regions of $\mu$;
- $R_i$ is a finite set of *developmental rules*, associated with region $i$. The rules in $R_i$ are of the forms:

$$r_j : \neg(a \to w)S, \quad r_j : (a \to w)S,$$
$$r_j : \neg(ca \to cw)S, \quad r_j : (ca \to cw)S$$

where $r_j \in H, a \in O - C, w \in ((O - C) \times TAR)^*, c \in C, S \subseteq H$, for $TAR = \{here, out, in\}$; when $S = \emptyset$, we omit writing it, and we also omit the parentheses around the rule;

- $i_0$ is the output region.

A configuration of an ID P system is described by using the $m$-tuple of multisets of objects, present in the $m$ regions of the system. To each region a finite number of objects is associated together with a finite number of rules. The $m$-tuple $(w_2, w_2, \cdots, w_m)$ is the initial configuration of the system. Some of the rules are initially inhibited (if the symbol $\neg$ is written immediately before the rule). A transition between two configurations is governed by the application in a non-deterministic and maximally parallel way of the rules that are not inhibited.

When a rule $r_j : (a \to w)S$ is applied, the object $a$ is rewritten with the objects in $w$ (as in standard P systems) and the rules from $S$ are de-inhibited (if they were inhibited) or inhibited (if they were de-inhibited). In the same way is defined the application of catalytic rules.

For simplicity, each element in $S$ is called a *switch*.

If simultaneously a rule $r$ is subject of two or more switches, then the effect is that if a single switch, hence a change from inhibited to de-inhibited or conversely.

The system continues the application of the rules in maximally parallel way until there remain no applicable rules in any region of the system. Then the system halts (the computation is successful) and we consider the number of objects contained in the output region $i_0$ as the result of the computation.

We use the notation

$$Ps_{gen}IDP_m(\alpha), \alpha \in \{ncoo\} \cup \{cat_k \mid k \geq 0\},$$

to denote the family of sets of vectors of natural numbers generated by ID P systems with at most $m$ membranes, evolution rules that can be non-cooperative ($ncoo$), or catalytic ($cat_k$), using at most $k$ catalysts (as usual, $*$ indicates that the corresponding number is not bounded). When using systems in the accepting case, the subscript $gen$ is replaced by $acc$.

**An Example** We now illustrate the functioning of an ID P system with an example that show how the simple mechanism of inhibiting/de-inhibiting rules can be very powerful.

We consider the ID P system *of degree 1*,

$$\Pi = (\{A, a\}, \emptyset, \{r_1, r_2, r_3\}, [\,]_1, A, R, 1),$$

where:

$$R = \{r_1 : A \to AA,\ r_2 : (A \to AA)\{r_1, r_2, r_3\},\ r_3 :\ \neg(A \to a)\}.$$

When the computation starts in the initial configuration with the object $A$ in the skin region, the rules $r_1$ or $r_2$ can be applied but the rule $r_3$ cannot be applied since it is inhibited. We use the rule $r_1$ $m-1$ times and then we apply the rule $r_2$. In this way, we produce $2^m$ copies of object $A$. Simultaneously the rules $r_1$ and $r_2$ are inhibited (so they cannot be used any more), and the rule $r_3$ is de-inhibited. We have used context-free and inhibiting/de-inhibiting rules to obtain $a^{2^m}$, hence

$$\{(2^m) \mid m \geq 1\} \in PsIDP_1(ncoo)$$

and this set is not in $PsCF$.

### 4.1.1 Using One Catalyst: Two Universality Results

In this paragraph we prove the universality of P systems using catalytic rules (one catalyst) and one membrane.

We first consider the generative case.

**Theorem 4.1** $Ps_{gen}IDP_1(cat_1) = PsRE$.

*Proof.* Consider a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the binary normal form, hence with $N = N_1 \cup N_2 \cup \{S, \#\}$ as introduced in Section 2.1.3. Assume that all matrices are labeled in an injective manner with $m_i, 1 \leq i \leq n$, and each terminal matrix $(X \to \lambda, A \to x)$ is replaced by $(X \to f, A \to x)$, where $f$ is a new symbol. We define the set of rules $R_\# = \{X \to \# \mid X \in N_1 \cup N_2\}$.

We construct the P system of degree 1,

$$\begin{aligned}
\Pi &= (O, C, H, \mu, w_1, R_1, i_0), \text{where:} \\
O &= N_1 \cup T \cup N_2 \cup \{p, p', p'', \overline{p'}, \overline{p''}, c, d, d', d'', d''', f, \#\}, \\
C &= \{c\}, \\
H &= \{r_i \mid 1 \leq i \leq 13\} \cup \{r_{2,i}, r_{3,i}, r_{8,i}, r_{9,i}, r_{12,i} \mid 1 \leq i \leq n\} \\
&\cup \{r'_4, r'_5, r'_6\} \cup Lab_\Pi(R_\#), \\
\mu &= [\ ]_1, \\
w_1 &= cpX_{init}A_{init}, \\
i_0 &= 0,
\end{aligned}$$

and the set $R_1$ is constructed in the following way.

- The simulation of a matrix of type 2, $m_i : (X_i \to Y_i, A_i \to x_i)$, with $X_i \in N_1, Y_i \in N_1, A_i \in N_2, x_i \in (N_2 \cup T)^*, |x_i| \leq 2$, is done by using the following rules added to the set $R_1$:

$$
\begin{aligned}
r_1 &: (p \to p'p'')\{r_{2,i}, r_{3,i}\} \\
r_{2,i} &: \neg(X_i \to Y_i)\{r'_5, r_5, r_{2,i}\} \\
r_{3,i} &: \neg(cA_i \to cx_id')\{r'_6, r_6, r_{3,i}\} \\
r_2 &: d' \to d \\
r_3 &: (d \to p)\{r'_5, r'_6\} \\
r_4 &: p' \to \overline{p'} \\
r'_4 &: p'' \to \overline{p''} \\
r_5 &: \neg(\overline{p'} \to \lambda)\{r_5, r'_5\} \\
r'_5 &: \overline{p'} \to \# \\
r_6 &: \neg(\overline{p''} \to \lambda)\{r_6, r'_6\} \\
r'_6 &: \overline{p''} \to \#
\end{aligned}
$$

The idea is the following one. The rule $r_1$ chooses the matrix $i$ to apply and this is made by the simultaneous de-inhibition of rules $r_{2,i}$ and $r_{3,i}$ (all other rules of other matrices remains inhibited). The execution of both $r_{2,i}$ and $r_{3,i}$ inhibits the rules $r'_5$ and $r'_6$ that are used to trash the computation in the case the matrix chosen is not correctly applied. If the matrix is applied in the correct way (both the rules are executed), then $d$ is changed to $p$ and the process can be iterated (the original configuration of inhibited / de-inhibited rules is re-established).

- The simulation of a matrix of type 3, $m_i : (X_i \to Y_i, A_i \to \#)$, with $X_i, Y_i \in N_1$ and $A_i \in N_2$, is done by using the following rules, added to the set of rules $R_1$:

$$
\begin{aligned}
r_7 &: (p \to p')\{r_{8,i}, r_{9,i}\} \\
r_{8,i} &: \neg(X_i \to Y_id'')\{r'_5, r_5, r_{8,i}, r_{9,i}\} \\
r_8 &: p' \to \overline{p'} \\
r_{9,i} &: \neg(A_i \to \#) \\
r_9 &: d'' \to d''' \\
r_{10} &: d''' \to p
\end{aligned}
$$

The idea of the simulation of this kind of matrix is the following one. The rule $r_7$ de-inhibits the rules corresponding to the matrix $i$ to be simulated. If the rule $r_{8,i}$ is not applied, then the rule $r'_5$ is not inhibited and then the computation never halts.

- The simulation of a terminal matrix $m_i : (X_i \to f, A_i \to x_i)$, with $X_i \in N_1$, $A_i \in N_2$, and $x_i \in T^*, |x_i| \leq 2$, is done using the following rules (added to the set $R_1$):

$$
r_{11} : (p \to \lambda)(Lab_\Pi(R_\#) \cup \{r_{12,i}\})
$$

$$r_{12,i} : \neg(cA_i \to cx_i)\{r_{12,i}\}$$
$$R_{11} = \{\neg(X \to \#) \mid X \in N_1 \cup N_2.$$

These rules are used to simulate a terminal matrix and then to halt the computation. In fact, $p$ is deleted and the rule $r_{12,i}$ is executed; the rules in $R_{11}$ are de-inhibited and they guarantee that, when the computation halts, only terminal objects are present.

$R_1$ also contains the following rules:

$$r_{12} : a \to (a, out)$$
$$r_{13} : \# \to \#.$$

The result of the computation is collected in the environment; from the above explanation follows that the set of vectors generated by $\Pi$ is exactly the Parikh image of $L(G)$. □

We consider now the accepting case. We can notice that, in the following proof, the switches are used only by non-cooperative rules.

**Theorem 4.2** $Ps_{acc}IDP_1(cat_1) = PsRE.$

*Proof.* In order to prove this assertion we will simulate an $m$–register machine $M = (m, B, l_0, l_h, I)$ (see Section 2.1.6). At each time during the computation, the current contents of register $r$ is represented by the multiplicity of the object $a_r$.

Formally, we define the P system *of degree 1*,

$$\begin{aligned}
\Pi &= (O, C, H, [\,]_1, w_1, R, i_0), \text{where:} \\
O &= \{a_r, S_r, S'_r, S''_r, S'''_r \mid 1 \leq r \leq m\} \cup \{c, e, e', F, p, l_h\} \cup B, \\
C &= \{c\}, \\
H &= \{r_i \mid 1 \leq i \leq 15\}, \\
w_1 &= cl_0, \\
i_0 &= 0,
\end{aligned}$$

and $R$ is defined as follows.

- for each instruction $l_i : (\mathsf{add}(r), l_j, l_j) \in I$, we add to $R$ the rule:

$$r_1 : l_i \to a_r l_j$$

- for each instruction $l_i : (\mathsf{sub}(r), l_j, l_k) \in I$, we add to $R$ the rules:

$$r_2 : l_i \to eS_r$$
$$r_3 : e \to e'$$

$$
\begin{aligned}
r_4 &: (S_r \to S_r')\{r_7\} \\
r_5 &: \neg(ca_r \to cF) \\
r_6 &: S_r' \to S_r'' \\
r_7 &: (F \to \lambda)\{r_7, r_{12}\} \\
r_8 &: S_r'' \to S_r''' \\
r_9 &: (S_r''' \to \lambda)\{r_{13}\} \\
r_{10} &: \neg(e' \to l_j p)\{r_{12}\} \\
r_{11} &: \neg(e' \to l_k)\{r_{13}, r_7\} \\
r_{12} &: (p \to \lambda)\{r_{13}\} \\
r_{13} &: l_h \to \lambda
\end{aligned}
$$

The system works as follows. We start by introducing in the system the objects $a_1^{k_1}, \ldots, a_m^{k_m}$ (representing the input to be accepted). We also have inside the catalyst $c$ and and the label $l_0$ of the first instruction of the register machine. The vector $(k_1, \cdots, k_m)$ represents the vector that has to be accepted by our P system.

The addition instructions are directly simulated by the corresponding rules $r_1$.

If a subtraction instruction $(l_i : \mathsf{sub}(r), l_j, l_k) \in I$ has to be simulated then the rule $l_i \to eS_r$ is executed. In the following step the object $S_r$ is used to de-inhibit rule $r_5$, and to produce object $S_r'$ in rule $r_4$, while the object $e$ evolves into $e'$.

In the next step, if the number of objects $a_r$ in register $r$ is greater than 0, then the execution of the de-inhibited rule $ca_r \to cF$ decreases the number of objects $a_r$ by 1, and produces object $F$ for the next step.

Meanwhile, the object $S_r'$ evolves into $S_r''$ as shown in rule $r_6$.

The object $F$ used in rule $r_7$ to inhibit the de-inhibited rule $r_5$ guarantees that $r_5$ is applied only once. The rule $r_{10}$ is also de-inhibited by the execution of the rule $r_7$.

The execution of the $r_{10}$ generates the label $l_j$ of the next register machine instruction (it is executed only once because it is inhibited by itself)

In the other case (i.e., if there is no object $a_r$ in region) the rule $a_r \to F$ cannot be executed, therefore the object $F$ is not produced and rule $r_7$ cannot be applied.

On the other hand, object $S_r''$ evolves into $S_r'''$ by the execution of rule $r_8$ and at the next step $S_r'''$ de-inhibits rule $r_{11}$, so $e'$ evolves to label $l_k$ (notice that the object $e'$ still appears, because rule $r_{10}$ has not been applied in the previous steps and then rule $r_{11}$ can generate label $l_k$ of the next register machine instruction); rule $r_{11}$ must also inhibit rule $r_5$ that has been previously de-inhibited by rule $r_4$.

In the next step the rule $r_{12}$ is executed and then the rule $r_{11}$ in again inhibited (in the case that the rule has not been used because the rule $r_{10}$ has

been applied). When the label of the next instruction has been generated then the entire process can be iterated. The simulation stops (and then the input is accepted) when label $l_h$ (which stands for halt instruction) is generated. □

### 4.1.2 Using Non-Cooperative Rules and One Switch

We give here the next result without proof. ID P systems using non-cooperative rules are able to generate (at least) the family of Parikh images of languages in ET0L. The proof is made by simulating an ET0L system by using a very restricted ID P system with at most one switch for each non-cooperative evolution rule, [15].

**Theorem 4.3** $PsIDP_1(ncoo) \supseteq PsET0L$.

## 4.2 Inhibiting/De-inhibiting (AID) P Systems with Active Membranes

As above, the basic idea of the AID P systems model is that, when a rule (acting on the membranes or on the objects) is inhibited, then it cannot be applied until another rule de-inhibits it. The application of a rule can inhibit other rules (and in particular might inhibit itself).

A P system *with active membranes and inhibiting/de-inhibiting mechanism*, in short, an AID P system, without electrical charges and without using catalysts, is a construct

$$\Pi = (O, H, I, \mu, w_1, \ldots, w_m, R),$$

where:

- $m \geq 1$ is the initial degree of the system;
- $O$ is the alphabet of *objects*;
- $H$ is a finite set of *labels* for membranes;
- $I$ is a finite set of *labels* for rules;
- $\mu$ is a *membrane structure*, consisting of $m$ membranes, labeled with elements of $H$;
- $w_1, \ldots, w_m$ are strings over $O$, describing the *multisets of objects* placed in the $m$ regions of $\mu$;
- $R$ is a finite set of *developmental rules* of various usual forms. Here are some examples:

(in) $r : a[\ ]_h \rightarrow [\ b]_h S$, for $r \in I, h \in H, a, b \in O, S \subseteq I$ (communication rules; an object is introduced in the membrane during this process);

(out) $r : [\ a\ ]_h \rightarrow b[\ ]_h S$, for $r \in I, h \in H, a, b \in O, S \subseteq I$ (communication rules; an object is sent out of the membrane during this process).

The rules in $R$ are written as $r_j : (\neg r)S$ or as $r_j : (r)S$, where $r_j \in I$ and $r$ is a rule of P systems with active membranes from Chapter 3 and $S$ is a subset of $I$. The AID P systems works like general P systems with active membranes. The only difference consists in the fact, that, at each step, only the non-inhibited rules can be used. When a rule $r_j : (r)S$ is applied, the rules whose labels are specified in $S$ are inhibited (if they were de-inhibited) or de-inhibited (if they were inhibited). Now, 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 to an universal clock. The system will make a successful computation if and only if it halts, meaning there is no applicable rule to the objects present in the halting configuration. The result of a successful computation is the number of objects present in the output membrane (or environment) in a halting configuration of $\Pi$. If the computation never halts, then we will have no output.

We use the notation $Ps_{gen}AIDP_m(\alpha)$ to denote the family of sets of vectors of natural numbers generated by AID P systems with at most $m$ membranes ($*$ if the number is not bounded) and developmental rules of types described by $\alpha$.

A system $\Pi$ as above can be also used in the accepting mode in the following way. Given a vector $v$ of natural numbers, let be $x$ a string over the alphabet $O$ such that $\Psi_O(x) = v$; the occurrences of objects corresponding to the multiset described by the string $x$ are inserted in a specified region and the vector $v$ is accepted by the system $\Pi$ if and only if the computation halts.

We use the notation $Ps_{acc}AIDP_m(\alpha)$ to denote the family of sets of vectors of natural numbers accepted by AID P systems with at most $m$ membranes ($*$ if the number is not bounded) and developmental rules of types $\alpha$.

We first investigate the possibility of simulating Boolean circuits by means of AID P systems.

### Boolean Functions and Circuits

An $n$-ary *Boolean function* is a function $f\{true, false\}^n \mapsto \{true, false\}$. $\neg$ (negation) is a unary Boolean function (the other unary functions are the constant functions and the identity function). We say that the Boolean

expression $\varphi$ with variables $x_1, \ldots, x_n$ *expresses* the $n$-ary Boolean function $f$ if, for any $n$-tuple of truth values $t = (t_1, \cdots, t_n)$, $f(t)$ is *true* if $T \models \varphi$, and $f(t)$ is false if $T \not\models \varphi$, where $T(x) = t_i$ for $i = 1, \ldots, n$.

There are three primary Boolean functions that are widely used: The NOT function – this is a just a negation; the output is the opposite of the input. The NOT function takes only one input, so it is called a unary function or operator. The output is true when the input is false, and vice-versa. The AND function – the output of an AND function is true only if all its inputs are true. The OR function – the output of an OR function is true if at least one of its inputs is true. Both AND and OR can have any number of inputs, with a minimum of two.

Any $n$-ary Boolean function $f$ can be expressed as a Boolean expression $\varphi_f$ involving variables $x_1, \ldots, x_n$.

There is a potentially more economical way than expressions for representing Boolean functions–namely *Boolean circuits.* A Boolean circuit is a graph $C = (V, E)$, where the nodes in $V = \{1, \ldots, n\}$ are called the *gates* of $C$. The graph $C$ has a rather special structure. First, there are no cycles in it, so we can assume that all edges are of the form $(i, j)$, where $i < j$. All nodes in the graph have the "in-degree" (number of incoming edges) equal to 0, 1, or 2. Also, each gate $i \in V$ has a *sort* $s(i)$ associated with it, where $s(i) \in \{true, false, \vee, \wedge, \neg\} \cup \{x_1, x_2, \ldots\}$. If $s(i) \in \{true, false\} \cup \{x_1, x_2, \ldots\}$, then the in degree of $i$ is 0, that is, $i$ must have no incoming edges. Gates with no incoming edges are called the *inputs* of $C$. If $s(i) = \neg$, then $i$ has "in-degree" one. If $s(i) \in \{\vee, \wedge\}$, then the in-degree of $i$ must be two. Finally, node $n$ (the largest numbered gate in the circuit, which necessarily has no outgoing edges) is called the *output gate* of the circuit.

This concludes our definition of the *syntax* of circuits. The *semantics* of circuits specifies a truth value for each appropriate truth assignment. We let $X(C)$ be the set of all Boolean variables that appear in the circuit $C$ (that is, $X(C) = \{x \in X \mid s(i) = x \text{ for some gate } i \text{ of } C\}$). We say that a truth assignment $T$ is appropriate for $C$ if it is defined for all variables in $X(C)$. Given such a $T$, *the truth value of gate* $i \in V$, $T(i)$, is defined, by induction on $i$, as follows: If $s(i) = true$ then $T(i) = true$, and, similarly, if $s(i) = false$, then $T(i) = false$. If $s(i) \in X$, then $T(i) = T(s(i))$. If now $s(i) = \neg$, there is a unique gate $j < i$ such that $(j, i) \in E$. By induction, we know $T(j)$, and then $T(i)$ is *true* if $T(j) = false$, and vice-versa. If $s(i) = \vee$, then there are two edges $(j, i)$ and $(j', i)$ entering $i$. $T(i)$ is then *true* if only if at least one of $T(j)$, $T(j')$ is *true*. If $s(i) = \wedge$, then $T(i)$ is *true* if only if both $T(j)$ and $T(j')$ are *true*, where $(j, i)$ and $(j', i)$ are the incoming edges. Finally, *the value of the circuit*, $T(C)$, is $T(n)$, where $n$ is the output gate.

### 4.2.1 Simulating Logical Gates

In this section we present AID P systems which simulate logical gates. We will consider that the input for a gate is given in the inner membrane, while the output will be computed and sent out to the outer region.

#### Simulation of AND Gate

**Lemma 4.1** *Boolean AND gate can be simulated by AID P systems with rules of types* in′ *and* out′*, using two membranes and two objects (only the input), in at most four steps.*

*Proof.* We construct the AID P system

$$\begin{aligned}
\Pi_{AND} &= (O, H, I, \mu, w_0, w_s, R), \text{with} \\
O &= \{0, 1\}, \\
\mu &= [\,[\,]_0]_s, \\
w_0 &= w_s = \lambda, \\
H &= \{0, 1, s\}, \\
I &= \{r_i \mid 0 \leq i \leq 9\},
\end{aligned}$$

and the set $R$ consisting of the following rules:

$$\begin{aligned}
r_1 &: [\,0]_0 \to [\,]_1 0 \\
r_2 &: [\,0]_1 \to [\,]_0 \lambda \{r_2, r_8\} \\
r_3 &: [\,1]_0 \to [\,]_1 1 \{r_2, r_4, r_5, r_6\} \\
r_4 &: [\,1]_1 \to [\,]_0 \lambda \{r_2, r_8\} \\
r_5 &: \neg[\,0]_1 \to [\,]_1 0 \{r_5, r_7\} \\
r_6 &: \neg[\,1]_1 \to [\,]_1 \lambda \{r_4, r_6, r_9\} \\
r_7 &: \neg 1[\,]_1 \to [\,\lambda]_0 \{r_4, r_6, r_7, r_8\} \\
r_8 &: \neg[\,0]_s \to [\,]_s 0 \{r_2, r_8\} \\
r_9 &: \neg[\,1]_s \to [\,]_s 1 \{r_2, r_5, r_9\}
\end{aligned}$$

We start by placing the input values $x_1$ and $x_2$ in the membrane with label 0. Depending on the value of the initial variables $x_1$ and $x_2$, the rules we apply for each of the four cases are:

for $x_1x_2 = 00$ – $r_1, r_2, r_8$; for $x_1x_2 \in \{01, 10\}$ – $r_1, r_4, r_8$, or $r_3, r_5, r_7, r_8$; and for $x_1x_2 = 11$ – $r_3, r_6, r_9$.

More precisely, if two 1s are in membrane 0, in the first step, rule $r_3$ is applied, a 1 is expelled and membrane's label is changed to 1. In the same time according to the inhibition/de-inhibition concept, rules $r_2$ and $r_4$ are inhibited, while rules $r_5$ and $r_6$ are de-inhibited and ready to be used. In the second step we notice that only rule $r_6$ can be applied, thus, object 1,

placed inside membrane labeled 1 is transformed, in its way out, into $\lambda$. One may notice that rule $r_6$, after is applied, restores the original status of rule $r_4$ and itself, and also de-inhibits rule $r_9$. In the third step, rule $r_9$ performs and the right answer 1 is sent out the skin membrane, while rules $r_2$, $r_5$, and $r_9$ come back to their original status.

In other words, after these three steps, our system sends out the skin membrane the right answer (given the input 11) and comes back to its initial configuration, thus being ready for a new input.

In the case when the input is 01 or 10, we can start by using $r_1$ or $r_3$. Let us examine the second case. Rule $r_3$ sends 1 out of membrane 0 and changes its label to 1. In the same time, rules $r_2$ and $r_4$ are inhibited, while rules $r_5$ and $r_6$ are de-inhibited. The only rule we can use in the second step is $r_5$ which expels 0 out of membrane 1, inhibits itself and de-inhibits rule $r_7$. In this moment we have the following configuration of our system $[\ [\ ]_1 01]_s$. We now apply rule $r_7$ which transforms object 1 to $\lambda$ on its way in the inner membrane and changes its label from 1 to 0. Rule $r_7$ de-inhibits the inhibited rule $r_4$, inhibits $r_6$ and itself, and de-inhibits rule $r_8$. The fourth step is the one in which the right answer 0 is sent out skin membrane, while the system gets back to its initial configuration. Thus, our system gives the right answer, in four steps, when we have input 01. In the other two cases (when we have the input 01 and we start by using first the rule $r_1$, or the input is 00) our system performs the rules mentioned above; the details are left to the reader. □

## Simulation of OR Gate

**Lemma 4.2** *Boolean OR gate can be simulated by AID P systems with rules of types* $(b'_0)$ *and* $(c'_0)$*, using two membranes and two objects (only the input), in at most four steps.*

*Proof.* We construct the AID P system

$$\begin{aligned}
\Pi_{OR} &= (O, H, I, \mu, w_0, w_s, R), \text{with} \\
O &= \{0, 1\}, \\
\mu &= [\ [\ ]_0]_s, \\
w_0 &= w_s = \lambda, \\
H &= \{0, 1, s\}, \\
I &= \{r_i \mid 0 \leq i \leq 9\},
\end{aligned}$$

and the following set of $R$ of rules:

$$r_1 : [\ 1]_0 \rightarrow [\ ]_1 1$$
$$r_2 : [\ 1]_1 \rightarrow [\ ]_0 \lambda \{r_2, r_8\}$$

$r_3 : [\ 0]_0 \to [\ ]_1 0\{r_2, r_4, r_5, r_6\}$
$r_4 : [\ 0]_1 \to [\ ]_0 \lambda\{r_2, r_8\}$
$r_5 : \neg[\ 1]_1 \to [\ ]_1 1\{r_5, r_7\}$
$r_6 : \neg[\ 0]_1 \to [\ ]_1 \lambda\{r_4, r_6, r_9\}$
$r_7 : \neg 0[\ ]_1 \to [\ \lambda]_0\{r_4, r_6, r_7, r_8\}$
$r_8 : \neg[\ 1]_s \to [\ ]_s 1\{r_2, r_8\}$
$r_9 : \neg[\ 0]_s \to [\ ]_s 0\{r_2, r_5, r_9\}$

As in the case of AND gate, we place initial values $x_1$ and $x_2$ in the membrane labeled 0 from the membrane structure. The succession of rules we apply for each case is (as expected due to the duality of the system) the following:

for $x_1x_2 = 00$ – $r_3, r_6, r_9$,
for $x_1x_2 \in \{01, 10\}$ – $r_3, r_5, r_7, r_8$, or $r_1, r_4, r_8$, and
for $x_1x_2 = 11$ – $r_1, r_2, r_8$.

We only give here the details of the case when $x_1$ and $x_2$ are both 1. Our system has the following initial configuration: $[\,[\,11]_0]_s$. As mentioned above, the only rule we can apply is $r_1$, and our system evolves to the following configuration: $[\ [\ 1]_1 1]_s$. The next rule we can apply is $r_2$ through which the object in membrane 1 is transformed into $\lambda$ and the membrane label changes to 0, the system evolving to $[\ [\ ]_0 1]_s$. After applying rule $r_2$, rule $r_8$ is de-inhibited while rule $r_2$ is inhibited. We now can apply $r_8$, which sends out the skin membrane the answer 1 and restores the initial configuration of the system inhibiting rule $r_8$ and de-inhibiting rule $r_2$. We showed how our systems expels, in three steps, the right answer, given the input 11. The details of the behavior of the system in the other three cases are left to the reader. □

## Simulation of NOT Gate

**Lemma 4.3** *Boolean unary NOT gate can be simulated by AID P systems with rules of type* $(b_0)$*, in one step.*

*Proof.* We construct the AID P system

$$\begin{aligned}
\Pi_{NOT} &= (O, H, S, \mu, w_s, R), \text{with} \\
O &= \{0, 1\}, \\
\mu &= [\ ]_s, \\
w_s &= x_1x_2, \\
H &= \{s\}, \\
S &= \{r_0, r_1\}, \\
R &= \{r_0 : [\ 0]_s \to [\ ]_s 1, r_1 : [\ 1]_s \to [\ ]_s 0\}.
\end{aligned}$$

The correct computation of the NOT gate is obvious. □

### 4.2.2 Simulating Boolean Circuits

We give now an example of how to construct a global AID P system which simulates a Boolean circuit, designed to evaluate a Boolean function, using sub-AID P systems in it, namely including $\Pi_{AND}$, $\Pi_{OR}$, and $\Pi_{NOT}$ constructed in the previous section.

#### An Example

We take into consideration the example used in [17], we consider the function $f : \{0,1\}^4 \to \{0,1\}$ given by the formula $f(x_1, x_2, x_3, x_4) = (x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$. Our circuit and its assigned membrane structure is represented in Figure 4.1. As shown below, the circuit has a tree as its underlying

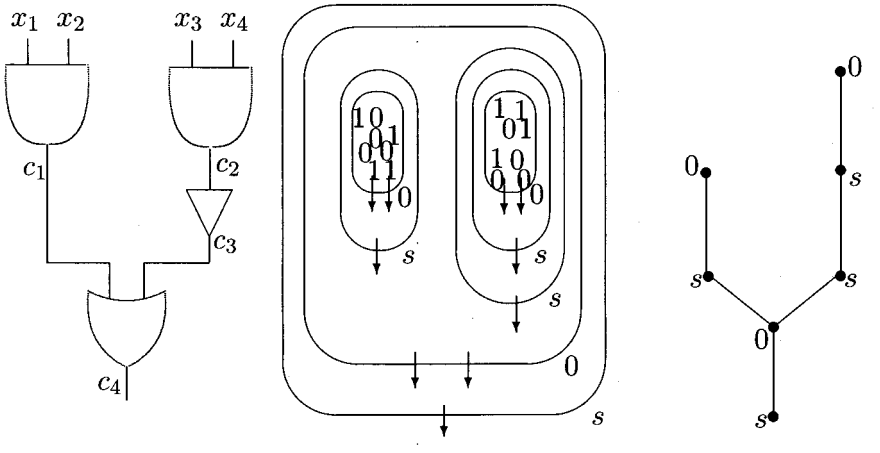



Figure 4.1: A Boolean circuit and its associated membrane structure of simulation by AID P systems.

graph, with the leaves as input gates, and the root as output gate. We simulate this circuit with the P system $\Pi_C = (\Pi^{(1)}_{AND}, \Pi^{(2)}_{AND}, \Pi^{(3)}_{NOT}, \Pi^{(4)}_{OR})$ constructed from the distributed sub-AID P systems which work in parallel in the global P system, and we obtain an unique result in the following way:

1. for every gate of the circuit with inputs from input gates, 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 a membrane the "environments" of the P systems which compute the gates at the previous level.

For the particular formula $(x_1 \wedge x_2) \vee \neg(x_3 \wedge x_4)$ and the circuit depicted in Figure 4.1 we will have:

- $\Pi^{(1)}_{AND}$ computes the first $\text{AND}_1$ gate $(x_1 \wedge x_2)$ with inputs $x_1$ and $x_2$.
- $\Pi^{(2)}_{AND}$ computes the second $\text{AND}_2$ gate $(x_3 \wedge x_4)$ with inputs $x_3$ and $x_4$; these two P systems, $\Pi^{(1)}_{AND}$ and $\Pi^{(2)}_{AND}$, act in parallel.
- $\Pi^{(3)}_{NOT}$ computes NOT gate $\neg(x_3 \wedge x_4)$ with input $(x_3 \wedge x_4)$. While $\Pi^{(3)}_{NOT}$ is working, the output value of the first $\text{AND}_1$ gate performs the rules that can be applied (in $\Pi^{(4)}_{OR}$) and at a point waits for the second input (namely, the output of $\Pi^{(3)}_{NOT}$) to come.
- after the second input enters in the inner membrane of OR gate, P system $\Pi^{(4)}_{OR}$ will be able to complete its task. The result of the computation for OR gate (which is the result of the global P system), is sent into the environment of the whole system.

The idea we want to stress here is that, as can be noticed from the above explanations, our system has a self-embedded synchronization. This means that if any gate AND or OR receives only one (part of the) input from an upper level of the tree, the gate will wait for the other part of the input to come in order to expel the output. So, an extra synchronization system is not needed in AID P systems as considered in [17].

Based on the previous explanations the following result holds:

**Theorem 4.4** *Every Boolean circuit $\alpha$, whose underlying graph structure is a rooted tree, can be simulated by a P system, $\Pi_\alpha$, in linear time. $\Pi_\alpha$ is constructed from AID P systems of type $\Pi_{AND}$, $\Pi_{OR}$ and $\Pi_{NOT}$, by reproducing the structure of the tree associated to the circuit in the architecture of the membrane structure.*

*Proof.* The result follows from the previous considerations, with the observation that the simulation lasts at most four times the duration of evaluating the Boolean circuit. □

### CIRCUIT-SAT Efficiency

There is an interesting computational problem related to circuits, called CIRCUIT-SAT. Given a circuit $C$, is there a truth assignment $T$ appropriate to $C$ such that $T(C) = true$? It is easy to argue that CIRCUIT-SAT is computationally equivalent to SAT and hence, **NP**-complete.

We can now appeal to a well-known construction (see, e.g., [74]) to reduce a CIRCUIT-SAT instance to a CNF formula. Given a circuit $C$, we will

construct a CNF formula $\varphi_C$ such that there is an assignment to the inputs of $C$ producing a 1 output if and only if the formula $\varphi_C$ is satisfiable. The formula $\varphi_C$ will have $n + |C|$ variables, where $|C|$ denotes the number of gates in $C$; if $C$ acts on inputs $x_1, \cdots, x_n$ and contains gates $g_1, \cdots, g_{|C|}$, then $\varphi_C$ will have a variable set $\{x_1, \cdots, x_n, g_1, \cdots, g_{|C|}\}$. For each gate $g \in C$, we define a set of clauses as follows:

1. if $c = \text{AND}(a, b)$, then add $(\neg c \vee a), (\neg c \vee b), (c \vee \neg a \vee \neg b)$,
2. if $c = \text{OR}(a, b)$, then add $(c \vee \neg a), (c \vee \neg b), (\neg c \vee a \vee b)$,
3. if $c = \text{NOT}(a)$, then add $(c \vee a), (\neg c \vee \neg a)$.

The formula $\varphi_C$ is simply the conjunction of all the clauses over all the gates of $C$.

We assume below that $C$ consists of gates from a standard complete basis such as AND, OR, NOT. Our results can easily be generalized to allow other gates (e.g., with a larger fan-in); the final bounds are interesting as long as the number of clauses per gate (and the maximum fan-in in the circuit) is upper bounded by a constant. Recall that a circuit $C$ is a directed acyclic graph (DAG). We define the underlying undirected graph as follows:

**Definition 4.1** *Given a circuit $C$ with inputs $X = \{x_1, \cdots, x_n\}$ and gates $S = \{g_1, \cdots, g_s\}$, let $G_C = (V, E)$ be the undirected and unweighted graph with $V = X \cup S$ and $E = \{\{x, y\} \mid x$ is an input to gate $y$ or vice versa$\}$.*

**Theorem 4.5** *For a circuit $C$ with gates from* {AND, OR, NOT}*, the* `CIRCUIT-SAT` *instance for $C$ can be solved by an AID P system.*

*Proof.* Here is the sketch of the proof.

We know that propositional formula $\varphi_C$ in CNF is simply the conjunction of all the clauses over all the gates of $C$. In our previous example, for the Boolean circuit considered in Section 4.2.2, $\varphi_C$ is:

$$\begin{aligned}\varphi_C \;=\; & (\neg c_1 \vee x_1) \wedge (\neg c_1 \vee x_2) \wedge (c_1 \vee \neg x_1 \vee \neg x_2) \wedge (\neg c_2 \vee x_3) \wedge \\ & (\neg c_2 \vee x_4) \wedge (c_2 \vee \neg x_3 \vee \neg x_4) \wedge (c_2 \vee c_3) \wedge (\neg c_2 \vee \neg c_3) \wedge \\ & (\neg c_1 \vee c_4) \wedge (\neg c_3 \vee c_4) \wedge (\neg c_4 \vee c_1 \vee c_3).\end{aligned}$$

There are already known algorithms which solve `SAT` (written as Boolean propositional formula in CNF) with P systems with active membranes (see, e.g., the previous Theorems 3.3, 3.4). Then our $\varphi_C$ can be solved following the proof ideas from the proofs of these theorems. □

### 4.2.3 Accepting and Generative Universality Results

P systems with active membranes and with particular combinations of several types of rules can reach universality without using polarizations, but only when additional features are used, for instance, the change of the labels of membranes.

We show here that P systems with active membranes using the inhibiting/ de-inhibiting mechanism are universal even without polarizations or other additional features.

The first universality result follows from the one shown in Theorems 4.1 and 4.2. There has been shown that P systems with catalytic inhibiting/de-inhibiting evolution rules are universal in the accepting and generative sense. The same proofs works also for our model by using only one membrane and catalytic developmental rules (with one catalyst).

Therefore

**Theorem 4.6** $Ps_{gen}AIDP_1(cat_1) = Ps_{acc}AIDP_1(cat_1) = PsRE.$

The next universality result is for the generative case and its proof is based on the simulation of matrix grammars with appearance checking.

**Theorem 4.7** $Ps_{gen}AIDP_4(\mathsf{rdn}) = PsRE.$

*Proof.* We construct the AID P system of degree 4

$$\begin{aligned}
\Pi &= (O, H, I, \mu, w_0, w_1, w_2, w_3, R),\\
O &= T \cup N_2 \cup \{\alpha_m \mid \alpha \in N_2 \cup T, m \in B\} \cup \{\lambda, \#\},\\
H &= \{0, 1, 2, 3\},\\
I &= \{r_{im} \mid 1 \leq i \leq 10, m \in B\} \cup \{r_{11}, r_{12} \cup \{r_a \mid a \in T\},\\
\mu &= [\,[\,[\,]_3]_2]_1]_0,\\
w_1 &= AX, w_0 = w_2 = w_3 = \lambda.
\end{aligned}$$

and the set $R$ contains the following rules.

The simulation of a matrix $m : (X \to Y, A \to x)$, with $X \in N_1, Y \in N_1 \cup \{\lambda\}$, and $A \in N_2$, $|x| \leq 2$, $x = x'x'', x', x'' \in N_2 \cup T \cup \{\lambda\}$, is done using the following inhibiting/de-inhibiting rules added to the set $R$:

$$\begin{aligned}
r_{1m} &: [\,X[\,]_2]_1 \to [\,[\,Y_m]_2]_1 \lambda \{r_{2m}\},\\
r_{2m} &: \neg[\,A[\,]_2]_1 \to [\,[\,x'_m]_2]_1 x''_m \{r_{2m}, r_{3m}, r_{5m}\},\\
&\quad |x'_m| \leq 1, |x''_m| \leq 1,\\
r_{3m} &: \neg[\,Y_m[\,]_3]_2 \to [\,[\,\lambda]_3]_2 Y \{r_{3m}, r_{4m}\},\\
r_{4m} &: \neg[\,x'_m[\,]_3]_2 \to [\,[\,\lambda]_3]_2 x' \{r_{4m}\},\\
r_{5m} &: \neg[\,x''_m[\,]_1]_0 \to [\,[\,x'']_1\,]_0 \{r_{5m}\}.
\end{aligned}$$

These rules simulate the matrices of the second and fourth type of $M$.

Initially, objects $X$ and $A$ are placed in the membrane with label 1. In the first step, by using rule $r_{1m}$ object $X$ is replicated to $Y_m$ and $\lambda$, and object $Y_m$ sent to the inner membrane.

In the same step rule $r_{2m}$ is de-inhibited so it can be executed in the next step. By using rule $r_{2m}$ the object $A$ is replicated to object $x'_m$ and object $x''_m$, $|x'_m| \leq 1$, $|x''_m| \leq 1$, and these objects are distributed into membrane 2 and membrane 0, respectively, while the rule inhibits itself; moreover, rules $r_{3m}$ and $r_{5m}$ are de-inhibited. In the next step, objects $Y$ and $x''$ are introduced in the membrane with label 1 simultaneously, while rule $r_{4m}$ is de-inhibited by rule $r_{3m}$. Then object $x'$ enters into membrane 1 from membrane 2. In this way the first rule of the matrix has been simulated.

The simulation of a matrix $m : (X \to Y, A \to \#)$, with $X, Y \in N_1$, and $A \in N_2$, is done using the rules:

$$
\begin{aligned}
r_{6m} &: [\, X[\ ]_2]_1 \to [\, [\, Y_m]_2]_1 \lambda \{r_{7m}, r_{8m}\}, \\
r_{7m} &: \neg[\, A[\ ]_2]_1 \to [\, [\, \lambda]_2]_1 \#, \\
r_{8m} &: \neg[\, Y_m[\ ]_3]_2 \to [\, [\, Y_m]_3]_2 \lambda \{r_{8m}\}, \\
r_{9m} &: [\, Y_m[\ ]_4]_3 \to [\, [\, \lambda]_4]_3 Y_m \{r_{10m}\}, \\
r_{10m} &: \neg[\, Y_m[\ ]_3]_2 \to [\, [\, \lambda]_3]_2 Y \{r_{10m} r_{7m}\}, \\
r_{11} &: [\, \#[\ ]_1]_0 \to [\, [\, \#]_1]_0 \lambda, \\
r_{12} &: [\, \#[\ ]_2]_1 \to [\, [\, \lambda]_2]_1 \#, \\
r_a &: [\, a]_0 \to [\ ]_0 a, \text{ for all } a \in T.
\end{aligned}
$$

When object $X$ is rewritten to the object $Y_m$, it enters to the inner membrane with label 2 by using the rule $r_{6m}$ and the rule $r_{7m}$ is de-inhibited. If any object $A$ appears in region 1, then the trap object $\#$ is produced and so the computation cannot halt. If the rule $r_{7m}$ is not applied, after 4 steps, the object $Y$ come in the region where object $X$ is present and the rule $r_{7m}$ will be inhibited again. Thus, also the matrix in appearance checking is simulated in the correct way and the process can be iterated.

The result of the computation is collected in the environment; from the above explanation it follows that the set of vectors generated by $\Pi$ is exactly the Parikh image of $L(G)$. □

### 4.2.4 An Efficiency Result for AID P systems

We proved in Theorem 3.15 of Chapter 3 that P systems with rules of types ndiv and rdn′, constructed in a semi-uniform manner, can deterministically solve SAT in linear time.

The next theorem shows how to solve the SAT problem by using the inhibiting/de-inhibiting mechanism without changing the labels of membranes, still in linear time.

**Theorem 4.8** *AID P systems with rules of types* ndiv *and* rdn, *constructed in a semi-uniform manner, can deterministically solve* SAT *in linear time with respect to the number of variables and the number of clauses.*

*Proof.* We construct the AID P system

$$\begin{aligned}
\Pi &= (O, H, I, \mu, w_0, \cdots, w_7, R), \text{ with} \\
O &= \{a_i \mid 1 \le i \le n\} \cup \{c_i \mid 1 \le i \le m\} \\
&\cup \{d_i \mid 1 \le i \le m\} \cup \{e_i \mid 0 \le i \le 2n+m+3\} \\
&\cup \{t_i, f_i \mid 1 \le i \le n\} \cup \{\mathtt{YES}, \mathtt{NO}\}, \\
H &= \{i \mid 0 \le i \le 6\}, \\
I &= \{g_i \mid 1 \le i \le 2n+m+2\} \cup \{h_i \mid 1 \le i \le 4\}, \\
\mu &= [\,[\,[\,[\,\,]_3]_2]_1[\,[\,[\,\,]_6]_5]_4]_0, \\
w_1 &= a_1 \cdots a_n, w_4 = e_0, w_0 = w_2 = w_3 = w_5 = w_6 = \lambda
\end{aligned}$$

The rules of P system $\Pi$ are the following:

The global control rules are:

$$\begin{aligned}
&h_1 : [\, e_i[\,\,]_5]_4 \to [\,[\, e_{i+1}]_5]_4\lambda, \\
&h_2 : [\, e_i[\,\,]_6]_5 \to [\,[\,\lambda]_6]_5 e_{i+1}, 0 \le i \le 2n+m.
\end{aligned}$$

The control variables $e_i$ counts the computing steps in nested control membranes. As we shall see, after $2n+m+2$ derivation steps, the answer YES appears outside the skin membrane if the given satisfiability problem has a solution. However, in the case that no solution exists, in one or two steps more, the answer NO appears in the environment.

*Generation phase:*

$$\begin{aligned}
&g_1 : [\, a_1]_1 \to [\, t_1]_1[\, f_1]_1\{g_2\}, \\
&g_i : \neg[\, a_i]_1 \to [\, t_i]_1[\, f_i]_1\{g_{i+1}\},\ 2 \le i \le n.
\end{aligned}$$

At the first step of the computation, using rule $g_1$ with object $a_1$, we produce the truth values *true* and $false$ assigned to the variable $x_1$, placed in two new separate copies of membrane 1. At the same time, rule $g_2$ is de-inhibited. Therefore, the previously de-inhibited rules $g_i, 2 \le i \le n$, will be executed in the next step. In this way, in $n$ steps we assign truth values to all variables, hence we get all $2^n$ truth assignments, placed in $2^n$ separate copies of membrane 1.

Objects $t_i$ corresponds to the *true* value of variable $x_i$, while object $f_i$ corresponds to the $false$ value of variable $x_i$.

In the $n$-th step, after applying the rule $g_n$, the rule $g_{n+1}$ can be executed.

$$\begin{aligned}
g_{n+i} : &\neg[\, t_i[\,\,]_2]_1 \to [\,[\, v_i]_2]_1\lambda\{g_{n+i+1}\}, \\
&[\, f_i[\,\,]_2]_1 \to [\,[\, v_i']_2]_1\lambda\{g_{n+i+1}\},\ 1 \le i \le n.
\end{aligned}$$

By using the rules $g_{n+i}$, $1 \leq i \leq n$, every object $t_i$ and $f_i$ evolves to objects $c_i$ (corresponding to clauses $C_i$, satisfied by the *true* or *false* values chosen for $x_i$) and object $\lambda$, respectively; they are also distributed into the inner and surrounding membranes.

*Checking phase:*

$$g_{2n+i} : [\ c_i[\ ]_3]_2 \rightarrow [\ [\ c_i]_3]_2 d_i \{g_{2n+i+1}, g_{2n+i}\},\ 1 \leq i \leq m.$$

In the checking phase, by using rules $g_{2n+i}$, object $c_i, 1 \leq i \leq n$, is placed in membranes labeled 2, and replicated into object $c_i$ and counter object $d_i$. Object $c_i$ is sent to the direct inner elementary membrane, and object $d_i$ is sent to the surrounding membrane. Meanwhile, the rule $g_{2n+i}$ itself is inhibited and the rule $g_{2n+i+1}$ is de-inhibited in order to check the next object. If all objects $c_i, 1 \leq i \leq m$, are present in any membrane then, object $d_m$ is produced in membranes with label 1 after $m$ steps.

*Output phase:*

$$g_{2n+m+1} : \neg[\ d_m[\ ]_2]_1 \rightarrow [\ [\ \lambda]_2]_1 d_m,$$
$$g_{2n+m+2} : [\ d_m[\ ]_1]_0 \rightarrow [\ [\ \lambda]_1]_0 \texttt{YES}\ \{g_{2n+m+2} h_4\}.$$

If $\beta$ has a solution objects $d_m$ appears in the skin membrane after $2n + m + 2$ steps, by using rules $g_{2n+m+1}$, and again, one object $d_m$, non-deterministically chosen, will output object `YES` to environment, while the rule is inhibited to avoid further output. This happens when the formula is satisfiable and the computation stops (this step was the $(2n+m+3)$th step of the entire computation).

$$h_3 : [\ e_{2n+m+1(2)}[\ ]_5]_4 \rightarrow [\ [\ \lambda]_5]_4 e_{2n+m+2(3)},$$
$$h_4 : [\ e_{2n+m+2(3)}[\ ]_4]_0 \rightarrow [\ [\ \lambda]_4]_0 \texttt{NO}.$$

If $\beta$ has no solution and if $2n + m + 1$ is an odd step, then after two more steps the counter object $e_{2n+m+3}$ will output the answer `NO` to the environment. Otherwise, after one more step object $e_{2n+m+2}$ will execute this operation. □

Concluding, a mechanism to control computations in P systems and inspired in neural-cells behavior was introduced and investigated in the P systems with inhibiting/de-inhibiting rules. The mechanism is based on the ability to inhibit and de-inhibit rules of the system during the computation.

We also have illustrated how this kind of controlling mechanism can be introduced into P systems with active membranes. Boolean gates and circuits are simulated by AID P systems. Then, we have shown that **NP**-complete problems, in particular `SAT`, can be solved in linear time by using this model. Moreover, universality (in the generative and accepting case) can be obtained by using simple inhibiting/de-inhibiting rules and without polarizations or changes of labels.

# Chapter 5

# Spiking Neural P Systems

Recently, ideas from neural computing based on spiking were incorporated in membrane computing in the form of spiking neural P systems (for short, SN P systems) in [49]. SN P systems proved [83, 84] to be Turing complete as number computing devices and also complete modulo direct and inverse morphisms in the case when they are used as language generators (see [18]).

In this chapter, we consider a couple of problems in the framework of SN P systems, namely *axon P systems* and, the *efficiency* of SN P systems.

## 5.1 Axon P Systems

In SN P systems, the main "information-processor" is the neuron, while the axon is only a channel of communication, without any other role – which is not exactly the case in neuro-biology. In the present section we introduce a class of SN-like P systems, where the computation is done along the axon (this time we ignore the neurons). Actually, a sort of linear SN P system is considered, corresponding to the Ranvier nodes of axons. Spikes are transmitted along the axon, to the left and to the right, from a node to another node, and an output is provided by the rightmost node. Specifically, a symbol $b_i$ is associated with a time unit when $i$ impulses (spikes) exit the system, and thus a string is associated with a computation.

The relationships of the families of languages generated in this way with families from Chomsky hierarchy are investigated, then axon P systems with states are considered. Several open problems and research topics are formulated.

We pass directly to considering the device we investigate, incorporating in the spiking neural P systems framework the way the axon processes information, as described in Section 1.2.

An *axon P system* of degree $m \geq 1$ is a construct of the form $\Pi = (O, \rho_1, \ldots, \rho_m)$, where:

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

2. $\rho_1, \ldots, \rho_m$ are (Ranvier) *nodes*, of the form $\rho_i = (n_i, R_i), 1 \leq i \leq m$, where:

   a) $n_i \geq 0$ is the *initial number of spikes* contained in $\rho_i$;

   b) $R_i$ is a finite set of *rules* of the form $E/a^c \to (a^l, a^r)$, where $E$ is a regular expression over $a$, $c \geq 1$, and $l, r \geq 0$, with the restriction that $R_1$ contains only rules with $l = 0$.

The intuition is that the nodes are arranged along an axon in the order $\rho_1, \ldots, \rho_m$, with $\rho_m$ at the end of the axon, hence participating to synapses (this is a way to say that $\rho_m$ is the *output* node of the system).

A rule $E/a^c \to (a^l, a^r)$ is used as follows. If the node $\rho_i$ contains $k$ spikes, and $a^k \in L(E), k \geq c$, then the rule can be applied, and this means that $c$ spikes are removed from $\rho_i$ (thus only $k - c$ remain in $\rho_i$), the node is fired, and it sends $l$ spikes to its left hand neighbor and $r$ spikes to its right hand neighbor; the first node, $\rho_1$ does not send spikes to the left, while in the case of the rightmost node, $\rho_m$, the spikes sent to the right are "lost" in the environment. The system is synchronized, a global clock is assumed, marking the time for all nodes.

If a rule $E/a^c \to (a^l, a^r)$ has $E = a^c$, then we will write it in the simplified form $a^c \to (a^l, a^r)$.

In each time unit, if a node $\rho_i$ can use one of its rules, then a rule from $R_i$ *must* be used. Since two rules $E_1/a^{c_1} \to (a^{l_1}, a^{r_1})$ and $E_2/a^{c_2} \to (a^{l_2}, a^{r_2})$, can have $L(E_1) \cap L(E_2) \neq \emptyset$, it is possible that two or more rules can be applied in a node, and in that case, only one of them is chosen non-deterministically.

During the computation, a configuration is described by the number of spikes present in each node. The initial configuration is $C_0 = \langle n_1, \ldots, n_m \rangle$.

Using the rules as described above, one can define transitions among configurations. A transition between two configurations $C_1, C_2$ is denoted by $C_1 \Longrightarrow C_2$. Any sequence of transitions starting in the initial configuration is called a *computation*. A computation halts if it reaches a configuration where no rule can be used. With any computation (halting or not) we associate a sequence of symbols by associating the symbol $b_i$ with a step when the system outputs $i$ spikes, with $b_0$ indicating the steps when no spike is emitted from $\rho_m$ to the environment. When the computation is halting, this sequence is finite.

Let us denote by $L(\Pi)$ the language of strings computed as above by halting computations of the system $\Pi$ and let $LAP_m(rule_k, cons_p)$ the family of languages $L(\Pi)$, generated by systems $\Pi$ with at most $m$ nodes, each node having at most $k$ rules, and each of these rules consuming at most $p$ spikes. As usual, a parameter $m, k, p$ is replaced with $*$ if it is not bounded.

## 5.2 Examples

We consider here some simple axon P systems, given in the graphical form, following the style of spiking neural P systems: we specify the nodes along the axon, with two way arrows among them and with an arrow which exits from the output node, pointing to the environment; in each node we give the rules and the spikes present in the initial configuration.

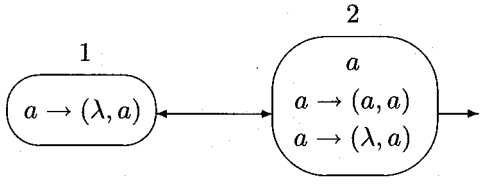



Figure 5.1: The initial configuration of system $\Pi_1$

Figure 5.1 presents the initial configuration of the system $\Pi_1$. We have two nodes, with node $\rho_2$ containing one spike. This spike will circulate among the two nodes as long as the second node uses the rule $a \to (a, a)$. In this way, in every second step one outputs a spike, starting with the first step of the computation. When node $\rho_2$ uses the rule $a \to (\lambda, a)$ no spike will remain inside and the computation halts. Therefore, $L(\Pi_1) = (b_1 b_0)^* b_1$.

Consider also the system $\Pi_2$ from Figure 5.2. This time each node starts with a spike, hence the two nodes can interchange a spike as long as $\rho_1$ uses the rule $a \to (\lambda, a)$ and $\rho_2$ uses one of the rules $a \to (a, a)$, $a \to (a, \lambda)$. If at some moment the two nodes use simultaneously the other rules, then no spike remains in the system and the computation halts. In this way, one generates all strings in $\{b_0, b_1\}^+$, hence we have $L(\Pi_2) = \{b_0, b_1\}^+$.

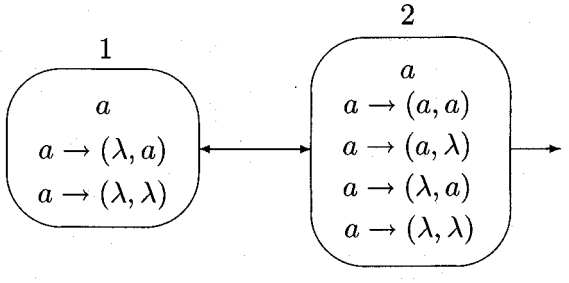



Figure 5.2: The initial configuration of system $\Pi_2$

Clearly, this system can be extended to a system which generates the language $V^+$ for any alphabet $V$ with at least two symbols.

The third example, $\Pi_3$, from Figure 5.3, is slightly more complex. It has $n + 1$ nodes, for some given $n \geq 2$. The leftmost node has only one rule,

the rightmost node contains the rules $a \to (a^2, a)$ and $a \to (\lambda, a)$, and all other nodes contain the rules $a \to (\lambda, a)$ and $a^2 \to (a^2, \lambda)$. We start with one spike in node $\rho_{n+1}$. This spike is moved continuously to the left and to the right along the axon, and always when it arrives in the rightmost node a spike exits the system. The computation can be finished by the rightmost node, by using the rule $a \to (\lambda, a)$, which leaves the system without any spike inside. Consequently, we have $L(\Pi_3) = \{b_1(b_0^{2n-1}b_1)^* \mid n \geq 1\}$.

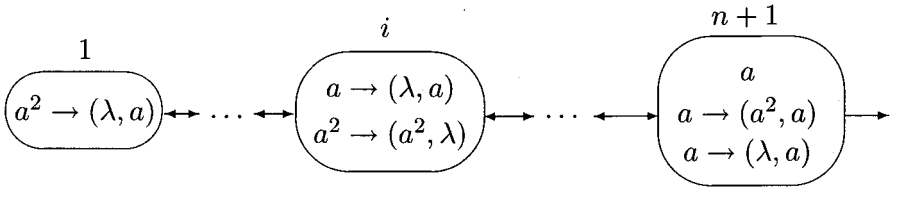



Figure 5.3: The initial configuration of system $\Pi_3$

We do not present further examples, because the results in the next section are also based on effective constructions of axon P systems.

## 5.3 The Generative Power of SN P Systems

### 5.3.1 A Characterization of FIN

**Lemma 5.1** *$LAP_1(rule_*, cons_*) \subseteq FIN$.*

*Proof.* In each step, the number of spikes present in an axon P system with only one node decreases by at least one, hence any computation lasts at most as many steps as the number of spikes present in the system at the beginning. Thus, the generated strings have a bounded length. □

**Lemma 5.2** *$FIN \subseteq LAP_1(rule_*, cons_*)$.*

*Proof.* Let $L = \{x_1, x_2, \ldots, x_n\} \subseteq V^*$, $n \geq 1$, be a finite language for some $V = \{b_1, \ldots, b_k\}$, $k \geq 1$. Let $x_i = x_{i,1} \ldots x_{i,r_i}$, $1 \leq i \leq n$. For $b \in V$, define $index(b) = i$ if $b = b_i$. Denote $\alpha_j = \sum_{i=1}^{j} |x_i|$, for all $1 \leq j \leq n$.

An axon P system that generates $L$ is shown in Figure 5.4.

Initially, only a rule $a^{\alpha_n+1}/a^{\alpha_n+1-\alpha_j} \to a^{index(x_{j,1})}$ can be used, and in this way we non-deterministically choose the string $x_j$ to generate. This rule outputs the necessary number of spikes for $x_{j,1}$. Then, because $\alpha_j$ spikes remain in the neuron, we have to continue with rules $a^{\alpha_j-t+2}/a \to a^{index(x_{j,t})}$, for $t = 2$, and then for the respective $t = 3, 4, \ldots, r_j - 1$; in this way we introduce $x_{j,t}$, for all $t = 2, 3, \ldots, r_j - 1$. In the end the rule $a^{\alpha_j-r_j+2} \to a^{index(x_{j,r_j})}$ is used, which produces $x_{j,r_j}$ and concludes the computation.

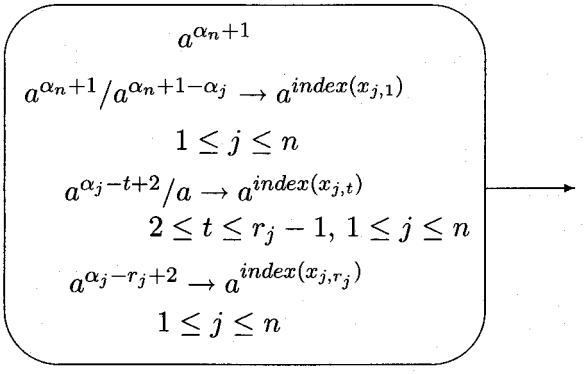



Figure 5.4: An axon P system which generates a finite language

It is easy to see that the rules which are used in the generation of a string $x_j$ cannot be used in the generation of a string $x_k$ with $k \neq j$. Also, in each rule the spikes consumed are not less than the spikes produced. □

**Theorem 5.1** $FIN = LAP_1(rule_*, cons_*)$.

### 5.3.2 Relationships with REG

**Theorem 5.2** $REG \subseteq LAP_2(rule_*, cons_*)$.

*Proof.* For $L \in REG$, consider a grammar $G = (N, B, S, P)$ such that $L = L(G)$, where $N = \{A_1, A_2, \ldots, A_n\}$, $n \geq 1$, $S = A_n$, $V = \{b_1, \ldots, b_m\}$, and the rules in $P$ are of the forms $A_i \to b_k A_j$, $A_i \to b_k$, $1 \leq i, j \leq n$, $1 \leq k \leq m$.

Then $L$ can be generated by an axon P system as shown in Figure 5.5.

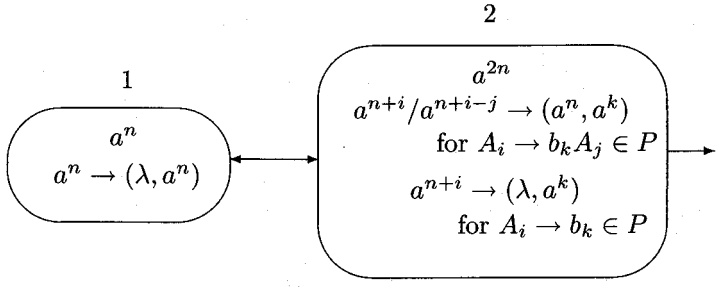



Figure 5.5: An axon P system which generates a regular language

In the first step, node $\rho_2$ fires by a rule $a^{2n}/a^{2n-j} \to (a^n, a^k)$ (or $a^{2n} \to (\lambda, a^k)$) associated with a rule $A_n \to b_k A_j$ (or $A_n \to b_k$) from $P$, and sends $k$

spikes to the environment. In this step node $\rho_1$ also fires and sends $n$ spikes to node $\rho_2$. It will send $n$ spikes back to node $\rho_2$ as long as it receives $n$ spikes from node $\rho_2$.

Assume that in some step $t$, the rule $a^{n+i}/a^{n+i-j} \to (a^n, a^k)$, for $A_i \to b_k A_j$, or $a^{n+i} \to (\lambda, a^k)$, for $A_i \to b_k$, is used, for some $1 \leq i \leq n$, and $n$ spikes are received from node $\rho_1$

If the first rule is used, then $n$ spikes are sent to node $\rho_1$, $k$ spikes are sent to the environment, $n+i-j$ spikes are consumed, and $j$ spikes remain in node $\rho_2$. Then in step $t+1$, we have $n+j$ spikes in node $\rho_2$, and a rule for $A_j \to b_k A_l$ or $A_j \to b_k$ can be used. In step $t+1$ node $\rho_2$ also receives $n$ spikes. In this way, the computation continues.

If the second rule is used, then no spike is sent to node $\rho_1$, $k$ spikes are sent to the environment, all spikes in node $\rho_2$ are consumed, and $n$ spikes are received in node $\rho_1$. Then the computation halts.

In this way, all the strings in $L$ can be generated. □

Actually, as we will see in the next section, the inclusion above is proper.

### 5.3.3 Beyond REG

**Theorem 5.3** *$LAP_m(rule_k, cons_p) - REG \neq \emptyset$ for all $m \geq 2, k \geq 3, p \geq 3$.*

*Proof.* An example of a non-regular language generated by an axon P system of the complexity mentioned in the theorem is presented in Figure 5.6. As long as the first rule of $R_1$ is used, no spike is output, and node $\rho_1$ accumulates continuously two more spikes. After $n$ steps of this type, node $\rho_2$ will contain $2(n+1)+1$ spikes. At any step, node $\rho_1$ can use the rule $a(aa)^+/a^3 \to (\lambda, a^2)$. This both leaves an even number of spikes in node $\rho_1$, and sends two spikes to node $\rho_2$. The number of spikes from node $\rho_1$ becomes $2(n+2)$ (it receives four spikes and consumes three). In the next step, we do not output any spike, but from now on we begin to output spikes in each step: node $\rho_1$ uses the rule $(aa)^+/a^2 \to (\lambda, a^3)$, thus continuously decreasing by two the number of spikes it contains. This can be done for $n+2$ steps, hence the generated string is $b_0^{n+2}b_1^{n+2}$, that is, $L(\Pi) = \{b_0^n b_1^n \mid n \geq 2\}$. This is not a regular language. □

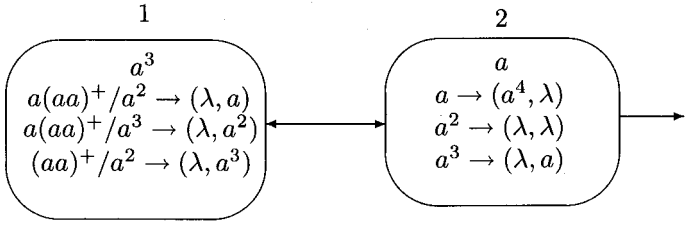



Figure 5.6: An axon P system which generates a non-regular language

Also much more complex languages can be generated:

**Theorem 5.4** *The family $LAP_2(rule_3, cons_3)$ contains non-semilinear languages.*

*Proof.* Consider the axon P system from Figure 5.7. Assume that we start from a configuration of the form $\langle 3^n + 1, 0\rangle$; initially, this is the case, with $n = 1$. As long as at least four spikes are present in node $\rho_1$, the rule $a(aaa)^+/a^3 \to (\lambda, a^9)$ is used and it moves all spikes to the second node, multiplied by 3. When we remain with only one spike in node $\rho_1$, we can use one of the other two rules of $R_1$.

If we use $a \to (\lambda, a)$, then in the second node we get a number of spikes of the form $3m + 1$, hence the first rule is applied as much as possible, thus returning the spikes to node $\rho_1$. In the end, we have to use the rule $a \to (a, \lambda) \in R_2$, which makes again the number of spikes from node $\rho_1$ to be of the form $3m + 1$ (note that no rule can be applied in any node when the number of spikes is multiple of 3). This process can be iterated any number of times, thus multiplying by 3 the number of spikes present in node $\rho_1$.

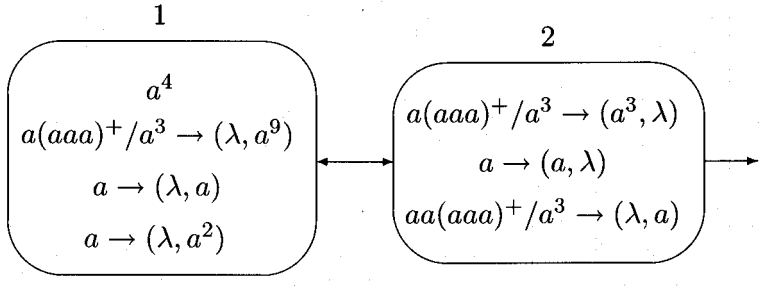



Figure 5.7: An axon P system which generates a non-semilinear language

At any time, node $\rho_1$ can also use the rule $a \to (\lambda, a^2)$ instead of $a \to (\lambda, a)$. This makes the number of spikes from node $\rho_2$ to be of the form $3m + 2$, hence the rule $aa(aaa)^+/a^3 \to (\lambda, a)$ should be applied. This rule does not change the 3-arity of the number of spikes, hence it is used as much as possible. In this way, a spike exits the system in each step, until exhausting the spikes from the output node (when only two spikes remain inside, no rule can be used).

This means that after a number of steps when no spike is sent to the environment, we output spikes for $3^n$ steps, for some $n \geq 1$. Therefore, the language $L(\Pi)$ consists of strings of the form $b_0^m b_1^{3^n}$, for some $m, n \geq 1$, and for all $n \geq 1$ there is such a string in $L(\Pi)$. Thus, $L(\Pi)$ is not a semilinear language. □

The previous language is not in $MAT$. However, as we will immediately see, this kind of systems has strong limitations.

**Lemma 5.3** *The number of configurations reachable after $n$ steps by an axon P system of degree $m$ is bounded by a polynomial $g(n)$ of degree $m$.*

*Proof.* Let us consider an axon P system $\Pi = (O, \rho_1, \ldots, \rho_m)$ of degree $m$, let $n_0$ be the total number of spikes present in the initial configuration of $\Pi$, and denote $\alpha = \max\{l + r \mid E/a^c \to (a^l, a^r) \in R_i, 1 \leq i \leq m\}$ (the maximal number of spikes produced by any of the rules of $\Pi$). In each step of a computation, each node $\rho_i$ produces some $l$ and $r$ spikes to be sent to the left and right nodes of node $\rho_i$, respectively. We have $l + r \leq \alpha$. Each node can do the same, hence the maximal number of spikes produced in one step is at most $\alpha m$. In $n$ consecutive steps, this means at most $\alpha mn$ spikes. Adding the initial $n_0$ spikes, this means that after any computation of $n$ steps we have at most $n_0 + \alpha mn$ spikes in $\Pi$, hence the number of configurations are no more than $(n_0 + \alpha mn)^m$. This is a polynomial of degree $m$ in $n$ ($\alpha$ is a constant) which bounds from above the number of possible configurations obtained after computations of length $n$ in $\Pi$. □

**Theorem 5.5** *If $f : V^+ \longrightarrow V^+$ is an injective function, $card(V) \geq 2$, then there is no axon P system $\Pi$ such that $L_f(V) = \{x\, f(x) \mid x \in V^+\} = L(\Pi)$.*

*Proof.* Assume that there is an axon P system $\Pi$ of degree $m$ such that $L(\Pi) = L_f(V)$ for some $f$ and $V$ as in the statement of the theorem. According to the previous lemma, there are only polynomially many configurations of $\Pi$ which can be reached after $n$ steps. However, there are $card(V)^n \geq 2^n$ strings of length $n$ in $V^+$. Therefore, for large enough $n$ there are two strings $w_1, w_2 \in V^+, w_1 \neq w_2$, such that after $n$ steps the system $\Pi$ reaches the same configuration when generating the strings $w_1\, f(w_1)$ and $w_2\, f(w_2)$, hence after step $n$ the system can continue any of the two computations. This means that also the strings $w_1\, f(w_2)$ and $w_2\, f(w_1)$ are in $L(\Pi)$. Due to the injectivity of $f$ and the definition of $L_f(V)$ such strings are not in $L_f(V)$, hence the equality $L_f(V) = L(\Pi)$ is contradictory. □

**Corollary 5.1** *The following languages are not in $LAP_*(rule_*, cons_*)$ (in all cases, $card(V) = k \geq 2$):*

$$L_1 = \{x\, mi(x) \mid x \in V^+\},$$
$$L_2 = \{xx \mid x \in V^+\},$$
$$L_3 = \{x\, c^{val_k(x)} \mid x \in V^+\}, c \notin V.$$

Note that language $L_1$ above is a non-regular minimal linear one (generated by linear grammars with only one nonterminal symbol), $L_2$ is context-sensitive non-context-free, and $L_3$ is non-semilinear.

**Theorem 5.6** $LAP_*(rule_*, cons_*) \subseteq REC$.

*Proof.* This is a direct consequence of the fact that a string of length $n$ is produced by means of a computation of length $n$; thus, given an axon P system $\Pi$ and a string $x$, in order to check whether or not $x \in L(\Pi)$ it is enough to produce all computations of length $|x|$ in $\Pi$ and to check whether any of them generates the string $x$. □

## 5.4 Axon P Systems with States

In this section, we consider a way to control the rule applications by means of node *states*. Specifically, the rules we are going to use are of the form $sE/a^c \rightarrow (s', a^l, a^r)$, where $s, s'$ are states, $E$ is regular expression, and $c \geq 1, l \geq 0, r \geq 0$.

Formally, an axon P system of degree $m \geq 1$ with states is a construct of the form

$$\Pi = (O, Q, \rho_1, \ldots, \rho_m),$$

where:

1. $O = \{a\}$ is the singleton alphabet;
2. $Q$ is the finite set of states;
3. $\rho_1, \ldots, \rho_m$ are (Ranvier) *nodes*, of the form

$$\rho_i = (s_0, n_i, R_i), 1 \leq i \leq m,$$

where:

a) $s_0 \in Q$ is the *initial state* of node;
b) $n_i \geq 0$, is the initial number of spikes in node;
c) $R_i$ is a finite set of *rules* of the form $sE/a^c \rightarrow (s', a^l, a^r)$, where $s$ is the current state of the node, $E$ is a regular expression over $a$, $c \geq 1$, and $l, r \geq 0$, with the restriction that $R_1$ contains only rules with $l = 0$.

A rule $sE/a^c \rightarrow (s', a^l, a^r)$ from $R_i$ is applied like the rule $E/a^c \rightarrow (a^l, a^r)$, but only if node $\rho_i$ is in state $s$; after the use of the rule, the state of $\rho_i$ becomes $s'$.

Let us examine some examples, in order to clarify the definitions and to illustrate the way our devices work.

Consider first the simple system

$$\Pi_1 = (O, Q, \rho_1, \rho_2),$$

where:

1. $O = \{a\}$;

2. $Q = \{s_0, s_1\}$;

3. $\rho_1 = (s_0, 1, R_1)$, $\rho_2 = (s_0, 1, R_2)$;
   $R_1 = \{r_1 : s_0 a^+/a \to (s_0, \lambda, a)\}$;
   $R_2 = \{r_1 : s_0 a^+/a \to (s_0, a^2, \lambda),$
   $r_2 : s_0 a \to (s_1, \lambda, \lambda),$
   $r_3 : s_1 a \to (s_1, \lambda, a)\}$.

It generates the non-regular context-free language

$$L(\Pi_1) = \{b_0^n b_1^n \mid n \geq 1\}.$$

The computation steps are presented in Table 5.1.

| step | $a$ | $a$ | | step | $a$ | $a$ | |
|---|---|---|---|---|---|---|---|
| 1 | $r_1 : a$ | $r_2 : a$ | $b_0$ | 1 | $r_1 : a$ | $r_1 : a$ | $b_0$ |
| 2 | – | $r_3 : a$ | $b_1$ | 2 | $r_1 : a^2$ | $r_1 : a$ | $b_0$ |
| | | | | 3 | $r_1 : a$ | $r_2 : a$ | $b_1$ |
| | | | | 4 | – | $r_3 : a$ | $b_1$ |

| step | $a$ | $a$ | | step | $a$ | $a$ | |
|---|---|---|---|---|---|---|---|
| 1 | $r_1 : a$ | $r_1 : a$ | $b_0$ | 1 | $r_1 : a$ | $r_1 : a$ | $b_0$ |
| 2 | $r_1 : a^2$ | $r_1 : a$ | $b_0$ | 2 | $r_1 : a^2$ | $r_1 : a$ | $b_0$ |
| 3 | $r_1 : a^3$ | $r_1 : a$ | $b_0$ | 3 | $r_1 : a^3$ | $r_1 : a$ | $b_0$ |
| 4 | $r_1 : a^4$ | $r_1 : a$ | $b_0$ | 4 | $r_1 : a^4$ | $r_1 : a$ | $b_0$ |
| 5 | $r_1 : a^5$ | $r_1 : a$ | $b_0$ | . | . . . | . . . | . |
| 6 | $r_1 : a^6$ | $r_2 : a$ | $b_0$ | $n-1$ | $r_1 : a^{n-1}$ | $r_1 : a$ | $b_0$ |
| 7 | $r_1 : a^5$ | $r_3 : a$ | $b_1$ | $n$ | $r_1 : a^n$ | $r_2 : a$ | $b_0$ |
| 8 | $r_1 : a^4$ | $r_3 : a$ | $b_1$ | $n+1$ | $r_1 : a^{n+1}$ | $r_3 : a$ | $b_1$ |
| 9 | $r_1 : a^3$ | $r_3 : a$ | $b_1$ | $n+2$ | $r_1 : a^n$ | $r_3 : a$ | $b_1$ |
| 10 | $r_1 : a^2$ | $r_3 : a$ | $b_1$ | . | . . . | . . . | . |
| 11 | $r_1 : a$ | $r_3 : a$ | $b_1$ | $2n-1$ | $r_1 : a$ | $r_3 : a$ | $b_1$ |
| 12 | – | $r_3 : a$ | $b_1$ | $2n$ | – | $r_3 : a$ | $b_1$ |

Table 5.1: Some computations in $\Pi_1$

Let us observe in Table 5.1 how the system works. Initially each node contains one spike ($a$), and they are in the initial state $s_0$. Node $\rho_1$ has only one rule and it never changes its state. In turn, node $\rho_2$ has 2 states and 3 rules; rules $r_1$ and $r_2$ apply in state $s_0$, but once rule $r_2$ is chosen the state of the node is changed to $s_1$, and from now on rule $r_3$ should apply until the computation halts.

It is easy to see that the system generates strings of the form $b_0^n b_1^n$ in $2n$ steps, for some $n \geq 1$. Let us follow the table. We apply rule $r_1$ in

node $\rho_2$ for a number of times, thus generating symbols $b_0$ (no spike is sent to the environment); in the meantime, node $\rho_1$ accumulates continuously spikes. Once rule $r_2$ is chosen (suppose this happens in the $n$-th step), the node changes its state to $s_1$ and starts to send the spikes of node $\rho_1$ to the environment, using the rule $r_3$.

Let us now consider a system with a more sophisticated functioning, namely

$$\Pi_2 = (O, Q, \rho_1, \rho_2),$$

where:

1. $O = \{a\}$;
2. $Q = \{s_0, s_1, s_1', s_2, s_2', s_3, s_\#\}$;
3. $\rho_1 = (s_0, 0, R_1)$, $\rho_2 = (s_0, 2, R_2)$;

$$\begin{aligned}
R_1 &= \{r_1 : s_0 a \to (s_0, \lambda, a^2)\};\\
R_2 &= \{r_1 : s_0 a^2 \to (s_\#, \lambda, \lambda),\\
&\quad r_2 : s_0 a^+/a \to (s_3, \lambda, a),\\
&\quad r_3 : s_0 a^+/a \to (s_3, \lambda, a^2),\\
&\quad r_4 : s_0 a^+/a \to (s_2, a, a^2),\\
&\quad r_5 : s_0 a^+/a \to (s_1, a, a),\\
&\quad r_6 : s_1 a^+/a \to (s_1, a, a),\\
&\quad r_7 : s_1 a^+/a \to (s_2, a, a^2),\\
&\quad r_8 : s_1 a^+/a \to (s_1', \lambda, a^3),\\
&\quad r_9 : s_1' a^+/a^3 \to (s_3, \lambda, a^3),\\
&\quad r_{10} : s_2 a^+/a \to (s_2, a, a^2),\\
&\quad r_{11} : s_2 a^+/a \to (s_2', \lambda, a^3),\\
&\quad r_{12} : s_2' a^+/a^3 \to (s_3, \lambda, a^3),\\
&\quad r_{13} : s_3 a^+/a \to (s_3, \lambda, a^3)\}.
\end{aligned}$$

The language generated is

$$L(\Pi_2) = \{b_1^n b_2^m b_3^{n+m} \mid n, m \geq 0\},$$

which is in $CF$. Initially, there is no spike in node $\rho_1$, but there are two spikes in node $\rho_2$. The main work of controlling the computation is done by node $\rho_2$, while node $\rho_1$ helps to grow spikes in node $\rho_2$. There are 5 rules in state $s_0$ in node $\rho_2$ and one of them is non-deterministically chosen when computation starts. Rule $r_1$ is for word $\lambda$ ($n = m = 0$), rule $r_2$ is for $b_1 b_3$ ($n = 1, m = 0$), rule $r_3$ is for $b_2 b_3$ ($n = 0, m = 1$), and rule $r_4$ is for

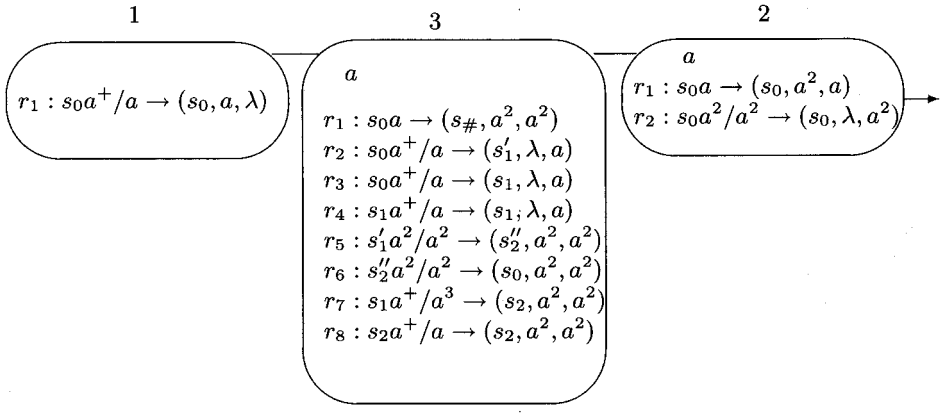



Figure 5.8: An axon P system generating a non-context-free language

words $b_2^m b_3^m$ $(n = 0, m \geq 1)$. Words of the form $b_1^n b_2^m b_3^{n+m}$ $(n \geq 1, m \geq 0)$ can be generated if the computation starts by rule $r_5$. We omit the detailed explanation here.

The last example we present is $\Pi_3 = (O, Q, \rho_1, \rho_2, \rho_3)$, given in a graphical form in Figure 5.8; it generates the non-context-free context-sensitive language $\{b_1^n b_2^n b_0^n \mid n \geq 1\}$.

| step | | $a$ | $a$ | | step | | $a$ | $a$ | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | – | $r_1 : a$ | $r_1 : a$ | $b_1$ | 1 | – | $r_3 : a$ | $r_1 : a$ | $b_1$ |
| 2 | $r_1 : a^2$ | $a^2$ | $r_2 : a^2$ | $b_2$ | 2 | – | $r_4 : a^2$ | $r_1 : a$ | $b_1$ |
| 3 | $r_1 : a$ | – | – | $b_0$ | 3 | – | $r_4 : a^3$ | $r_1 : a$ | $b_1$ |
| | | | | | 4 | – | $r_4 : a^4$ | $r_1 : a$ | $b_1$ |
| | | | | | 5 | – | $r_7 : a^5$ | $r_1 : a$ | $b_1$ |
| | | | | | 6 | $r_1 : a^2$ | $r_8 : a^4$ | $r_2 : a^2$ | $b_2$ |
| | | $a$ | $a$ | | 7 | $r_1 : a^3$ | $r_8 : a^3$ | $r_2 : a^2$ | $b_2$ |
| 1 | – | $r_2 : a$ | $r_1 : a$ | $b_1$ | 8 | $r_1 : a^4$ | $r_8 : a^2$ | $r_2 : a^2$ | $b_2$ |
| 2 | – | $r_5 : a^2$ | $r_1 : a$ | $b_1$ | 9 | $r_1 : a^5$ | $r_8 : a$ | $r_2 : a^2$ | $b_2$ |
| 3 | $r_1 : a^2$ | $r_6 : a^2$ | $r_2 : a^2$ | $b_2$ | 10 | $r_1 : a^6$ | – | $r_2 : a^2$ | $b_2$ |
| 4 | $r_1 : a^3$ | – | $r_2 : a^2$ | $b_2$ | 11 | $r_1 : a^5$ | – | – | $b_0$ |
| 5 | $r_1 : a^2$ | – | – | $b_0$ | 12 | $r_1 : a^4$ | – | – | $b_0$ |
| 6 | $r_1 : a$ | – | – | $b_0$ | 13 | $r_1 : a^3$ | – | – | $b_0$ |
| | | | | | 14 | $r_1 : a^2$ | – | – | $b_0$ |
| | | | | | 15 | $r_1 : a$ | – | – | $b_0$ |

Table 5.2: Some computations in $\Pi_3$

The string $b_1^n b_2^n b_0^n, n \geq 1$, is generated in $3n$ steps. Initially, node $\rho_1$ contains no spike, while each node $\rho_2$ and $\rho_3$ contains only one spike. The computation steps are mainly controlled by node $\rho_2$. There are three rules in node $\rho_3$ $(r_1, r_2, r_3)$ which can be applied in the first step of the computation.

If computation starts applying rule $r_1$ ($r_2$, or $r_3$), a word $b_1b_2b_0$ (resp. $b_1^2b_2^2b_0^2$ or $b_1^nb_2^nb_0^n$) will be generated. The reader can easily trace the computation steps in Table 5.2.

### Further Remarks

Many open problems and research topics about axon P systems remain to be considered. We only mention here some of them.

Actually, many questions are suggested by the research about SN P systems. For instance, in the systems considered here we have no delay associated with the rules, the spikes are emitted immediately after firing the rule – otherwise stated, the delay is always 0. However, an arbitrary delay can be considered, as usual in SN P systems. Is this of any help? What about considering infinite sequences generated by axon P systems, as investigated in [84] for SN P systems? Can any interesting class of languages or of infinite sequences be characterized/represented in this framework?

Are the hierarchies on the number of nodes infinite? The universality implies the fact that the hierarchies on the number of nodes collapse, but, in view of Theorem 5.6, our systems are not universal. Another problem related to the non-universality result is to find decidable properties other than the membership one.

What about associating a language in the following way: for each node $i$ we consider a symbol $c_i$ and a configuration $\langle k_1, \ldots, k_m \rangle$ is described by the string $c_1^{k_1} \ldots c_m^{k_m}$. We obtain a language (strictly bounded). Variant: to take only the strings which describe configurations which send out a spike (thus we have a selection of strings). Any relation with L systems?

## 5.5 Spiking Neural P Systems with Self-Activation

In this section we show that if some precomputed resources (namely, the neurons disposed in a particular – initially inactive – structure) are considered, SN P systems having a self-activation behavior prove to be also computationally efficient, and can solve SAT in constant time.

The basic idea, initially mentioned in [81], is that, instead of producing in linear time an exponential workspace, we start from the beginning with an exponentially large precomputed workspace in the form of an exponentially large number of inactive neurons, which will be activated and used in constant time in our computation.

We assume a structure of neurons given "for free" as a result of a precomputation whose duration does not matter – although the structure is not completely independent from the problem we want to solve. The neurons in the structure are initially inactive and become active immediately when a spike enters them.

This strategy corresponds to the well-known fact that the human brain contains a huge number of neurons out of which only a small part are currently used, and (some of) the inactive neurons can change their status as soon as they receive (electrical) signals from active neurons.

We consider a neuron to be *inactive* if it contains no spike (hence no rule can be applied in it). As soon as a spike enters a neuron (as input in the initial configuration or from another neuron through a synapse), it makes it *active* altogether with the synapses that it establishes with other neurons.

In a self-activating SN P system we have an arbitrarily large number of neurons which differ by the number of spikes and/or of rules they contain. Some of these neurons will be active, the others will be inactive.

In what follows, we construct an SN P system for solving SAT problem in constant time. Let us consider $n$ variables $x_1, x_2, \ldots, x_n, n \geq 1$, and a propositional formula with $m$ clauses, $\gamma = C_1 \wedge \cdots \wedge C_m$, such that each clause $C_i, 1 \leq i \leq m$, is of the form $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}, k_i \geq 1$, where $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}$.

The set of all instances of SAT with $n$ variables and $m$ clauses is denoted by $SAT(\langle n, m \rangle)$.

The instance $\gamma$ is encoded as a set over

$$X = \{x_{i,j}, x'_{i,j} \mid 1 \leq i \leq m, 1 \leq j \leq n\},$$

where $x_{i,j}$ represents variable $x_j$ appearing in clause $C_i$ without negation, while $x'_{i,j}$ represents variable $x_j$ appearing in clause $C_i$ with negation.

Now, we give an informal description of the precomputed resource structure devoted to $SAT(\langle n, m \rangle)$. Figure 5.9 depicts an SN P system working in self-activating manner using precomputed resources, where the nodes and the arrows represent the neurons and the synapses, respectively. One can notice that the nodes have (four) different shapes (◯, ◎, □, ▷), but this is just a way to make the construction easier to understand (the shape does not imply any differences in the behavior of the nodes). Also, we see that the structure has a sort of symmetry. Namely, for each clause we have a block of ◯-neurons and ◎-neurons.

*The Device Structure:* The precomputed device (initially inactive) able to deal with any $\gamma \in SAT(\langle n, m \rangle)$ is formed by $2^n(m+1) + 2nm + 1$ neurons and $2^n(3m+1)$ synapses. Further on we are giving some details on the components of this structure.

*Neurons of type* $\bigcirc_{c_i x_j 1/0}$: For each variable $x_j$ of a clause $C_i$, we associate 2 neurons $\bigcirc_{c_i x_j 1}$ and $\bigcirc_{c_i x_j 0}$, $1 \leq i \leq m, 1 \leq j \leq n$. Obviously, the subscript of the neurons indicates the clause ($c_i$ for the clause $C_i$), and the variable ($x_j$ for the $j$th variable in the clause). 1 and 0 are used to mark differently the two neurons needed to encode the same variable; their use will be detailed further on in the paper where the encoding part will be explained. However,

each clause is described by $2n$ neurons, and there are exactly $2nm$ neurons of type $\bigcirc_{c_i x_j 1/0}$ associated with $m$ clauses.

*Neurons of type* $\circledcirc_{c_i bin}$: There are $2^n$ $\circledcirc_{c_i bin}$ neurons, associated to each clause $C_i$, injectively labeled with elements of $\{c_i bin \mid bin \in \{1,0\}^n\}$. They correspond to the $2^n$ truth assignments for variables variables $x_1, \ldots, x_n$. In total, the device has $2^n m$ $\circledcirc_{c_i bin}$ neurons ($2^n$ for each of the $m$ clauses). Later on, we will see that these neurons will handle, during the computation, Boolean operation $\vee$ (OR) present in the clauses of the formula.

*Synapses* $\bigcirc_{c_i x_j 1/0} \longrightarrow \circledcirc_{c_i bin}$: The connections are in one direction from $\bigcirc_{c_i x_j 1/0}$ to $\circledcirc_{c_i bin}$. The synapses are designed in such a way that the two neurons linked by a connection have the same prefix of the labels ($c_i$), and the last symbol of label $c_i x_j 0/1$ is the same with the $j$th symbol (0 or 1) of the string $bin$. Each $\circledcirc_{c_i bin}$ neuron is connected to $n$ $\bigcirc_{c_i x_j 1/0}$ neurons.

*Neurons of type* $\square_{bin}$: There are exactly $2^n$ $\square_{bin}$ neurons in the device labeled injectively with strings from $bin \in \{0,1\}^n$. These neurons are designed to handle the Boolean operation $\wedge$ (AND) between the clauses of the formula.

*Synapses* $\circledcirc_{c_i bin} \longrightarrow \square_{bin}$: Each $\square_{bin}$ neuron is connected to one $\circledcirc_{c_i bin}$ neuron from each clause block, hence, $m$ double circled neurons may send spikes to each square neuron. Strings $bin$ from the labels of the connected $\circledcirc_{c_i bin}$ and $\square_{bin}$ neurons are the same.

*Neuron of type* $\triangleright_4$: Finally, there is a unique output neuron $\triangleright$ with label 4. By choosing label 4 for this neuron we emphasize that it spikes in the 4th step of the computation if the problem has at least a solution and does not spike in this step, otherwise. All $\square_{bin}$ neurons are connected to the output neuron, hence, there are $2^n$ connections of type $\square_{bin} \longrightarrow \triangleright_4$.

*Rules*: Here are the rules which apply to each type of neurons:

$$\begin{aligned} R_{\circ} &= \{a \to \lambda, a^2 \to a; 0, a^3 \to \lambda, a^4 \to a; 0\} \\ R_{\circledcirc} &= \{a^+/a \to a; 0\} \\ R_{\square} &= \{a^m \to a; 0\} \\ R_{\triangleright} &= \{a^+/a \to a; 0\} \end{aligned}$$

We have described the precomputed device structure for solving SAT problem as depicted in Figure 5.9, with the neurons in the inactive mode. Note that this structure is independent of the instance of SAT we want to solve, and it depends only on $n$ and $m$. Let us explain now how we encode the particular instance of the problem into the device.

*The Problem Encoding*: The variables are encoded by spikes as follows: one assigns values 1 and 0 to each variable $x_j$ and $\neg x_j$. Further, a variable $x_j$ is encoded by two spikes ($a^2$), and one spike ($a$) if we assign to $x_j$ values 1 and 0, respectively. Similarly, we use $a^3$ and $a^4$ to encode variable $\neg x_j$, which has assigned values 1 and 0, respectively.

In Table 5.3 one can notice how we introduce the encoded variables (the spikes) into the precomputed device. The (encoded) variables $x_j$ or $\neg x_j$, from a clause $C_i$, assigned with value 1 are introduced in the neuron with label $c_i x_j 1$ ($\bigcirc_{c_i x_j 1}$), while the other "half" of the encoding, the one where variable is assigned with value 0 is introduced in the neuron labeled $c_i x_j 0$, (hence $\bigcirc_{c_i x_j 0}$). Some of the $\bigcirc_{c_i x_j 1/0}$ neurons will not be activated in the case the corresponding variables are missing form the given instance of the problem. Anyway, we stress the fact that at most $2nm$ $\bigcirc_{c_i x_j 1/0}$ neurons are activated in when the device is initialized, the other neurons in the system remaining inactive.

*The Computation*: Starting this moment (when the device is initialized and the corresponding neurons activated), the computation can be performed and the problems will be solved in 4 steps. Once the system starts to evolve the spikes follow the one-directional path:

$$\bigcirc\text{-neurons} \longrightarrow \circledcirc\text{-neurons} \longrightarrow \square\text{-neurons} \longrightarrow \triangleright\text{-neuron},$$

as shown in Figure 5.10.

## An Example

In order to illustrate the procedure discussed above, let us examine a simple example.

We consider the following instance of SAT, with two variables and three clauses, $\gamma \in SAT(\langle 2, 3 \rangle)$,

$$\gamma = (x_1) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_2).$$

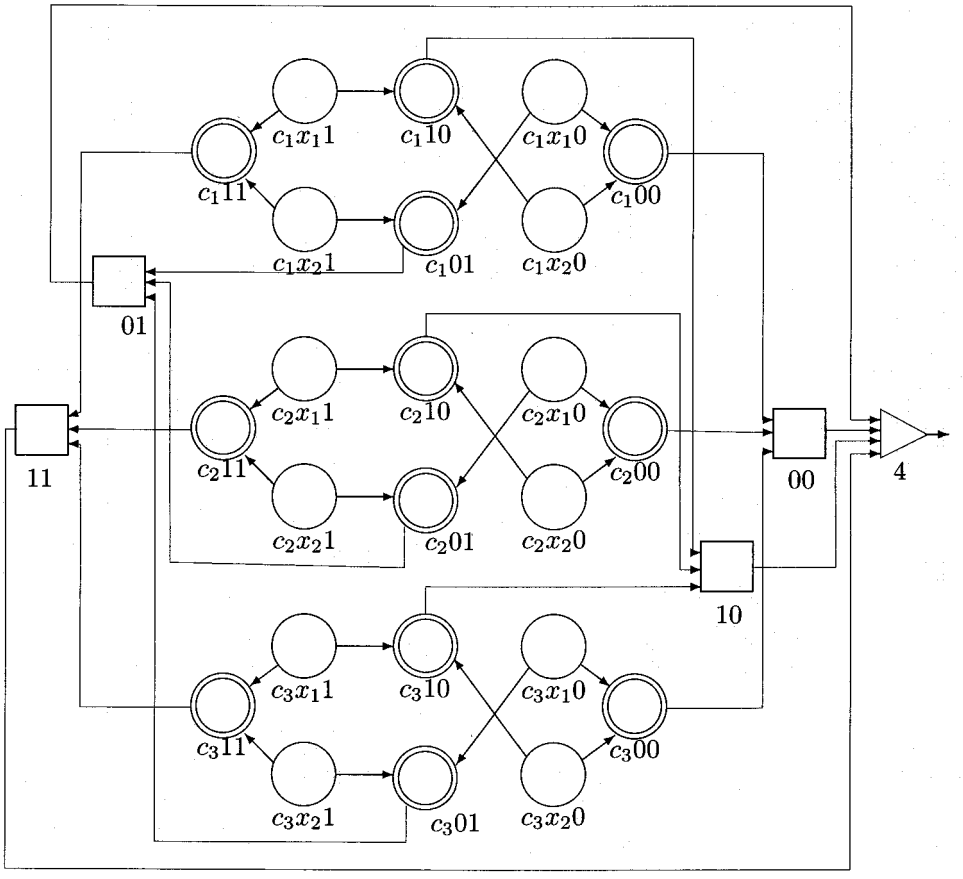
*The device structure*: The system we construct is given in a pictorial way in Figure 5.9, and has 29 inactive neurons ($4*2^2+2*2*3+1$) and 40 synapses. There are 3 blocks of neurons, each dealing with a clause of the problem. Each block contains 4 solid circled ($\bigcirc_{c_i x_1 1}$, $\bigcirc_{c_i x_1 0}$, $\bigcirc_{c_i x_2 1}$, $\bigcirc_{c_i x_2 0}$) neurons – 2 for each variable – to which we assign 1 and 0 (see the labels). In each block, there are also 4 double circled neurons ($\circledcirc_{c_i 11}, \circledcirc_{c_i 10}, \circledcirc_{c_i 01}, \circledcirc_{c_i 00}$) connected to the $\bigcirc_{c_i x_j 1/0}$ neurons, corresponding to the $2^2$ truth assignments (see the labels). Moreover, we have 4 $\square_{bin}$ neurons connected to the corresponding $\circledcirc_{c_i bin}$ neurons.

*Encoding*: According to the formula $\gamma$, in the first clause $C_1$, there is only one variable $x_1$. We assign values 1 and 0 to $x_1$ and encode it by two spikes ($a^2$) and one spike ($a^1$) that are placed in $\bigcirc_{c_1 x_1 1}$ and $\bigcirc_{c_1 x_1 0}$, respectively. The other two neurons $\bigcirc_{c_1 x_2 1}$ and $\bigcirc_{c_1 x_2 0}$ corresponding to variable $x_2$ remain empty, since there is no variable $x_2$ or $\neg x_2$ in the clause.

The second clause ($C_2$) is encoded in the following clause block. To each variable $\neg x_1$ and $x_2$ are assigned values 1 and 0. Further, they are encoded by $a^3$ in $\bigcirc_{c_2 x_1 1}$, $a^4$ in $\bigcirc_{c_2 x_1 0}$, for $\neg x_1$, and $a^2$ in $\bigcirc_{c_2 x_2 1}$, $a^1$ in $\bigcirc_{c_2 x_2 0}$, for $x_2$.

| | 1 | 0 |
|---|---|---|
| $x_{i,j}$ | $(a^2, \bigcirc_{c_i x_j 1})$ | $(a, \bigcirc_{c_i x_j 0})$ |
| $\neg x_{i,j}$ | $(a^3, \bigcirc_{c_i x_j 1})$ | $(a^4, \bigcirc_{c_i x_j 0})$ |

Table 5.3: The variable encoding.




Rules used in $\Pi_{SAT(\langle 2,3\rangle)}$ :

$$R_\circ = \{a \to \lambda, a^2 \to a; 0, a^3 \to \lambda, a^4 \to a; 0\}$$
$$R_\circledcirc = \{a^+/a \to a; 0\}$$
$$R_\square = \{a^3 \to a; 0\}$$
$$R_\triangleright = \{a^+/a \to a; 0\}$$

Figure 5.9: Precomputed spiking neural net for $\texttt{SAT}(\langle 2, 3\rangle)$

The last clause $C_3$ has only one variable, $\neg x_2$, which is encoded in the neurons $\bigcirc_{c_3x_21}$ and $\bigcirc_{c_3x_20}$. The other two neurons corresponding to this clause remain empty. See Step 1 of Figure 5.10.

*Computation*: Once the encoding is done, single circled neurons are activated and send signals, or erase spikes according to the rules they contain. In the first step, neurons having inside $a^2$ and $a^4$ fire, while spikes $a^1$ and $a^3$ from the other neurons are deleted. We see in Step 2 from Figure 2, that only 7 double circled neurons out of 12 contain spikes, hence only 7 will be activated in the second step. Then, in next step of computation, double circled neurons containing spikes fire because of the rules $a^+/a \to a$ inside. One can notice that neurons $\square_{11}$, $\square_{00}$, and $\square_{10}$ have received two spikes, and neuron $\square_{01}$ has received only one spike. In the third step of the computation, only neuron $\circledcirc_{c_201}$ fires since there was one spike remained, and nothing else happens in the system. The rule present inside the square neurons, $a^3/a \to a$, cannot be applied, because there is no such neuron containing three spikes. At the fourth step, the output neuron does not spike, since no spike has arrived here in the third step of the computation, hence the given problem has no solution.

We have mentioned before that what happens after the fourth step of the computation is out of our interest because it does not give further details with respect to the satisfiability of the given problem. We just want to mention that the system may not stop after giving the answer to our problem.

### 5.5.1 A solution to SAT

Formally, for a given $(n, m) \in \mathbb{N}^2$, an SN P system using precomputed resources, working in a self-activating manner, devoted to solve **SAT** problem with $n$ variables and $m$ clauses, is a construct

$$\Pi^{n,m}_{\mathbf{SAT}} = (\Pi_{SAT(\langle n,m\rangle)}, \Sigma(\langle n, m\rangle))$$

with:

- $\Pi_{SAT(\langle n,m\rangle)} = (O, \mu, \triangleright_4)$, where:
  1. $O = \{a\}$ is the singleton alphabet;
  2. $\mu = (H, \Omega, R_H, syn)$ is the precomputed device structure, where:
     - $H = H_1 \cup H_2 \cup H_3 \cup H_4$ is a finite set of neuron labels, where
       $H_1 = \{c_ix_j1, c_ix_j0 \mid 1 \leq i \leq m, 1 \leq j \leq n\}$,
       $H_2 = \{c_ibin \mid 1 \leq i \leq m, bin \in \{0,1\}^n\}$,
       $H_3 = \{bin \mid bin \in \{0,1\}^n\}$,
       $H_4 = \{4\}$;
     - $\Omega = \{(0, \bigcirc_{h_1}), (0, \circledcirc_{h_2}), (0, \square_{h_3}), (0, \triangleright_{h_4}) \mid h_1 \in H_1, h_2 \in H_2, h_3 \in H_3, h_4 \in H_4\}$ are the empty (inactive) neurons

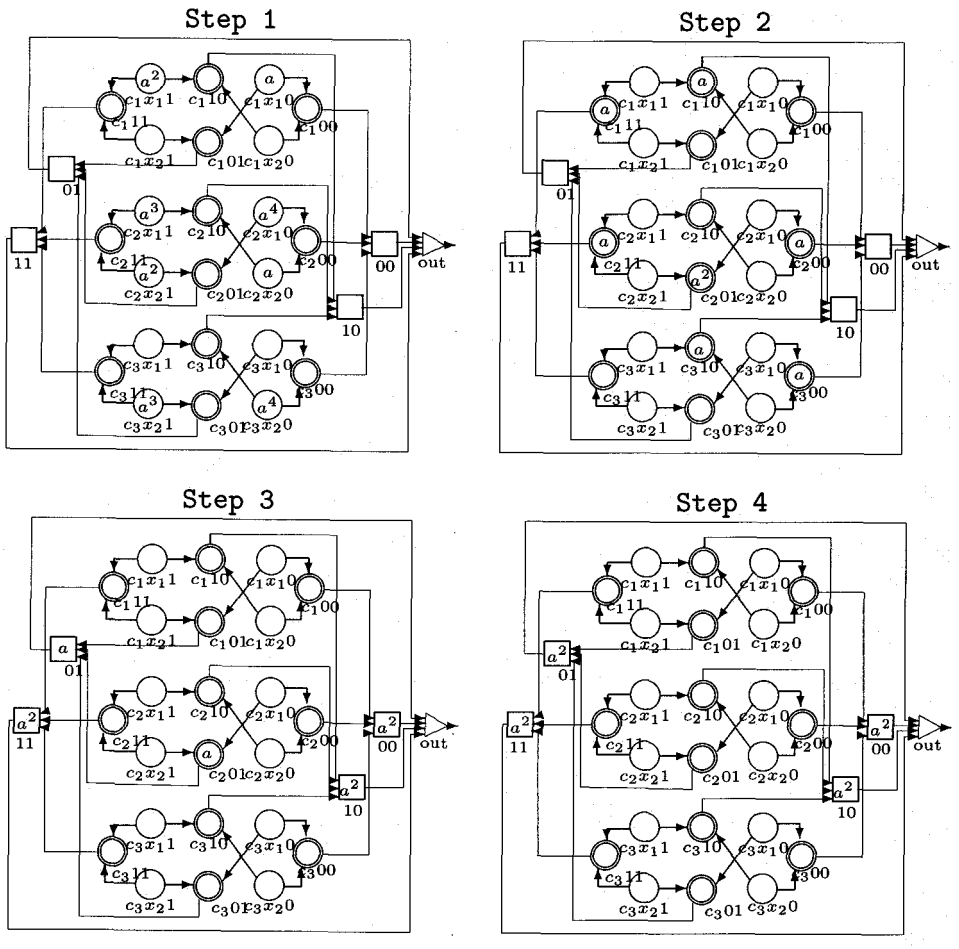



Figure 5.10: The steps of the computation for $\gamma \in SAT(\langle 2,3 \rangle)$.

present in the precomputed structure (with $|\Omega| = 2^n(m+1)+2nm+1$);

- $R_H = R_{H_1} \cup R_{H_2} \cup R_{H_3} \cup R_{H_4}$ is a finite set of rules associated to the neurons, where
  $R_{H_1}$=$\{a^1 \to \lambda,\ a^2 \to a; 0,\ a^3 \to \lambda,\ a^4 \to a; 0\}$,
  $R_{H_2} = R_{H_4} = \{a^+/a \to a; 0\}$,
  $R_{H_3} = \{a^m \to a; 0\}$;
- $syn = \bigcup_{i=1}^{m} \{(\bigcirc_{c_i x_j 1}, \circledcirc_{c_i bin}) \mid bin|_j = 1, 1 \leq j \leq n, bin \in \{0,1\}^n\}$
  $\cup \bigcup_{i=1}^{m} \{(\bigcirc_{c_i x_j 0}, \circledcirc_{c_i bin}) \mid bin|_j = 1, 1 \leq j \leq n, bin \in \{0,1\}^n\}$
  $\cup \{(\circledcirc_{c_i bin}, \square_{bin}) \mid bin \in \{0,1\}^n, 1 \leq i \leq m\} \cup \{\square_{bin}, \triangleright_4 \mid bin \in \{0,1\}^n\}$;

3. $\triangleright_4$ is the output neuron;

- $\Sigma(\langle n,m\rangle)$ is a polynomial encoding from an instance $\gamma$ of SAT into $\Pi_{SAT(\langle n,m\rangle)}$, providing the initialization of the system such that

$$\begin{aligned}\Sigma(\langle n,m\rangle)(\gamma) &= \{(1,\bigcirc_{c_i x_j 0}) \mid x_{i,j} \in \gamma, 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{(2,\bigcirc_{c_i x_j 1}) \mid x_{i,j} \in \gamma, 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{(3,\bigcirc_{c_i x_j 0}) \mid x'_{i,j} \in \gamma, 1 \leq i \leq m, 1 \leq j \leq n\} \\ &\cup \{(4,\bigcirc_{c_i x_j 1}) \mid x'_{i,j} \in \gamma, 1 \leq i \leq m, 1 \leq j \leq n\}.\end{aligned}$$

In the precomputed structure of $\Pi(\langle n,m\rangle)$ (for any SAT problem with $n$ variables and $m$ clauses), neurons are described as a pair *(spikes inside, neuron)*. There are $2^n(m+1)+2nm+1$ neurons and $2^n(3m+1)$ synapses initially inactive. $\Sigma(\langle n,m\rangle)$ encodes the given instance of the problem in spikes, that is, the encoded problem is introduced into the precomputed structure (spikes are assigned to solid circled neurons) activating the corresponding neurons. It is important to note that at most $2nm$ neurons will be activated, hence the initialization takes a polynomial time.

The system evolves exactly in the same manner as the basic SN P systems. The result of the computation is obtained in its 4th step. If the system (output neuron) spikes, then the given problem has at least a solution. Otherwise, it does not have any solution. The evolution of the system after this step of the computation is ignored.

Thus, the system evolves as follows:

Step 1: After encoding the problem (the variables) through (the number of) spikes, we introduce into the system, namely in at most $2nm$ neurons of type $\bigcirc_{c_i 0 x_j 1/0}$, these neurons are activated and evolve according to the rules inside.

Step 2: When we have at most $2^n m$ initially inactive neurons, neurons of type $\circledcirc_{c_i bin}$ which provide information on the truth assignments at the level of clauses (OR operations are simulated) will be activated and they will send spikes to the $2^n$ $\square_{bin}$ neurons corresponding to the truth assignments at the level of the system.

Step 3: Now, only those neurons of type $\square_{bin}$ in which the threshold is reached (AND operations are simulated, if all $m$ clauses are satisfied, hence there exist $m$ spikes inside) will spike (because of the rule $a^m \to a;0$) towards the output neuron.

Step 4: If the output neuron will spike, it means that our problem has at least a solution. Otherwise, we do not have any solution for the given problem.

As we have already mentioned neither in the definition of the system nor in the example, it is mandatory for the system to halt. We only observe its behavior in the fourth step of the computation.

Based on the previous explanations we can state that:

**Theorem 5.7** $\Pi_{\mathtt{SAT}}^{n,m}$ *can deterministically solve each instance of size* $(n, m)$ *of* SAT *in constant time.*

## Final Remarks

In this section, we have shown that SN P systems are not only computationally universal, but also computationally efficient devices. We have shown that the idea of using an already existing, but inactive, workspace proves to be very efficient in solving **NP**-complete problems. We have illustrated this possibility with the satisfiability problem. The initial system is fixed, depending only on the number of variables $(n)$ and the number of clauses $(m)$. Then, any instance of SAT with size $(n, m)$ is encoded in a polynomial time in spikes introduced in the system and then it is solved in exactly 4 steps by our device.

The framework is the following: an arbitrarily large net of neurons is given of a regular form (as the synapse graph) and with only a few types of neurons (as contents and rules), and it is repeated indefinitely; the problem to be solved is plug-in by introducing a polynomial number of spikes in certain neurons (of course, a polynomial quantity), then the system is left to work autonomously; in a polynomial time, it activates an exponential number of neurons, and, after a polynomial time, it outputs the solution to the problem.

Let us suppose that in the system $(\Pi_{SAT(\langle n,m\rangle)}, \Sigma(\langle n, m\rangle))$ we consider rules of type $R_{\circ} = R_{\triangleright} = \{a^+ \to a; 0\}$, which state that if at least one spike is present inside a neuron at a given moment, then, it should spike immediately and all spikes would be consumed. In this case, if the problem has at least a solution, then the computation *halts* at the fourth step with the system spiking. Otherwise, the problem would not have any solution. After the fourth step, no spike will remain in the system.

Another way to stop the computation after sending the answer out is the following: Let us introduce two intermediate neurons in between each $\square_{bin}$ and $\triangleright_4$ neuron, so that at step 4 (the intermediate neurons take one step for transmitting the spikes further) neuron $\triangleright_4$ receives an even number of spikes. Then, instead of the rule $a^+/a \to a; 0$, in neuron $\triangleright_4$ we use the rule $(a^2)^+/a \to a; 0$. Thus, if any spike reaches neuron $\triangleright_4$, then an even number of spikes arrive here; the rule $(a^2)^+/a \to a; 0$ can be used only once, at step 5, because the number of remaining spikes is odd after that.

In this way, we add $2 \cdot 2^n$ neurons and further $3 \cdot 2^n$ synapses, while the computation lasts five steps only.

Moreover, if we consider the 3SAT problem, then each clause block will contain exactly $14 = 3 \cdot 2 + 2^3$ neurons.

This strategy of computing is attractive from a natural computing point of view because it enable us to assume that the brain may be arbitrarily large with respect to the small number of neurons currently used, the same would happen to the cells in liver, etc.). Moreover, a formal framework for defining acceptable solutions to problems by making use of precomputed resources needs to be formulated and investigated. What kind of pre-computed workspace is acceptable? i.e, how much information may be provided for free there?, what kind of net of neurons and what kind of neurons? What means introducing a problem in the existing device (only spikes, also rules, or maybe also synapses?) Defining complexity classes in this case remains as an interesting research topic.

Another idea from [75] is to introduce tools in order to increase the working space in an exponential way in a polynomial time. A suggestion coming from usual P systems is to use an operation similar to cell (membrane) division and cell (membrane) creation, in such a way to make possible polynomial solutions to computationally hard problems.

# Chapter 6

# Computability of Gene Assembly in Ciliates

DNA computing *in vivo* concerns the investigation of computations taking place naturally in a living cell, with the goal of understanding computational properties of DNA molecules in their native environment. Gene assembly in ciliates is perhaps the most involved process of DNA manipulation yet known in living organisms. The computational nature of this process has attracted much attention in recent years. The results obtained so far demonstrate that this process of gene assembly is a splendid example of computing taking place in nature.

Most organisms store their genomic DNA in a linear sequence consisting of coding blocks interspersed with non-coding blocks. For example, if a functional copy of a gene consists of the coding blocks arranged in the order 1-2-3-4-5, it may appear in the order 3-5-4-1-2 in the genome. This presents an interesting problem for organism, which must somehow descramble these genes in order to generate functional proteins required for its continued existence.

**The genetic rearrangement mechanism in ciliates.** Ciliates are unicellular eukaryotes (nucleated cells) that possess two types of nuclei: an active *macronucleus* (soma) and a functionally inert *micronucleus* (germline) which contributes only to sexual reproduction. The active macronucleus forms from the germline micronucleus after sexual reproduction, during the course of development.

This chapter does not address the biological aspects and implications of the gene assembly in ciliates, but the computational aspects of gene assembly.

There were proposed two formal models to explain the gene assembly process in ciliates: the intermolecular model in [55], [58], [59], and the intramolecular model in [34] and [91]. They both are based on so-called "pointers" – short nucleotide sequences (about 20 bp) lying on the borders

between coding and non-coding blocks. Each next coding block starts with a pointer-sequence repeating exactly the pointer-sequence in the end of the preceding coding block from the assembled gene. It is supposed that the pointers guide the alignment of coding blocks during the gene assembly process. The intramolecular model proposes three operations: ld (loop with direct pointers), hi (hairpin loop with inverted pointers), dlad (double loop with alternating direct pointers).

It turns out that the molecular operations performing the gene assembly process in ciliates may be powerful enough to carry out universal computations. In [55] the so-called guided recombination systems were considered for intermolecular operations, defining the contextual sensitivity for molecular operations according to a splicing scheme. The authors in [55] succeeded to show that the intermolecular guided recombination system with *insertion/deletion* operations is computationally universal.

In this chapter, we propose a computing model called *accepting intramolecular recombination* (AIR) *system* based on the contextual variants of the intramolecular recombination operations (dlad, ld). We prove that the AIR systems have the computational power of Turing machines. This is rather interesting, because processing a single stranded molecule by a process using only two well motivated gene rearrangement operations (ctrl, del) provides a computing model equivalent with Turing machines.

In Section 6.2.2, we also extend the AIR systems with the string (molecule) replication operation (repl), and consider the maximally parallel application of rules. Such *extended* AIR systems can solve computationally hard problems (in particular SAT) in feasible time.

## 6.1 The Gene Assembly Operations

We briefly mention here the two operations suggested by the intermolecular model [58].

**Intramolecular Recombination:** excises from a linear molecule a circular molecule, i.e., $\alpha p \beta p \gamma \rightarrow \alpha p \gamma + \bullet \beta p$, where $\alpha p \beta p \gamma$ is an initial linear molecule, $\alpha p \gamma$ and $\bullet \beta p$ are resulting molecules;

**Intermolecular Recombination:** inserts a circular molecule into a linear one, i.e., $\bullet \beta p + \alpha p \gamma \rightarrow \alpha p \beta p \gamma$, where $\alpha p \gamma$ and $\bullet \beta p$ are initial molecules, while $\alpha p \beta p \gamma$ is the resulting molecule.

In the expressions from above $\alpha p \beta p \gamma$ and $\alpha p \gamma$ are linear molecules, while $\bullet \beta p$ is a circular molecule, $p$ is the pointer, $\alpha$, $\beta$ and $\gamma$ are sequences of base pairs of nucleotides.

### Intramolecular Gene Assembly Operations

The intramolecular operations excise non-coding blocks from the micronuclear DNA-molecule, interchange positions of some portions of the molecule or invert them, so as to obtain after some rearrangements the DNA-molecule containing a continuous succession of codding blocks, i.e., the assembled gene. Contrary to the intermolecular model, all the molecular operations in the intramolecular model are performed within a single molecule.

We recall bellow three intramolecular operations conjectured in [34] and [91] for the gene assembly, which were proved to be complete [33], i.e., any sequence of coding and non-coding blocks can be assembled to the macronuclear gene by means of these operations (for details related to the intramolecular model we refer to [32]):

- ld excises non-coding block flanked by repeated pointers from the molecule in the form of a circular molecule as shown in Figure 6.1.
- hi inverts part of the molecule flanked by pointers, where one pointer is the inversion of another one as shown in Figure 6.2.
- dlad swaps two parts of the molecule delimited by a repeating pair of nonequal pointers as shown in Figure 6.3.

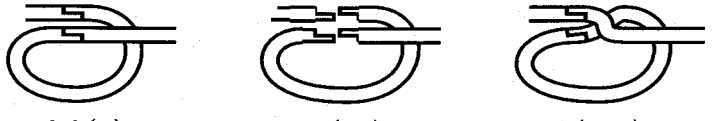



Figure 6.1: Loop Recombination: (i) the molecule folds on itself aligning pointers in the direct repeat to form the loop, (ii) enzymes cut on the pointer sites, (iii) hybridization happens. As the result, a portion of the molecule in the loop is excised in the form of circular molecule.

### The Intermolecular Operations with Contexts

In what follows, following [55], we consider operations as above, controlled by pair of triples like in splicing scheme. Specifically, taking a splicing scheme $(\Sigma, \sim)$ (see Section 2.1.5) the pairs $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ define the contexts necessary for a recombination between the repeats $p$. Then the contextual intramolecular recombination is defined by

$$\{upwpv\} \Rightarrow \{upv, \bullet wp\}, \text{ where } u = u'\alpha, w = \beta w' = w''\alpha', v = \beta' v'.$$

Similarly, if $(\alpha, p, \beta) \sim (\alpha', p, \beta')$, then the contextual intermolecular recombination is defined by

$$\{upv, \bullet wp\} \Rightarrow \{upwpv\}, \text{ where } u = u'\alpha, w = \beta' w'' = w'\alpha', v = \beta v'.$$

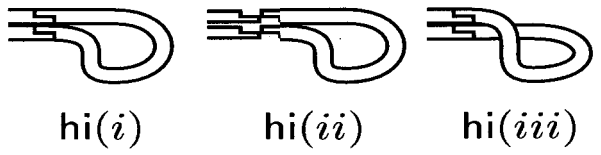



Figure 6.2: Hairpin Recombination: (i) the molecule folds on itself aligning pointers in the inverted repeat to form the hairpin, (ii) enzymes cut on the pointer sites, (iii) hybridization happens. As the result, a portion of the molecule in the hairpin is inverted.

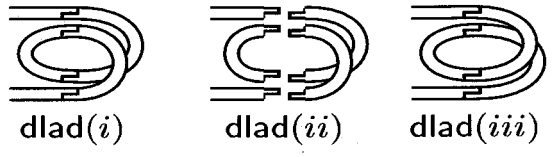



Figure 6.3: Double-Loop Recombination: (i) the molecule folds on itself aligning equal pointers from the repeated pair to form a double loop, (ii) enzymes cut on the pointer sites, (iii) hybridization happens. As the result, portions of the molecule in the loops interchange their places.

**Definition 6.1** *A guided recombination system is a triple* $R = (\Sigma, \sim, A)$ *where* $(\Sigma, \sim)$ *is a splicing scheme, and* $A \in \Sigma^+$ *is a linear string called the axiom.*

A guided recombination system $R$ defines a derivation relation that produces a new multiset from a given multiset of linear and circular strands, as follows. Starting from a "collection" (multiset) of strings with a certain number of available copies of each string, the next multiset is derived from the first one by an intra- or inter-molecular recombination between existing strings. The strands participating in the recombination are "consumed"

(their multiplicity decreases by 1) whereas the products of the recombination are added to the multiset (their multiplicity increases by 1).

For two multisets $S$ and $S'$ in $\Sigma^* \cup \Sigma^\bullet$, we say that $S$ derives $S'$ and we write $S \Longrightarrow_R S'$, iff one of the following two cases hold:

(1) there exist $x \in S, y, \bullet z \in S'$ such that

- $\{x\} \Longrightarrow \{y, \bullet z\}$ according to an intramolecular recombination step in $R$,
- $S'(x) = S(x) - 1, S'(y) = S(y) + 1, S'(\bullet z) = S(\bullet z) + 1$, and $S'(u) = S(u)$ for all $u \notin \{x, y, \bullet z\}$;

(2) there exist $x', \bullet y' \in S, z' \in S'$ such that

- $\{x', \bullet y'\} \Longrightarrow \{z'\}$ according to an intermolecular recombination step in $R$,
- $S'(x') = S(x') - 1, S'(\bullet y') = S(\bullet y') - 1, S'(z') = S(z') + 1$, and $S'(u) = S(u)$ for all $u \notin \{x', y', \bullet z'\}$.

Those strands which, by repeated recombinations with initial and intermediate strands eventually produce the axiom, form the language of the guided recombination system. Formally,

$$L_a^k(R) = \{w \in \Sigma^* \mid \{(w, k)\} \Longrightarrow_R^* S \text{ and } A \in S\}$$

($(w, k)$ indicates the fact that the multiplicity of $w$ equals $k$).

The guided recombination systems are proved in [55] to be equivalent to Turing machine:

**Theorem 6.1** *Let $L$ be a language over $T^*$ accepted by a Turing machine $TM = (S, \Sigma \cup \{\#\}, P)$. Then there exist an alphabet $\Sigma'$, a sequence $\pi \in \Sigma'^*$, depending on $L$, and a recombination system $R$ such that a word $w$ over $T^*$ is in $L$ iff $\#^6 s_0 w \#^6 \pi$ belongs to $L_a^k(R)$ for some $k \geq 1$.*

## 6.2 The Contextual Intramolecular Operations

We now define the contextual intramolecular *translocation* and *deletion* operations based on dlad and ld, respectively. We follow here the style of contextual intermolecular recombination operations defined above. The contextual intramolecular recombinations are a generalization of the intramolecular recombination operations on strings: recombinations can be done only if certain *contexts* are present.

Let us consider a splicing scheme $R = (\Sigma, \sim)$ where $\Sigma$ is the alphabet and $\sim$ is the pairing relation of the scheme. Denote $core(R) = \{p \mid (\alpha, p, \beta) \sim (\gamma, p, \delta) \in R \text{ for some } \alpha, \beta, \gamma, \delta \in \Sigma^+\}$.

**Definition 6.2** *Let $u$ and $v$ be two nonempty words in $\Sigma^*$. The contextual intramolecular translocation operation with respect to $R$ is defined as*

$$\mathsf{ctrl}_{p,q}(xpuqypvqz) = xpvqypuqz,$$
where
$$x = x'\alpha,\ uqy = \beta u' = u''\alpha',\ vqz = \beta' v',$$
$$xpu = u'''\gamma,\ ypv = \delta v'' = v'''\gamma',\ z = \delta' z',$$
with $(\alpha, p, \beta) \sim (\alpha', p, \beta')$ and $(\gamma, q, \delta) \sim (\gamma', q, \delta')$.

Intuitively, $\mathsf{ctrl}_{p,q}$ is applicable if the contexts of the two occurrences of $p$ as well as the contexts of the two occurrences of $q$ are in the relation $\sim$. The result of applying $\mathsf{ctrl}_{p,q}$ is that the strings $u$ and $v$, each one is flanked by pointers $p$ and $q$, are swapped. If $\mathsf{ctrl}_{p,q}(w) = w'$, we also write $w \Longrightarrow_{\mathsf{ctrl}_{p,q}} w'$.

**Definition 6.3** *Let $u$ be a nonempty word in $\Sigma^*$. The contextual intramolecular deletion operation with respect to $R$ is defined as:*

$$\mathsf{del}_p(xpupy) = xpy,$$
where
$$x = x'\alpha,\ u = \beta u' = u''\alpha',\ y = \beta' y',$$
with $(\alpha, p, \beta) \sim (\alpha', p, \beta')$.

As a result of applying $\mathsf{del}_p$, the substring $u$ flanked by the two occurrences of $p$ is removed together with one string $p$, provided that the contexts of those occurrences of $p$ are in the relation $\sim$. If $\mathsf{del}_p(w) = w'$, we also write $w \Longrightarrow_{\mathsf{del}_p} w'$.

We define the set of all contextual intramolecular operations under the guiding of $\sim$ as follows:

$$\widetilde{R} = \{\mathsf{ctrl}_{p,q}, \mathsf{del}_p \mid p, q \in core(R)\}.$$

The recombination relation $\Longrightarrow_{\mathsf{ctrl}_{p,q}}, \Longrightarrow_{\mathsf{del}_p}$ are denoted in general by $\Longrightarrow_{\widetilde{R}}$.

Now we define an accepting intramolecular recombination (AIR) system that models the series of dispersed homologous recombination events that take place on a single scrambled gene in ciliates.

**Definition 6.4** *An accepting intramolecular recombination system is a quadruple $G = (\Sigma, \sim, \alpha_0, w_t)$ where $R = (\Sigma, \sim)$ is a splicing scheme, $\alpha_0 \in \Sigma^*$ is the start word – the recombination starts from $\alpha_0$, and $w_t \in \Sigma^+$ is a linear string called the target.*

The language accepted by $G$ is defined by

$$L(G) = \{w \in \Sigma^* \mid \alpha_0 w \Longrightarrow_{\widetilde{R}}^* w_t\}.$$

### 6.2.1 Computational Universality

**Theorem 6.2** *Let $M = (S, \Sigma \cup \{\#\}, P)$ be a Turing machine. Then there exists an alphabet $\Sigma'$, an accepting intramolecular recombination system $G_M = (\Sigma', \sim, \alpha_0, w_t)$, and a string $\pi_M \in \Sigma'^*$ such that for any word $w$ over $\Sigma^*$ there exists $k_w \geq 1$ such that $w \in L(M)$ if and only if $w\#^4\$_0\pi_M^{k_w}\#^2 \in L(G_M)$.*

*Proof.* We use here a proof idea/construction similar to that in [55], but, for the sake of completeness, we present the proof with full details.

(I) Let a word $w$ be accepted by the given Turing machine $M$. Consider that the rules of $P$ are ordered in an arbitrary fashion and numbered. Thus, if there are $m$ rules in $P$, then a rule is of the form $i : u_i \rightarrow v_i$, where $2 \leq |u_i| \leq 3$, $1 \leq |v_i| \leq 4$, $1 \leq i \leq m$. For the Turing machine $M$ we construct an intramolecular recombination system

$$G_M = (\Sigma', \sim, \alpha_0, w_t)$$

where

$$\begin{aligned} \Sigma' &= S \cup \Sigma \cup \{\#\} \cup \{\$_i \mid 0 \leq i \leq m\}, \\ \alpha_0 &= \#^4 s_0, \\ w_t &= \#^4 s_f \#^3, \\ R_G &= (\Sigma', \sim) \text{ is defined below.} \end{aligned}$$

We also consider the sequence $\pi_M \in \Sigma'^*$ consisting of the concatenation of the right-hand sides of the rewriting rules in $P$ bounded by markers, as follows:

$$\pi_M = (\prod_{\substack{1 \leq i \leq m \\ p_i, q_i \in \Sigma \cup \#}} \$_i p_i v_i q_i \$_i)\$_{m+1},$$

where $i : u_i \rightarrow v_i, 1 \leq i \leq m$, are the rules of $M$.

We denote by $dol(\pi_M)$ the string obtained by removing from $\pi_M$ all symbols different form the dollar sign.

If a word $w \in \Sigma^*$ is accepted by $M$, then a computation of $G_M$ starts from a string of the form $\delta = \#^4 s_0 w \#^4 \$_0 \pi_M^{k_w} \#^2$, where $k_w$ is a number which will be defined below. We refer to the subsequence $\#^4 s_0 w \#^4$ as the "data", and to the subsequence $\$_0 \pi_M^{k_w} \#^2$ as the "program" of $\delta$.

A rewriting rule $i : u \rightarrow v \in P$ of $M$ is simulated by a contextual translocation rule $\mathsf{ctrl}_{p,q} \in R_G$. We construct the relation $R_G$ so that when the left-hand side of a rule $i : u_i \rightarrow v_i$ appears in the data the application of the rule can be simulated in $G_M$ by swappping $u_i$ (left-hand side of the rule in the "data") and corresponding $v_i$ (right-hand side of the rule in the "program").

If $u_i$ appears in the data several times during the computation, each time the corresponding $v_i$ is chosen from the concatenated original copies of

$\pi_M$ in the program. In order to have enough copies of each $u_i$, we take $k_w$ sufficiently large, for instance, equal with the number of steps the machine $M$ performs in order to recognize $w$.

According to the above construction of the alphabet $\Sigma'$, sequence $\pi_M$ and recombination system $G_M$, for any $x, y, z \in \Sigma'^*$, to accomplish the translocation step for a rule $i : u_i \rightarrow v_i \in P$ we want to perform the contextual intramolecular translocation operation $\mathsf{ctrl}_{p_i,q_i}$:

$$xcp_iu_iq_idy\$_ip_iv_iq_i\$_iz \Longrightarrow_{\mathsf{ctrl}_{p_i,q_i}} xcp_iv_iq_idy\$_ip_iu_iq_i\$_iz,$$

and to this aim we introduce the following contexts for the pointers $p_i$ and $q_i$ of $\mathsf{ctrl}_{p_i,q_i}$:

(i) $(c, p_i, u_iq_id) \sim (\$_0o\$_i, p_i, v_iq_i\$_i)$ and

(ii) $(cp_iu_i, q_i, d) \sim (\$_ip_iv_i, q_i, \$_i)$,

for all $c \in \{\#\}^*\Sigma^*, d \in \Sigma^*\{\#\}^*, |c| = |d| = 2$,

$p_i, q_i \in \Sigma \cup \{\#\},\ o \in \Sigma'^*, |o| \leq |\pi_M^{k_w}|$.

Flankers $p_i$ and $q_i$ are a sort of "place holders" to allow for all the possible combinations of contexts. Consequently, for one rule of $M$ we have as many possible operations as many letters the alphabet has, to take into account all the possibilities.

We know that $u_i$ and $v_i$ contain exactly one state $s_i \in S \subset \Sigma'$, hence when choosing $u_i$ the flankers $p_i$ and $q_i$ are closest to each other, in other words there is no pointer $p_i$ or $q_i$ between them. The case of strings $v_i$ will be discussed immediately.

By the application of $\mathsf{ctrl}_{p_i,q_i}$, the substrings $u_i$ and $v_i$ flanked by pointers $p_i$ and $q_i$ are swapped.

$$xcp_iu_iq_idy\$_1 \cdots \$_{i-1}\$_ip_iv_iq_i\$_i\$_{i+1} \cdots \$_mz \Longrightarrow_{\mathsf{ctrl}_{p_i,q_i}}$$
$$xcp_iv_iq_idy\$_1 \cdots \$_{i-1}\$_ip_iu_iq_i\$_i\$_{i+1} \cdots \$_mz.$$

It is possible that a wrong substring would be swapped with $u_i$ by $\mathsf{ctrl}_{p_i,q_i}$ if the pointers $\$_ip_i$ and $q_i\$_i$ chosen by the contexts $(i)$ and $(ii)$ in the program are not closest, but contains other similar contexts in between them. For instance, we can have a string of the form

$$\ldots \$_ip_iv_iq_i\$_i \ldots \$_ip_iv_iq_i\$_i \ldots,$$

and we swap the string $u_i$ from the data with the string $v_iq_i\$_i \ldots \$_ip_iv_i$, hence the one bounded by the first $p_i$ and the last $q_i$ (and not by $v_i$).

In this case, at least a symbol $\$_i$ (such a symbol is not in $\Sigma \cup S$) appears in the data. The occurrence of \$ will never be removed from data by $\mathsf{ctrl}_{p_i,q_i}$, because the second occurrence of $p$ in $(i)$ is imposed to be in the program,

more precisely, such $p$ must to be preceded by a substring $\$_0 o \$_i$ as in the relation $(i)$. Consequently, the simulation will get stuck, the data will never arrive to a string of the form $\#^4 s_f \#^4$ as needed below for reaching the target string.

Moreover, in the context of the relation (i) (or in the context of the relation (ii)), if the deletion operation $\mathsf{del}_p$ (or $\mathsf{del}_q$) of $G_M$ is applied to the repeats of pointer $p$ (or $q$) one of which is from the left side and another one is from the right side of the substring $\ldots \$_0 \ldots$, substring $\ldots \$_0 \ldots$ is deleted. Then again the simulation will get stuck since the relations (i) as above and (iv) as below will not be satisfied any longer.

When we get the final state $s_f$ in the data, we clean the used "program". For this we use the next deletion operation and delete all $u_i$ and $v_i$ from the program.

$$\mathsf{del}_{\$_i} : \#^4 s_f \#^4 x \$_i a_i \$_i \$_{i+1} y \Longrightarrow_{\mathsf{del}_{\$_i}} \#^4 s_f \#^4 x \$_i \$_{i+1} y.$$

To this aim we need the following relations

$$\text{(iii)} \quad (\#\#s_f\#^4 x, \$_i, a_i) \sim (a_i, \$_i, \$_{i+1}),\ x \in \{\$_j \mid 0 \leq j \leq m+1\}^*$$
$$a_i \in \{u_j, v_j \mid u_j \to v_j \in P, 1 \leq j \leq m\},\ 1 \leq i \leq m.$$

The left- and right-hand sides ($u_r$ and $v_s$) of the rules of $M$ are deleted. At the end of the deletion $\mathsf{del}_{\$_i}$, the string becomes

$$\#^4 s_f \#^4 \$_0 (dol(\pi_M))^{k_w} \#\#,\ k_w \geq 1.$$

Then we use one more deletion operation:

$$\mathsf{del}_{\#} : \#^4 s_f \#^4 \$_0 (dol(\pi_M))^{k_w} \#\# \Longrightarrow_{\mathsf{del}_{\#}} \#^4 s_f \#^3$$

which is applied in the context of the repeated occurrences of #, according to the splicing relation

$$\text{(iv)} \quad (\#\#s_f\#\#\#, \#, \$_0 (dol(\pi_M))^{k_w}) \sim (\$_0 (dol(\pi_M))^{k_w}, \#, \#).$$

Thus, we reach to the target $\#\#\#\#s_f\#\#\#$.

Concluding, a derivation step of machine $M$ can be simulated by the contextual translocation operation in the intramolecular recombination system $G_M$. If the TM accepts a word $w$, then we can start a recombination in $G_M$ from the string $\#^4 s_0 w \#^4 \$_0 \pi_M^{k_w} \#^2$ and reach the target $\#\#\#\#s_f\#\#\#$ by only using the recombinations according to $G_M$:

$$\#^4 s_0 w \#^4 \$_0 \pi_M^{k_w} \#^2 \Longrightarrow^*_{R_G} \#\#\#\#s_f\#\#\#.$$

This means that our word is accepted by $G_M$.

(II) For the converse implication, it suffices to prove the next two claims. (1) Starting from the string $w\#^4 \$_0 \pi_M^{k_w} \#^2 \in L(G_M)$, no other recombinations except those that simulate steps of $M$ are possible in $G_M$. (2) No

other word $\phi$ different from $w\#^4\$_0\pi_M^{k_w}\#^2$ corresponding to $w \in L(M)$ is in $L(G_M)$.

(1) Assume that the current recombination step of $G_M$ is of the form

$$\begin{aligned}\delta' &= xcp_iu_iq_idyr_{j_1}r_{j_2}\dots r_{j_s}z,\\ &\quad r_{j_k} \in \{\$_ip_iv_iq_i\$_i \mid 1 \le i \le m\} \cup \{\$_i \mid 1 \le i \le m\}.\end{aligned}$$

Then,

- we cannot use relation $(i)$ in the program because that would require the presence of strands $cpuqd$, which cannot appear in the program since $c$ and $d$ do not contain \$,

- we cannot use relation $(ii)$ in the data, because that would require a substring of type $\$pvq\$$, and data does not contain any \$ symbol.

These two arguments show that $puq$ and $pvq$ in $\mathsf{ctrl}_{p,q}$ are chosen only from the data and the program, respectively.

Assuming that a subword $pu_iq$ contains the left-hand side of a rule $i : u_i \to v_i$ which appears in the data, and assuming that the necessary $pv_iq$ is in the program, the next step is to show that the only possible translocation is simulating a rewriting step of $M$ using rule $i$. In other words, $pu_iq$ cannot be changed by $pv_jq$ in the data where $i \neq j$, because once $u_i$ appears in the data, it has to be changed by only $v_i$. The indices of the subwords $pu_iq$ and $\$_ipv_iq\$_i$, and the contexts of $u$ and $v$ used in the translocation operation $\mathsf{ctrl}_{p,q}$ are identical. As we have seen above, when choosing the strings to swap, they should correspond to strings $u_i$ (in the data) and $v_i$ (in the program), otherwise we cannot reach the target string.

This implies that the only possible operations done by translocation steps simulate (correspond to) legal (correct) rewriting of the Turing machine.

(2) Assume $\phi = \#^4s_0w'\#^4\$_0\pi'^{k_w}\#\# \in L(G_M)$ and it contains a substring $X \notin \{u, v \mid u \to v \in P\}$. If $|w'|_X \neq 0$, then the translocation operation will stuck when it reaches $X$ since $X \notin (\Sigma \cup S)^*$; however, we cannot get the substring $\#^4s_f\#^4$ in the data. If $|\pi'^{k_w}|_X \neq 0$, when $G_M$ deletes $u_i$ and $v_i$, $u_i, v_i \in \{u, v \mid u \to v \in P\}$, from the program by $\mathsf{del}_{\$_i}$ after $s_f$ appeared in the data, the deletion rule $\mathsf{del}_{\$_i}, 1 \le i \le m$ cannot delete $X$ from $\pi'^{k_w}$ because $(iii)$ is not satisfied since $X \notin \{u, v \mid u \to v \in P\}$. Then $\#^4s_f\#^4\$_0(dol(\pi_M))^{k_w}\#\#$ cannot be obtained by $\mathsf{del}_{\$_i}$. This implies that the deletion rule $\mathsf{del}_{\#}$ cannot apply anymore since $(iv)$ cannot be satisfied. The target of $G_M$ cannot be reached, $\#^4s_0w'\#^4\$_0\pi'^{k_w}\#\# \not\Rightarrow_{R_G}^* \#^4s_f\#^3$, which means $\phi \notin L(G_M)$. Thus, the converse implication is proved.

The conclusions of (I) and (II) altogether complete the proof of the theorem. □

### 6.2.2 Computational Efficiency

Now, we concentrate on computational complexity issue of the computing model for intramolecular gene assembly process, and we show that this model can solve **NP**-complete problems (in particular, SAT) in a feasible time.

To this aim, we need an additional operation, *the contextual intramolecular replication.* Let again $R = (\Sigma, \sim)$ be a splicing scheme.

**Definition 6.5** *Let $u$ and $v$ be two nonempty words in $\Sigma^*$. The contextual intramolecular replication operation is defined as:*

$$\begin{aligned}&\mathsf{repl}_p(xpupy) = xpuupy,\\ &\text{where}\\ &x = x'\alpha,\ u = \beta = \alpha',\ y = \beta' y',\\ &\text{with } (\alpha, p, \beta) \sim (\alpha', p, \beta').\end{aligned}$$

As a result of applying $\mathsf{repl}_p$, the string $u$ flanked by the two occurrences of $p$ is replicated.

We define the set of the contextual intramolecular operations under the guiding of $\sim$ as follows:

$$\widetilde{P} = \{\mathsf{ctrl}_{p,q}, \mathsf{del}_p, \mathsf{repl}_p \mid p, q \in core(R)\}.$$

We consider the parallelism for the intramolecular recombination model. Intuitively, a number of operations can be applied in parallel to a string if the applicability of each operation is independent of the applicability of the other operations. The parallelism in intramolecular gene assembly was initially studied in a different setting in [41]. We recall the definition of parallelism following [41] with a small modification adapted to our model.

**Definition 6.6** *Let $S \subseteq \widetilde{P}$ be a set of $k$ rules and let $u$ be a string. We say that the rules in $S$ can be applied in parallel to $u$ if for any ordering $\varphi_1, \varphi_2, \ldots, \varphi_k$ of $S$, the composition $\varphi_k \circ \varphi_{k-1} \circ \cdots \circ \varphi_1$, is applicable to $u$.*

In our proof below we use a different notion of parallelism. We introduce it here is the form of the maximally parallel application of a rule to a string.

First, we define the working places of a operation $\varphi \in \widetilde{P}$ on a given string where $\varphi$ is applicable.

**Definition 6.7** *Let $w$ be a string. The working places of a operation $\varphi \in \widetilde{P}$ for $w$ is a set of substrings of $w$ written as $Wp(\varphi(w))$ and defined by*

$$\begin{aligned}Wp(\mathsf{ctrl}_{p,q}(w)) &= \{(w_1, w_2) \in Sub(w) \mid \mathsf{ctrl}_{p,q}(xw_1yw_2z) = xw_2yw_1z\}.\\ Wp(\mathsf{del}_p(w)) &= \{w_1 \in Sub(w) \mid \mathsf{del}_p(xpw_1py) = xpy\}.\\ Wp(\mathsf{repl}_p(w)) &= \{w_1 \in Sub(w) \mid \mathsf{repl}_p(xw_1y) = xw_1w_1y\}.\end{aligned}$$

**Definition 6.8** *Let $w$ be a string. The smallest working places of a operation $\varphi \in \widetilde{P}$ for $w$ is a subset of $Wp(\varphi)(w)$ written as $Wp_s(\varphi(w))$ and defined by*

$$\begin{aligned}
Wp_s(\mathsf{ctrl}_{p,q}(w)) &= \{(w_1, w_2) \in Wp(\mathsf{ctrl}_{p,q}(w)) \mid w_1' \in Sub(w_1) \text{ and } \\
&\quad w_2' \in Sub(w_2) \text{ and } (w_1', w_2') \neq (w_1, w_2), \\
&\quad (w_1', w_2') \notin Wp(\mathsf{ctrl}_{p,q}(w))\}. \\
Wp_s(\mathsf{del}_p(w)) &= \{w_1 \in Wp(\mathsf{del}_p(w)) \mid w_1' \in Sub(w_1), \\
&\quad \text{and } w_1' \neq w_1, w_1' \notin Wp(\mathsf{del}_p(w))\}. \\
Wp_s(\mathsf{repl}_p(w)) &= \{w_1 \in Wp(\mathsf{repl}_p(w)) \mid w_1' \in Sub(w_1), \\
&\quad \text{and } w_1' \neq w_1, w_1' \notin Wp(\mathsf{repl}_p(w))\}.
\end{aligned}$$

**Definition 6.9** *Let $\Sigma$ be a finite alphabet and $\widetilde{P}$ the set of rules defined above. Let $\varphi \in \widetilde{P}$ and $u \in \Sigma^*$. We say that $v \in \Sigma^*$ is obtained from $u$ by applying $\varphi$ in a maximally parallel way, denoted $u \Longrightarrow_{\varphi}^{max} v$, if*

$$u = \alpha_1 u_1 \alpha_2 u_2 \ldots \alpha_k u_k \alpha_{k+1}, \text{ and } v = \alpha_1 v_1 \alpha_2 v_2 \ldots \alpha_k v_k \alpha_{k+1},$$

*where $u_i \in Wp_s(\varphi)(w)$ for all $1 \leq i \leq k$, and also, $\alpha_i \notin Wp(\varphi(w))$, for all $1 \leq i \leq k+1$.*

Note that a rule $\varphi \in \widetilde{P}$ may be applied in parallel to a string in several different ways, as shown in the next example.

**Example 6.1** *Let $\mathsf{ctrl}_{p,q}$ be the contextual translocation operation applied in the context $(x_1, p, x_2) \sim (x_3, p, x_4)$ and $(y_1, q, y_2) \sim (y_3, q, y_4)$. We consider the string*

$$u = x_1 p x_2 \$_1 y_1 q y_2 \$_2 x_3 p x_4 \$_3 x_3 p x_4 \$_4 y_3 q y_4.$$

*Note that there are two occurrences of $p$ with context $x_3$ and $x_4$. We can obtain two different strings from $u$ by applying $\mathsf{ctrl}_{p,q}$ in a parallel way as follows:*

$$\begin{aligned}
u &\Longrightarrow_{\mathsf{ctrl}_{p,q}}^{max} v' \text{ where } v' = x_1 p x_4 \$_3 x_3 p x_4 \$_4 y_3 q y_2 \$_2 x_3 p x_2 \$_1 y_1 q y_4. \\
u &\Longrightarrow_{\mathsf{ctrl}_{p,q}}^{max} v'' \text{ where } v'' = x_1 p x_4 \$_4 y_3 q y_2 \$_2 x_3 p x_4 \$_3 x_3 p x_2 \$_1 y_1 q y_4.
\end{aligned}$$

*Here only the second case satisfies the definition of maximally parallel application of $\mathsf{ctrl}_{p,q}$ because it applies for the smallest working place.*

**Example 6.2** *Let $\mathsf{del}_p$ be the contextual deletion operation applied in the context $(x_1 x_2, p, x_3) \sim (x_3, p, x_1)$, and consider the string $u = x_1 x_2 p x_3 p x_1 x_2 p x_3 p x_1$. The unique correct result obtained by maximally parallel application of $\mathsf{del}_p$ to $u$ is:*

$$x_1 x_2 p x_3 p x_1 x_2 p x_3 p x_1 \Longrightarrow_{\mathsf{del}_p}^{max} x_1 x_2 p x_1 x_2 p x_1.$$

Now we define a computing model as follows.

**Definition 6.10** *An extended accepting intramolecular recombination (AIR) system is a triple $G = (\Sigma, \sim, A)$ where $(\Sigma, \sim)$ is a splicing scheme with the set of rules $\widetilde{P} = \{\mathsf{ctrl}_{p,q}, \mathsf{del}_p, \mathsf{repl}_p \mid p, q \in core(R)\}$ which are applied in maximally parallel manner, and $A \in \Sigma^+$ is a linear string called the axiom.*

*The language accepted by $G$ is defined by*

$$L(G) = \{w \in \Sigma^* \mid w \Longrightarrow_{\widetilde{P}}^* A\}.$$

We use extended accepting intramolecular recombination (AIR) systems as decision problem solvers. A possible correspondence between decision problems and languages can be done via an encoding function which transforms an instance of a given decision problem into a word, see, e.g., [37].

**Definition 6.11** *We say that a decision problem $X$ is solved in time $O(t(n))$ by extended accepting intramolecular recombination systems if there exists a family $\mathcal{A}$ of extended AIR systems such that the following conditions are satisfied:*

1. *The encoding function of any instance $x$ of $X$ having size $n$ can be computed by a deterministic Turing machine in time $O(t(n))$.*
2. *For each instance $x$ of size $n$ of the problem one can effectively construct, in time $O(t(n))$, an extended accepting intramolecular recombination system $G(x) \in \mathcal{A}$ which decides, again in time $O(t(n))$, the word encoding the given instance. This means that the word is accepted if and only if the solution to the given instance of the problem is* YES.

**Theorem 6.3** SAT *can be solved deterministically in linear time by an extended accepting intramolecular recombination (AIR) system constructed in polynomial time.*

*Proof.* Let us consider a propositional formula in the conjunctive normal form, $\alpha = C_1 \wedge \cdots \wedge C_m$, such that each clause $C_i, 1 \leq i \leq m$, is of the form $C_i = y_{i,1} \vee \cdots \vee y_{i,k_i}, k_i \geq 1$, where $y_{i,j} \in \{x_k, \bar{x}_k \mid 1 \leq k \leq n\}$.

We construct an extended accepting intramolecular recombination system

$$G = (\Sigma, \sim, \mathtt{YES}),$$

where

$$\begin{aligned}
\Sigma &= \{\$_i \mid 0 \leq i \leq m+1\} \cup \{x_i, x_{\bar{i}} \mid 1 \leq i \leq n\} \cup \{\langle_i, \langle_{\bar{i}}, \rangle_i, \rangle_{\bar{i}} \mid 1 \leq i \leq n\} \\
&\cup \{\dagger_i, \dagger_{\bar{i}} \mid 1 \leq i \leq n\} \cup \{f_i \mid 0 \leq i \leq n+1\} \cup \{T, F, \vee, \mathtt{Y}, \mathtt{E}, \mathtt{S}\}, \\
\tilde{P} &= \{\mathsf{repl}_{f_i} \mid 0 \leq i \leq n-1\} \cup \{\mathsf{del}_{f_i}, \mathsf{del}_{\dagger_i}, \mathsf{del}_{\dagger_{\bar{i}}} \mid 1 \leq i \leq n\} \\
&\cup \{\mathsf{ctrl}_{\langle_i,\rangle_i}, \mathsf{ctrl}_{\langle_{\bar{i}},\rangle_{\bar{i}}} \mid 1 \leq i \leq n\} \cup \{\mathsf{del}_{\$_i} \mid 1 \leq i \leq m\} \cup \{\mathsf{del}_{\mathtt{E}}\}.
\end{aligned}$$

We encode each clause $C_i$ as a string bounded by $\$_i$ in the following form:

$$c_i = \$_i \vee \langle_{\sigma(i_b)} x_{\sigma(i_b)} \rangle_{\sigma(i_b)} \vee \cdots \vee \langle_{n_{\sigma(i_b)}} x_{n_{\sigma(i_b)}} \rangle_{n_{\sigma(i_b)}} \vee \$_i,$$

where $b \in \{0,1\}, \sigma(i_1) = i, \sigma(i_0) = \bar{i}$, and $x_i$ stands for variable $x_i$ in the formula, while $x_{\bar{i}}$ stands for negated variable $\bar{x}_i$, $1 \leq i \leq n$.

The instance $\alpha$ is encoded as follows:

$$\delta = \$_0 c_1 \ldots c_m \$_{m+1}.$$

In order to generate all possible truth-assignments for all variables $x_1, x_2, \ldots, x_n$ of the formula, we consider a string of the form

$$\gamma = \dagger_1 \langle_1 T \rangle_1 \dagger_1 \dagger_{\bar{1}} \langle_{\bar{1}} F \rangle_{\bar{1}} \dagger_{\bar{1}} \ldots \dagger_n \langle_n T \rangle_n \dagger_n \dagger_{\bar{n}} \langle_{\bar{n}} F \rangle_{\bar{n}} \dagger_{\bar{n}}.$$

Here $T$ and $F$ denote the truth-values *true* and *false*, respectively. Then $m$ copies of $\gamma$ are attached to the end of the encoded formula, leading to $\beta = \delta\gamma^m$.

Let us have an example:

$$\begin{aligned}
\varepsilon &= (x_1 \vee \bar{x}_2) \wedge (x_1 \vee x_2). \\
\varepsilon' &= \$_0\$_1 \vee \langle_1 x_1 \rangle_1 \vee \langle_{\bar{2}} x_{\bar{2}} \rangle_{\bar{2}} \vee \$_1\$_2 \vee \langle_1 x_1 \rangle_1 \vee \langle_2 x_2 \rangle_2 \vee \$_2\$_3. \\
\gamma' &= \dagger_1 \langle_1 T \rangle_1 \dagger_1 \dagger_{\bar{1}} \langle_{\bar{1}} F \rangle_{\bar{1}} \dagger_{\bar{1}} \dagger_2 \langle_2 T \rangle_2 \dagger_2 \dagger_{\bar{2}} \langle_{\bar{2}} F \rangle_{\bar{2}} \dagger_{\bar{2}}.
\end{aligned}$$

We use the next notations:

$$\begin{aligned}
\gamma &= \gamma_1\gamma_2 \ldots \gamma_n \text{ where } \gamma_i = \gamma_i^{(1)}\gamma_i^{(0)}, \\
&\quad \gamma_i^{(1)} = \dagger_i \langle_i T \rangle_i \dagger_i, \gamma_i^{(0)} = \dagger_{\bar{i}} \langle_{\bar{i}} F \rangle_{\bar{i}} \dagger_{\bar{i}}. \\
\gamma_{i,n} &= \gamma_i\gamma_{i+1} \ldots \gamma_n. \\
\gamma_{i,j}^{(b_{i,j})} &= \gamma_i^{(b_i)}\gamma_{i+1}^{(b_{i+1})} \ldots \gamma_j^{(b_j)}, b_{i,j} = b_i b_{i+1} \ldots b_j \in \{0,1\}^+, 1 \leq i < j \leq n, \\
&\quad \gamma_{1,1} = \gamma_1, \gamma_{n,n} = \gamma_n, \gamma_{1,0} = \lambda, \gamma_{n+1,n} = \lambda. \\
\Gamma &\in Sub(\gamma). \\
\alpha_{i,n} &= f_i f_{i+1} \ldots f_n, \\
\bar{\alpha}_{i,n} &= f_n f_{n-1} \ldots f_i, 0 \leq i \leq n, \ \alpha_{n+1,n+1} = \bar{\alpha}_{n+1,n+1} = f_{n+1}. \\
\sigma(i_1) &= i, \ \sigma(i_0) = \bar{i}.
\end{aligned}$$

The input string $\pi_0$ (we call it in-string) containing the encoded propositional formula $\alpha$ is of the form:

$$\pi_0 = \mathtt{YE}\bar{\alpha}_{1,n+1} f_0 \alpha_{1,n+1} \beta \bar{\alpha}_{1,n+1} f_0 \alpha_{1,n+1} \mathtt{ES}.$$

The size of the input is polynomial, $|\pi_0| \leq 4nm + 14n + 3m + 12$, hence it is constructible by a Turing machine.

Roughly speaking, the main idea of the algorithm by which the AIR system solves SAT is as following: (i) We generate all possible truth-assignments

on the in-string according to the variables of the given instance of the propositional formula, while the encoded propositional formula with the attached $\Gamma$ is replicated, in a linear time. (ii) Then each truth-assignment is assigned to its attached formula in the maximally parallel way. (iii) Following this step, we check the satisfiability of the formula with regard to the truth-assignments. (iv) Finally, the AIR system decides to accept or not the input string $\pi_0$. We stress here that the computation is guided very much by the contexts of rules to be applied.

Let us start the computation accomplishing the above steps.

The generation of all possible truth-assignments takes $3n$ steps.

*(i) Generating truth-value assignments:*

In order to generate all possible truth-assignments, we use a combination of three modules, which are applied one after other in a cyclic order: $\mathsf{repl}_{f_i} \Longrightarrow \mathsf{del}_{\dagger_i} \Longrightarrow \mathsf{del}_{f_i}$. We emphasize here that the repeated pointers used in each $\mathsf{del}$ and $\mathsf{repl}$ are the closest to each other.

1. The replication operation $\mathsf{repl}_{f_i}$ duplicates the substring $t_i$ flanked by the repeats of $f_i$.

$$\mathsf{repl}_{f_i} : \ \alpha f_i s_i f_i \alpha' \Longrightarrow^{max}_{\mathsf{repl}_{f_i}} \alpha f_i s_i s_i f_i \alpha', \ \alpha, \alpha' \in \Sigma^*, 0 \leq i \leq n-1.$$

The contexts for the replication operation to be applied are

- $(\bar{\alpha}_{i+1,n+1}, f_i, s_i) \sim (s_i, f_i, \alpha_{i+1,n+1})$
  where $s_i = \alpha_{i+1,n+1}\delta(\gamma_{1,i}^{(b_{1,i})}\gamma_{i+1,n})^m\bar{\alpha}_{i+1,n+1}$.

At each $3k-2$th $(1 \leq k \leq n)$ step, the in-string contains substrings of the form:

$$f_{n+1}f_n \cdots f_{i+1}\underline{f_i}f_{i+1} \cdots f_n f_{n+1}\delta(\gamma_1^{(b_1)} \cdots \gamma_i^{(b_i)}\gamma_{i+1} \cdots \gamma_n)^m$$
$$f_{n+1}f_n \cdots f_{i+1}\underline{f_i}f_{i+1} \cdots f_n f_{n+1}.$$

Each substring flanked by $f_i$ contains only one $\Gamma$. The only possible rule to be applied to this substring is $\mathsf{repl}_{f_i}$ for the repeats $f_i, i = 3k-3$ (no other rules contexts are satisfied on the string). No replication using $\mathsf{repl}_{f_j}, j \neq 3k-3, j \geq 0$, may occur, because no $f_j, j < 3k-3$, has remained in the string; on the other hand, at this moment there is no pattern of the form $f_{j+1}f_jf_{j+1}$, $j > 3k-3$, to which $\mathsf{repl}_{f_j}$ is applicable. No deletion using $\mathsf{del}_{f_i}, i \geq 1$, may occur; the only possible pattern to which $\mathsf{del}_{f_{i+1}}$ could apply is $f_{i+1}f_if_{i+1}$, but $\mathsf{del}_{f_{i+1}}$ asks the substring $\gamma_{i+1}$ has to be evolved as $\gamma_{i+1}^{(b^{i+1})}$, it is not the case at the moment. Moreover, the other rules of type $\mathsf{ctrl}$ are not possible to be applied here.

Then in this step, only $\mathsf{repl}_{f_i}$ can be applied in the maximally parallel way to the string. Each substring flanked by $f_i$ is replicated as follows.

$$\underline{f_i} f_{i+1} \cdots f_{n+1} \delta (\gamma_{1,i}^{(b_{1,i})} \gamma_{i+1,n})^m f_{n+1} \cdots$$
$$f_{i+1} f_{i+1} \cdots f_{n+1} \delta (\gamma_{1,i}^{(b_{1,i})} \gamma_{i+1,n})^m f_{n+1} \cdots f_{i+1} \underline{f_i}.$$

Now we check which operation among $\mathsf{repl}_{f_i}$, $\mathsf{del}_{f_i}$, $\mathsf{del}_{\dagger_i}$, $\mathsf{del}_{\dagger_{\bar{i}}}$, $\mathsf{ctrl}$ is applicable to the in-string. The replication $\mathsf{repl}_{f_i}$ cannot be repeated immediately because more than one $\delta$s are between two repeats of $f_i$, which contradicts the context of $\mathsf{repl}_{f_i}$. Moreover, no deletion by $\mathsf{del}_{f_i}$ is possible at this step, because $\gamma_i$ has not been evolved yet. Only the rules $\mathsf{del}_{\dagger_i}$ and $\mathsf{del}_{\dagger_{\bar{i}}}$ are applicable to the current in-string.

2. It is the $3k-1$th $(1 \leq k \leq n)$ step. The deletion $\mathsf{del}_{\dagger}$ is applied in the maximally parallel way on the $2^k$ copies of $\Gamma$. By deletion $\mathsf{del}_{\dagger_i}$ (resp. $\mathsf{del}_{\dagger_{\bar{i}}}$), the truth-values $\langle_i T\rangle_i$ (resp. $\langle_{\bar{i}} F\rangle_{\bar{i}}$) are deleted wherever the contexts of $\mathsf{del}_{\dagger}$ are satisfied.

$$\alpha \dagger_{\sigma(i_b)} \langle_{\sigma(i_b)} B\rangle_{\sigma(i_b)} \dagger_{\sigma(i_b)} \alpha' \Longrightarrow^{max}_{\mathsf{del}_{\dagger_{\sigma(i_b)}}} \alpha \dagger_{\sigma(i_b)} \alpha',$$
$$\alpha, \alpha' \in \Sigma^*, B \in \{T, F\}, 1 \leq i \leq n.$$

$\mathsf{del}_{\dagger_{\bar{i}}}$ : $(f_{i-1}\alpha_{i,n+1}\delta(\gamma_{1,i-1}^{(b_{1,i-1})}\gamma_{i,n})^{j-1}\gamma_{1,i-1}^{(b_{1,i-1})}\gamma_i^1, \dagger_{\bar{i}}, \langle_{\bar{i}} F\rangle_{\bar{i}}) \sim$
$(\langle_{\bar{i}} F\rangle_{\bar{i}}, \dagger_{\bar{i}}, \gamma_{i+1,n}(\gamma_{1,i-1}^{(b_{1,i-1})}\gamma_{i,n})^{m-j}\bar{\alpha}_{i,n+1}\alpha_{i,n+1})$
where $b_i \in \{0,1\}, 1 \leq j \leq m, 1 \leq i \leq n, \gamma_{1,1} = \gamma_1, \gamma_{n,n} = \gamma_n, \gamma_{1,0} = \lambda, \gamma_{n+1,n} = \lambda$.

The context says that the substring $\dagger_{\bar{i}}\langle_{\bar{i}} F\rangle_{\bar{i}}\dagger_{\bar{i}}$, which is going to be deleted, has to be preceded by a string which is bordered by $f_{i-1}$ in the left side, and followed by a string of the form $f_{n+1} \cdots f_i f_i \cdots f_{n+1}$. One more requirement is that all $\gamma_j, j < i$, have to be broken already and $\gamma_j, j > i$, have not been processed up to now. $\mathsf{del}_{\dagger}$ applies to $m$ copies of $\Gamma$ attached to $\delta$, namely, it applies to all $\Gamma$ in the maximally parallel way on the in-string which satisfies the contexts.

$\mathsf{del}_{\dagger_i}$ : $(\bar{\alpha}_{i,n+1}\alpha_{i,n+1}\delta(\gamma_{1,i-1}^{(b_{1,i-1})}\gamma_{i,n})^{j-1}\gamma_{1,i-1}^{(b_{1,i-1})}, \dagger_i, \langle_i T\rangle_i) \sim$
$(\langle_i T\rangle_i, \dagger_i, \gamma_i^0\gamma_{i+1,n}(\gamma_{1,i-1}^{(b_{1,i-1})}\gamma_{i,n})^{m-j}\bar{\alpha}_{i,n+1}f_{i-1}),$
where $b_i \in \{0,1\}, 1 \leq j \leq m, 1 \leq i \leq n, \gamma_{1,1} = \gamma_1, \gamma_{n,n} = \gamma_n, \gamma_{1,0} = \lambda, \gamma_{n+1,n} = \lambda$.

A substring $\dagger_i\langle_i T\rangle_i\dagger_i$ with its preceding substring bordered by $f_{i-1}$ in the right side is deleted by $\mathsf{del}_{\dagger_i}$.

Up to now, $2^i$ truth-assignments of the form

$$\gamma_1^{(b_1)}\gamma_2^{(b_2)}\dots\gamma_i^{(b_i)}\gamma_{i+1}\dots\gamma_n, b_i \in \{0,1\}$$

have been generated on the in-string.

3. At the $3k-1$th $(1 \leq k \leq n)$ step of the computation, the deletion operation $\mathsf{del}_{f_i}$ is allowed. It applies for the repeats of $f_i$ and deletes one copy of $f_i$ with one $f_{i-1}$ if it is available between those $f_i$.

$$\mathsf{del}_{f_i}: \ \alpha f_i f_{i-1} f_i \alpha' \Longrightarrow^{max}_{\mathsf{del}_{f_i}} \alpha f_i \alpha',$$

$\alpha, \alpha' \in \Sigma^*, 1 \leq i \leq n.$

The next two contexts are in the splicing scheme for $\mathsf{del}_{f_i}$:

- $(\bar{\alpha}_{i+1,n+1}, f_i, f_{i-1}) \sim (f_{i-1}, f_i, \alpha_{i+1,n+1}\delta(\gamma_{1,i}^{b_{1,i}}\gamma_{i+1,n})^m),$
- $(\delta(\gamma_{1,i}^{(b_{1,i})}\gamma_{i+1,n})^m\bar{\alpha}_{i+1,n+1}, f_i, f_{i-1}) \sim (f_{i-1}, f_i, \alpha_{i+1,n+1}).$

Remember that at the previous two steps $\mathsf{del}_{f_i}$ was not applicable because $\gamma_i$ was not operated. Since this was done at the previous step by $\mathsf{del}_{\dagger_i}$, now $\mathsf{del}_{f_i}$ can be applied. Note that the subscripts are shifted from $i$ to $i-1$ here.

$$\mathsf{del}_{f_i}: \ \alpha f_i f_i \alpha' \Longrightarrow^{max}_{\mathsf{del}_{f_i}} \alpha f_i \alpha', \text{ where}$$

- $(\delta(\gamma_{1,i}^{(b_{1,i})}\gamma_{i+1,n})^m\bar{\alpha}_{i+1,n+1}, f_i, f_i) \sim (f_i, f_i, \alpha_{i+1,n+1})$
  where $\alpha_{n+1,n+1} = \bar{\alpha}_{n+1,n+1} = f_{n+1}.$

As the truth-assignments are generated completely, the cyclic iteration $(1) \Longrightarrow (2) \Longrightarrow (3)$ is ended at the $3n$th step of the computation after $n$ repeats. Now the in-string is of the form:

$$\pi' = \mathrm{YE}(f_{n+1}f_nf_{n+1}\delta\Gamma^m f_{n+1}f_nf_{n+1})^{2^n}\mathrm{ES}.$$

*(ii) Assigning the truth-values to the variables.*

Now the truth-values (truth-assignments) are assigned to each variable (to each clause).

$$\alpha'\langle_{\sigma(i_b)}x_{\sigma(i_b)}\rangle_{\sigma(i_b)}\alpha''\langle_{\sigma(i_b)}B\rangle_{\sigma(i_b)}\alpha''' \Longrightarrow^{max}_{\mathsf{ctrl}_{\langle_{\sigma(i_b)},\rangle_{\sigma(i_b)}}}$$
$$\alpha'\langle_{\sigma(i_b)}B\rangle_{\sigma(i_b)}\alpha''\langle_{\sigma(i_b)}x_{\sigma(i_b)}\rangle_{\sigma(i_b)}\alpha''', \text{ where}$$

$b \in \{0,1\}, \sigma(i_1) = i, \sigma(i_0) = \bar{i}, 1 \leq i \leq n, \ B \in \{T,F\}, \ \alpha', \alpha'', \alpha''' \in \Sigma^*,$
$(\vee, \langle_{\sigma(i_b)}, x_{\sigma(i_b)}\rangle_{\sigma(i_b)} \vee u\dagger_{\sigma(i_b)}) \sim (x_{\sigma(i_b)}\rangle_{\sigma(i_b)} \vee u\dagger_{\sigma(i_b)}, \langle_{\sigma(i_b)}, B\rangle_{\sigma(i_b)}\dagger_{\sigma(i_b)}v)$
and

$(\langle_{\sigma(i_b)} x_{\sigma(i_b)}, \rangle_{\sigma(i_b)}, \vee u \dagger_{\sigma(i_b)} \langle_{\sigma(i_b)} B) \sim (\vee u \dagger_{\sigma(i_b)} \langle_{\sigma(i_b)} B, \rangle_{\sigma(i_b)}, \dagger_{\sigma(i_b)} v)$, where $u \neq u' f_{n+1} u''$, $v \neq v' f_{n+1} v''$, $u', u'', v', v'' \in \Sigma^*$.

By the application of $\mathsf{ctrl}_{\langle_i,\rangle_i}$, every variable $x_i$ flanked by $\langle_i$ and $\rangle_i$ in each clause $c_j$ and the corresponding truth-value $T$ (*true*) flanked by $\langle_i$ and $\rangle_i$ are swapped if such correspondence exist. Similarly, for assigning the truth-value $F$ (*false*) to $x_{\bar{i}}$, the contents of $\langle_{\bar{i}} x_{\bar{i}} \rangle_{\bar{i}}$ and $\langle_{\bar{i}} F \rangle_{\bar{i}}$ are swapped by $\mathsf{ctrl}_{\langle_{\bar{i}},\rangle_{\bar{i}}}$. Thus, all truth-assignments are assigned to their attached propositional formula at the same time. The truth-values assigned to a fixed formula should not be mixed from the different assignments, that is why the constraints $u \neq u' f_{n+1} u''$, $v \neq v' f_{n+1} v''$ are asked in the context.

*(iii) Checking the satisfiability of the propositional formula.*

If a clause $c_k$ is satisfied by the corresponding truth-assignment, then at least one substring of type $\langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)}$ exist in $c_k$ as $\$_k x \langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)} y \$_k$. However, if the propositional formula $\alpha$ is satisfied by a truth-assignment, then each clause $c_k$ of $\alpha$ is of the form $\$_k x \langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)} y \$_k, 1 \leq k \leq m$. Then a formula satisfied by an assignment is of the form:

$$\$_0 \$_1 x \langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)} y \$_1 \$_2 x \langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)} y \$_2 \ldots \$_m x \langle_{\sigma(i_b)} B \rangle_{\sigma(i_b)} y \$_m \$_{m+1},$$

where $b \in \{0,1\}, \sigma(i_1) = i, \sigma(i_0) = \bar{i}, 1 \leq i \leq n$, $B \in \{T, F\}$, for some $x, y, u, v \in \Sigma^*$.

The deletion operation $\mathsf{del}_{\$_i}$ applies to the string in a maximally parallel way, and deletes the clauses which contain at least a truth-value, $\$_i \alpha' \langle_{\sigma(j_b)} B \rangle_{\sigma(j_b)} \alpha'' \$_i$.

$$\mathsf{del}_{\$_i} : \alpha \$_i \alpha' \langle_{\sigma(j_b)} B \rangle_{\sigma(j_b)} \alpha'' \$_i \alpha''' \Longrightarrow^{max}_{\mathsf{del}_{\$_i}} \alpha \$_i \alpha''', \text{ where}$$

- $(\$_{i-1}, \$_i, x \langle_{\sigma(j_b)} B \rangle_{\sigma(j_b)} y) \sim (x \langle_{\sigma(j_b)} B \rangle_{\sigma(j_b)} y, \$_i, \$_{i+1})$, $1 \leq i \leq m, 1 \leq j \leq n$.

*(iv) Deciding.*

In the end of the computation, we obtain some sequences of the form $\$_0 \$_1 \$_2 \ldots \$_m \$_{m+1}$ on the in-string if there exist truth-assignments which satisfy the formula $\alpha$.

Then $\mathsf{del}_{\mathtt{E}}$ is applicable to the in-string $\mathtt{YE}u\$_1\$_2 \ldots \$_{m-1}\$_m v\mathtt{ES}$ for the repeats $\mathtt{E}$ at the $3n+3$th step and we get the axiom.

$$\mathtt{YE}u\$_1\$_2 \ldots \$_{m-1}\$_m v\mathtt{ES} \Longrightarrow_{\mathsf{del}_{\mathtt{E}}} \mathtt{YES}, \text{ where}$$

- $(\mathtt{Y}, \mathtt{E}, \$) \sim (\$, \mathtt{E}, \mathtt{S})$, $\$ = u\$_1\$_2 \ldots \$_{m-1}\$_m v$, for some $u, v \in \Sigma^*$.

If no sequence $\$_1\$_2 \ldots \$_m$ is obtained in the in-string, then the computation just halt at the $3n+2$th step since no rule is possible to apply from now on, hence, $\pi_0$ is not accepted by $G$. Thus, the problem $\alpha$ is solved in a linear time. □

Mathematically, the AIR systems are elegant because only one string and a very restricted number of operations (two or three) are involved. The context-sensitivity for string operations are already well-known and much investigated in formal language theory, see [63, 78, 56]. The operations ctrl and del of AIR systems are based on dlad and ld which are formalizations of gene assembly in ciliates, hence, our recombination operations are well motivated from biology. Moreover, we use parallel application of rules when we solve SAT, and the parallelism is a feature characteristic of bio-inspired computing models, starting with DNA computing. From computer science point of view, AIR systems are both as powerful as Turing machines, and, in the extended variant, they solve intractable problems in feasible time. The idea of the algorithm solving SAT by AIR is clearly applicable to the guided intermolecular recombination systems, see Definition 6.1. However, the possibility of generating languages using these rules remains be investigated.

A research topic for link membrane computing and ciliate computing is formulated by Gh. Păun in [76] as follows.

*Links P systems with Ciliates.* One of the most intriguing "computations" held in nature is that done by ciliates, during their reproduction. The permutations of genes in macronuclei–micronuclei are exquisite list processing operations of a surprising power and complexity. Ciliates are unicellular organisms. Up to now they were interpreted as single membrane structures, which is not exactly the case in reality. What about "marrying" membrane computing and "ciliate computing"? The first suggestion is to use ciliate–inspired operations with strings in handling string–objects in P systems. Which combinations of operations (in what kinds of membrane structures) lead to universality? Then, a question of a possible interest from a biological point of view is to have a look at the ciliate structure, distinguishing the compartments of their cell–body and the specific operations (including the way of communicating among compartments), and to define a suitable model of this structure. The third issue of interest is to use ciliate-like P systems as computing devices, adding parallelism (and/or further ways to get speeding-up features, such as an exponential workspace) in such a way to solve computationally hard problems in a feasible (polynomial) time. This last idea could hopefully be related to the possibility to "implement" ciliate-P-systems in real–ciliates, thus dreaming at using ciliates as living computers.

**Example 6.3**

$$
\begin{aligned}
\varepsilon &= (x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2).\\
\varepsilon' &= \$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3.\\
\gamma &= \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2.\\
\pi_0 &= \mathtt{YE}\bar{\alpha}_{1,3}f_0\alpha_{1,3}\delta\gamma^2\bar{\alpha}_{1,3}f_0\alpha_{1,3}\mathtt{ES}\\
&= \mathtt{YE}f_3f_2f_1f_0f_1f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2f_3f_2f_1f_0f_1f_2f_3\mathtt{ES}.
\end{aligned}
$$

*Step* 1 :

$$
\begin{aligned}
\pi_0' &= \mathsf{repl}_{f_0}(\pi_0),\ (\bar{\alpha}_{1,3}, f_0, s_0) \sim (s_0, f_0, \alpha_{1,3}),\ s_0 = \alpha_{1,3}\delta(\gamma_1\gamma_2)^2\bar{\alpha}_{1,3}.\\
\pi_0' &= \mathtt{YE}\bar{\alpha}_{1,3}f_0\alpha_{1,3}\delta\gamma^2\bar{\alpha}_{1,3}\alpha_{1,3}\delta\gamma^2\bar{\alpha}_{1,3}f_0\alpha_{1,3}\mathtt{ES}\\
&= \mathtt{YE}f_3f_2f_1\underline{f_0}f_1f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2f_1f_1f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2f_3f_2f_1\underline{f_0}f_1f_2f_3\mathtt{ES}.
\end{aligned}
$$

*Step* 2 :

$$
\begin{aligned}
\pi_0'' &= \mathsf{del}_{\dagger_1}\mathsf{del}_{\bar{\dagger}_1}(\pi_0')\\
\pi_0'' &= \mathtt{YE}\bar{\alpha}_{1,3}f_0\alpha_{1,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2)^2\bar{\alpha}_{1,3}\alpha_{1,3}\delta(\dagger_1\gamma_1^0\gamma_2)^2\bar{\alpha}_{1,3}f_0\alpha_{1,3}\mathtt{ES}\\
&= \mathtt{YE}f_3f_2f_1\underline{f_0}f_1f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2f_1f_1f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2f_1\underline{f_0}f_1f_2f_3\mathtt{ES}.
\end{aligned}
$$

*Step* 3 :

$$
\begin{aligned}
\pi_1 &= \mathsf{del}_{f_1}(\pi_0'')\\
\pi_1 &= \mathtt{YE}\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2)^2\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\gamma_2)^2\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\mathtt{ES}\\
&= \mathtt{YE}f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2\underline{f_1}f_2f_3\mathtt{ES}.
\end{aligned}
$$

*Step* 4 :

$$\begin{aligned}
\pi_1' &= \mathsf{repl}_{f_1}(\pi_1)\\
\pi_1' &= \mathtt{YE}\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2)^2\bar{\alpha}_{2,3}\\
&\quad \underline{f_1}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\gamma_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\gamma_2)^2\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\mathtt{ES}\\
&= \mathtt{YE}f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2\underline{f_1}f_2f_3\mathtt{ES}.
\end{aligned}$$

*Step* 5 :

$$\begin{aligned}
\pi_1'' &= \mathsf{del}_{\dagger_2}\mathsf{del}_{\bar{\dagger}_2}(\pi_1')\\
\pi_1'' &= \mathtt{YE}\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2^1\bar{\dagger}_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\dagger_2\gamma_2^0)^2\bar{\alpha}_{2,3}\\
&\quad \underline{f_1}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\gamma_2^1\bar{\dagger}_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\dagger_2\gamma_2^0)^2\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\mathtt{ES}.\\
&= \mathtt{YE}f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\\
&\quad f_3f_2f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\\
&\quad f_3f_2\underline{f_1}f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\\
&\quad f_3f_2f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\bar{\langle}_1\mathtt{A}\bar{\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\bar{\langle}_2\mathtt{A}\bar{\rangle}_2\bar{\dagger}_2f_3f_2\underline{f_1}f_2f_3\mathtt{ES}.
\end{aligned}$$

*Step* 6 :

$$\begin{aligned}
\pi_2 &= \mathsf{del}_{f_2}(\pi_1'')\\
\pi_2 &= \mathtt{YE}f_3f_2f_3\delta(\gamma_1^1\bar{\dagger}_1\gamma_2^1\bar{\dagger}_2)^2f_3f_2f_3\delta(\gamma_1^1\bar{\dagger}_1\dagger_2\gamma_2^0)^2f_3f_2f_3\\
&\quad \delta(\dagger_1\gamma_1^0\gamma_2^1\bar{\dagger}_2)^2f_3f_2f_3\delta(\dagger_1\gamma_1^0\dagger_2\gamma_2^0)^2f_3f_2f_3\mathtt{ES}\\
&= \mathtt{YE}f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \bar{\langle}_1\mathtt{C}\bar{\rangle}_1 \vee \bar{\langle}_2\mathtt{C}\bar{\rangle}_2 \vee \$_2\$_3\\
&\quad \dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2
\end{aligned}$$

$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2f_3f_2f_3\mathtt{ES}.$$

*Step* 7 :

$$\pi_3 = \mathsf{ctrl}_{\langle_i,\rangle_i}\mathsf{ctrl}_{\overline{\langle_i},\overline{\rangle}_i}(\pi_2),\ i=1,2.$$

$$\pi_3 = \mathtt{YE}\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\gamma_2^1\bar{\dagger}_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\gamma_1^1\bar{\dagger}_1\dagger_2\gamma_2^0)^2\bar{\alpha}_{2,3}$$
$$\underline{f_1}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\gamma_2^1\bar{\dagger}_2)^2\bar{\alpha}_{2,3}\alpha_{2,3}\delta(\dagger_1\gamma_1^0\dagger_2\gamma_2^0)^2\bar{\alpha}_{2,3}\underline{f_1}\alpha_{2,3}\mathtt{ES}.$$

$$= \mathtt{YE}f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{T}\rangle_1 \vee \langle_2\mathtt{T}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\langle_1\mathtt{G}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{G}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{T}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{A}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\langle_1\mathtt{G}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{C}\rangle}_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{T}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{A}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{G}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{C}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2 \vee \overline{\langle_1\mathtt{A}\rangle}_1 \vee \overline{\langle_2\mathtt{A}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{C}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{C}\rangle}_2\bar{\dagger}_2f_3f_2f_3\mathtt{ES}.$$

*Step* 8 :

$$\pi_4 = \mathsf{del}_{\$_i}(\pi_3),\ (\$_{i-1},\$_i,x\langle_{\sigma(j_b)}B\rangle_{\sigma(j_b)}y) \sim (x\langle_{\sigma(j_b)}B\rangle_{\sigma(j_b)}y,\$_i,\$_{i+1}),$$
$$B\in\{\mathtt{A},\mathtt{T}\},\ i,j=1,2.$$

$$\pi_4 = \mathtt{YE}f_3f_2f_3\$_0\$_1\$_2 \vee \overline{\langle_1\mathtt{C}\rangle}_1 \vee \overline{\langle_2\mathtt{C}\rangle}_2 \vee \$_2\$_3$$
$$\dagger_1\langle_1\mathtt{G}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{G}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1\$_2\$_3\dagger_1\langle_1\mathtt{G}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\langle_1\mathtt{T}\rangle_1\dagger_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{C}\rangle}_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1\$_2\$_3\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{G}\rangle_2\dagger_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{C}\rangle}_1\bar{\dagger}_1\dagger_2\langle_2\mathtt{T}\rangle_2\dagger_2\bar{\dagger}_2$$
$$f_3f_2f_3\$_0\$_1 \vee \langle_1\mathtt{G}\rangle_1 \vee \langle_2\mathtt{G}\rangle_2 \vee \$_1\$_2\$_3$$
$$\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{A}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{A}\rangle}_2\bar{\dagger}_2\dagger_1\bar{\dagger}_1\overline{\langle_1\mathtt{C}\rangle}_1\bar{\dagger}_1\dagger_2\bar{\dagger}_2\overline{\langle_2\mathtt{C}\rangle}_2\bar{\dagger}_2f_3f_2f_3\mathtt{ES}.$$

*Step* 9 :

$$\mathsf{del}_{\mathtt{E}}(\pi_4) = \mathtt{YES}.$$

# Bibliography


[1] Adleman, L. M., Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266 (1994), 1021–1024.

[2] Adleman, L. M., On Constructing a Molecular Computer. *1st DIMACS Workshop on DNA Based Computers*, Princeton, 1995. In *DIMACS Series*, 27 (1996), 1–21.

[3] Alhazov, A., *Communication in Membrane Systems with Symbol Objects.* PhD thesis, University of Rovira i Virgili, 2006.

[4] Alhazov, A., Ishdorj, T.-O., Membrane Operations in P Systems with Active Membranes. In [85], 37–44.

[5] Alhazov, A., Pan, L., Păun, Gh., Trading Polarizations for Labels in P Systems with Active Membranes. *Acta Informaticae*, 41, 2-3 (2004), 111–144.

[6] Baum, E., Landweber, L., eds., DNA Based Computers. *Proceedings Second Annual Meeting.* Vol. 44 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, AMS, 1996.

[7] Benenson, Y., Adar, R., Paz-Elizur, T., Livneh, Z., Shapiro, Eh., DNA Molecule Provides a Computing Machine with Both Data and Fuel. *Proc. Nat. Acad. Sci. (PNAS)*, 100, 5 (2003), 2191–2196.

[8] Benenson, Y., Paz-Elizur, T., Adar, R., Keinan, Eh., Livneh, Z., Shapiro, Eh., Programmable and Autonomous Computing Machine Made of Biomolecules. *Nature*, 414 (2001), 430–434.

[9] Bennett, C. H., Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17 (1973), 525–532.

[10] Bennett, C. H., Thermodynamics of Computation – a Review. *International Journal of Theoretical Physics*, 21 (1982), 905–940.

[11] Calude, C. S., Păun, Gh., *Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing.* Taylor and Francis, London, 2000.

[12] Calude, C., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.), *Multiset Processing.* LNCS 2235, Springer, Berlin, 2001.

[13] Cardelli, L., Brane Calculi. Interactions of Biological Membranes. In [30], 257–278.

[14] Cavaliere, M., *Evolution, Communication, Observation: From Biology to Membrane Computing and Back.* PhD thesis, University of Sevilla, 2006.

[15] Cavaliere, M., Ionescu, M., Ishdorj, T.-O., Inhibiting/De-inhibiting Rules in P Systems. In [65], 60–73, and *Lecture Notes in Computer Science* LNCS 3365, 224–238, Springer, Berlin, 2005.

[16] Cavaliere, M., Ionescu, M., Ishdorj, T.-O, Inhibiting/De-inhibiting Rules in P Systems with Active Membranes. In *Proc. Cellular Computing (Complexity Aspects) ESF PESC Exploratory Workshop* (M. A. Gutiérrez-Naranjo, Gh. Păun, M. J. Pérez-Jiménez, Eds.), Sevilla, January 31 – February 2, 2005, 117–130, Fénix Editora, Sevilla, 2005.

[17] Ceterchi, R., Sburlan, D., Simulating Boolean Circuits with P Systems. *Workshop on Membrane Computing WMC*, Tarragona, Spain, 2003, (C. Martín-Vide, Gh. Păun, G. Rozenberg, A. Salomaa, eds), LNCS 2933, 104–122, 2004.

[18] Chen, H., Freund, R., Ionescu, M., Păun, Gh., Pérez-Jiménez, M. J., On String Languages Generated by Spiking Neural P Systems. In [39], Vol. I, 169–194.

[19] Chen, H., Ionescu, M., Ishdorj, T.-O., On the Efficiency of Spiking Neural P Systems. In [39], Vol. I, 195–206, and *Proc. 8th International Conference on Electronics, Information, and Communication (ICEIC2006)*, Ulaanbaatar, June 2006, 49–52.

[20] Chen, H., Ishdorj, T.-O., Păun, Gh., Computing Along the Axon. In [39], Vol. I, 225–240, and *Pre-proceedings of International Conference on Bio-inspired Computing: Theory and Applications (BIC-TA)*, September 2006, Wuhan, China, 60–70.

[21] Chen, H., Ishdorj, T.-O., Păun, Gh., Pérez-Jiménez, M., Spiking Neural P Systems with Extended Rules. In [39], Vol. I, 241–265, and *Romanian Journal of Information Sceince and Technology*, 9 (2006), 151–162.

[22] Chen, H., Ionescu, M., Ishdorj, T.-O., Păun, A., Păun, Gh., Pérez-Jiménez, M., Spiking Neural P Systems with Extended Rules: Universality and Languages. *Natural Computing*, to appear.

[23] Chomsky, N., Three Models for the Description of Languages. *IRE Transactions on Information Theory*, 2: (1956) 113–124.

[24] Ciobanu, G., Pan, L., Păun, Gh., Pérez-Jiménez, M. J., P Systems with Minimal Parallelism. *Theoretical Computer Science,* to appear.

[25] Ciobanu, G., Păun, Gh., Perez-Jimenez, M. J., eds, *Applications of Membrane Computing.* Springer, Berlin, 2006.

[26] Crick, F., *Of Molecules and Men.* Great Minds Series, University of Washington Press, 1966.

[27] Crick, F., Watson, J. D., A Structure of Deoxyribonucleic Acid. *Nature*, Vol. 171, (1953), 737–738.

[28] Csuhaj-Varju, E., Freund, R., Kari, L., Păun, Gh., DNA Computing Based on Splicing: Universality Results. *Proc. First Annual Pacific Symp. on Biocomputing*, Hawaii, 1996 (L. Hunter, T. E. Klein, eds.), World Scientific, Singapore, 1996, 179–190.

[29] Daley, M., Kari, L., Some Properties of Ciliate Bio-operations. *DLT 2002*, LNCS 2450, Springer, Berlin, (2003) 116–127.

[30] Danos, V., Schachter, V. eds., *Computational Methods in Systems Biology, International Conference CMSB 2004.* Paris, France, 2004, LNCS 3082, Springer, Berlin, 2005.

[31] Dassow, J., Păun, Gh., *Regulated Rewriting in Formal Language Theory.* Springer, Berlin, 1989.

[32] Ehrenfeucht, A., Harju, T., Petre, I., Prescott, D. M., Rozenberg, G., *Computation in Living Cells: Gene Assembly in Ciliates.* Springer, Berlin, 2003.

[33] Ehrenfeucht, A., Petre, I., Prescott, D. M., Rozenberg, G., Universal and Simple Operations for Gene Assembly in Ciliates. In: V. Mitrana, C. Martin-Vide (eds.) *Words, Sequences, Languages: Where Computer Science, Biology and Linguistics Meet*, Kluwer Academic, Dortrecht, (2001) 329–342.

[34] Ehrenfeucht, A., Prescott, D. M., Rozenberg, G., Computational Aspects of Gene (Un)Scrambling in Ciliates. In: L. F. Landweber, E. Winfree (eds.) *Evolution as Computation*, Springer, Berlin, Heidelberg, New York (2001) 216–256.

[35] Feynman, R. P., There's Plenty of Room at the Bottom. H. D. Gilbert ed., *Miniaturization.* Reinhold, New York, 1961, 282–286.

[36] Galiukschov, B. S., Semicontextual Grammars. *Mathematika Logica i Matematika Linguistika*, Talinin University (1981), 38–50 (in Russian).

[37] Garey, M., Johnson, D., *Computers and Intractability. A Guide to the Theory of NP-completeness.* Freeman, San Francisco, CA, 1979.

[38] Gutiérrez-Naranjo, M. A., Păun, Gh., Riscos-Núñez, A., Romero-Campero, F. J., Sburlan, D., eds., *Proceedings of the Third Brainstorming Week on Membrane Computing.* Seville, 2005, GCN Technical Report 01/2005, University of Seville, 2005.

[39] Gutiérrez-Naranjo, M. A., Păun, Gh., Riscos-Núñez, A., Romero-Campero, F. J., eds., *Proceedings of the Fourth Brainstorming Week on Membrane Computing.* Vol. I, Seville, 2006, Fenix Editora, Sevilla, 2006.

[40] Gutiérrez-Naranjo, M. A., Păun, Gh., Riscos-Núñez, A., Romero-Campero, F. J., eds., *Proceedings of the Fourth Brainstorming Week on Membrane Computing.* Vol. II, Seville, 2006, Fenix Editora, Sevilla, 2006.

[41] Harju, T., Petre, I., Li, C., Rozenberg, G., Parallelism in Gene Assembly. In: *Natural Computing*, to appear, 2006.

[42] Harju, T., Petre, I., Rozenberg, G., Gene Assembly in Ciliates: Molecular Operations. In: Păun, Gh., Rozenberg, G., Salomaa, A., (Eds.) *Current Trends in Theoretical Computer Science*, World Scientific, 2004.

[43] Harju, T., Petre, I., Rozenberg, G., Two Models for Gene Assembly in Ciliates. LNCS 3113, Springer, Berlin, 2004, 89–101.

[44] Head, T., Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors. *Bull. Math. Biology* 49: 737–759, 1987.

[45] Head, T., Splicing Schemes and DNA. In *Lindenmayer systems* (Rozenberg, G., Salomaa, A., Eds.) Springer, Berlin, 295–358.

[46] Head, T., Aqueous Simulations of Membrane Computations. *Romanian Journal of Informatics Science and Technology*, 5, 4 (2002).

[47] Hopcroft, J. E., Ullman, J. D., *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading MA, 1979.

[48] Ionescu, M., Ishdorj, T.-O., Boolean Circuits and a DNA Algorithm in Membrane Computing. In *Proc. of the 6th Workshop on Membrane Computing*, Vienna, Austria, 18–23 July 2005, and LNCS 3850, Springer, Berlin, 2006, 274–293.

[49] Ionescu, M., Păun, Gh., Yokomori, T., Spiking Neural P Systems. *Fundamenta Informaticae*, 71, 2–3 (2006), 279–308.

[50] Ishdorj, T.-O., Power and Efficiency of Minimal Parallelism in Polarizationless P Systems. *Journal of Automata, Languages, and Combinatorics*, to appear, 2006.

[51] Ishdorj, T.-O., Minimal Parallelism for Polarizationless P systems. *Pre-proceedings of the 12th International Meeting on DNA Computing*, (C. Mao, T. Yokomori, B.-T. Zang, editors), Seoul, June 2006, 203–214, and *Lecture Notes in Computer Science LNCS 4287*, Springer, Berlin, (2006) 17–32.

[52] Ishdorj, T.-O., Ionescu, M., Replicative-Distribution Rules in P Systems with Active Membranes. *Pre-proceedings of First International Colloquium on Theoretical Aspects of Computing ICTAC*, Guiyang, China, September 20-24, 2004 - UNU/IIST Report No. 310, 263–278, Zhiming Liu (Ed.) Macau, and *Lecture Notes in Computer Science LNCS 3407*, Springer, Berlin, (2005) 69–84.

[53] Ishdorj, T.-O., Petre, I., An Efficient Computing Paradigm Inspired by Gene Assembly in Ciliates. Manuscript, 2006.

[54] Ishdorj, T.-O., Petre, I., Vladimir, R., Computational Power of Intramolecular Gene Assembly. Manuscript, 2006.

[55] Kari, L., and Landweber, L. F., Computational Power of Gene Rearrangement. In: Winfree, E., Gifford, D. K., (eds.) *Proceedings of DNA Based Computers, V* American Mathematical Society (1999) 207–216.

[56] Kari, L., and Thierrin, G., Contextual Insertion/Deletions and Computability. *Information and Computation* 131 (1996) 47–61.

[57] Kleene, S. C., Representation of Events in Nerve Nets and Finite Automata. In *Automata Studies*, Princeton University Press, Princeton, NJ, 1956, 3–42.

[58] Landweber, L. F., and Kari, L., The Evolution of Cellular Computing: Nature's Solution to a Computational Problem. In: *Proceedings of the 4th DIMACS Meeting on DNA-Based Computers*, Philadelphia, PA (1998) 3–15.

[59] Landweber, L. F., and Kari, L., Universal Molecular Computation in Ciliates. In: Landweber, L. F., Winfree, E (eds.) *Evolution as Computation*, Springer, Berlin, 2002.

[60] Lindenmayer, A., Mathematical Models for Cellular Interaction in Development I and II. *Journal of Theoretical Biology* 18(1968), 280–315.

[61] Lipton, R. J., Using DNA to Solve NP-Complete Problems. *Science*, 268 (1995), 542–545.

[62] Maass, W., Computing with Spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8, 1 (2002), 32–36.

[63] Marcus, S., Contextual Grammars. *Revue Roumaine de Matématique Pures et Appliquées*, 14 (1969), 1525–1534.

[64] Martín-Vide, C., Ishdorj, T.-O., Modeling Neural Processes in Lindenmayer Systems. *The 8th International Work-Conference on Artificial Neural Networks* (IWANN'2005) (Computational Intelligence and Bioinspired Systems), Vilanova i la Geltrú (Barcelona, Spain) June 8-10, 2005 and LNCS 3512, Springer, Berlin, (2005), 145–152.

[65] Mauri, G., Păun, Gh., Pérez-Jiménez, M. J., Rozenberg, G., Salomaa, A., eds., *Membrane Computing, 5th International Workshop, WMC2004*. Milano, Italy, June 2004, LNCS 3365, Springer, Berlin, 2005.

[66] McCulloch, W. S., Pitts, W. H., A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5(1943), 115–133.

[67] Minsky, M. L., *Computations. Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, 1967.

[68] Von Neumann, J., *Theory of Self-Reproducing Automata.* The Univeristy of Illinois Press, Champaign, Illinois, 1966.

[69] Von Neumann, J., *The Computer and the Brain.* Yale Univeristy Press, New haven and London, 1958.

[70] Ogihara, M., Ray, A., Simulating Boolean Circuits on a DNA Computer. *Proceedings of the First Annual International Conference on Computational Molecular Biology*, Santa Fe, New Mexico, USA, 1997, 226–231.

[71] Pan, L., Alhazov, A., Solving HPP and SAT by P Systems with Active Membranes and Separation Rules. *IEEE Transactions on Computers*, submitted, 2005.

[72] Pan, L., Alhazov, A., Ishdorj, T.-O., Further Remarks on P Systems with Active Membranes, Separation, Merging and Release Rules. *Soft Computing. A Fusion of Foundations, Methodologies and Applications*, (2005) 686–690.

[73] Pan, L., Ishdorj, T.-O., P Systems with Active Membranes and Separation Rules. *Journal of Universal Computer Science*, 10(5) (2004), 630–649.

[74] Papadimitriou, Ch. P., *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[75] Păun, Gh., *Twenty Six research Topics About Spiking Neural P Systems.* 2006, in [106].

[76] Păun, Gh., *Further Open Problems in Membrane Computing.* 2004, in [106].

[77] Păun, Gh., On the Power of the Splicing Operation. *Int. J. Comp. Math* 59 (1995) 27–35.

[78] Păun, Gh., *Marcus Contextual Grammars.* Kluwer, Dordrecht, 1997.

[79] Păun, Gh., Computing with Membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and *Turku Center for Computer Science-TUCS Report* No 208, 1998 (www.tucs.fi).

[80] Păun, Gh., P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal Automata Languages and Combinatorics*, 6–1 (2001), 75–90, and CDMTCS Research Report 102, Auckland University, 1999.

[81] Păun, Gh., *Membrane Computing. An Introduction.* Springer, Berlin, 2002.

[82] Păun, Gh., Introduction to Membrane Computing. In *Proceedings First Brainstorming Workshop on Uncertainty in Membrane Computing*, Palma de Mallorca, Spain, November 2004, 1–42.

[83] Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G., Spike Trains in Spiking Neural P Systems. *Intern. J. Found. Computer Sci.*, 17, 4(2006), 975–1002.

[84] Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G., Infinite Spike Trains in Spiking Neural P Systems. Submitted, 2006.

[85] Păun, Gh., Riscos-Núñez, A., Romero-Jiménez, A., Sancho-Caparrini, F., eds., *Proceedings Second Brainstorming Week on Membrane Computing.* Seville, 2004, GCN Technical Report 01/2004, University of Seville, 2004.

[86] Păun, Gh., Rozenberg, G., A Guide to Membrane Computing. *Theoretical Computer Science*, 287-1 (2002), 73–100.

[87] Păun, Gh., Rozenberg, G., Salomaa, A., *DNA Computing - New Computing Paradigms.* Springer, Berlin, 1998.

[88] Pérez-Jiménez, M. J., Romero-Jiménez, A., Sancho-Caparrini, F., Complexity Classes in Models of Cellular Computation with Membranes. *Natural Computing*, 2, 3 (2003), 265–285.

[89] Prescott, D. M., Cutting, Splicing, Reordering, and Elimination of DNA Sequences in Hypotrichous Ciliates. *BioEssays* 14 (1992) 317–324.

[90] Prescott, D. M., and DuBois, M., Internal Eliminated Segments (IESs) of Oxytrichidae. *J. Eukariot. Microbiol.* 43 (1996) 432–441.

[91] Prescott, D. M., Ehrenfeucht, A., Rozenberg, G., Molecular Operations for DNA Processing in Hypotrichous Ciliates. *Europ. J. Protistology* 37 (2001) 241–260.

[92] Prescott, D. M., Rozenberg, G., Encrypted Genes and Their Reassembly in Ciliates. In: M. Amos (ed.) *Cellular Computing*, Oxford University Press, Oxford (2003).

[93] Riscos-Núñez, A., *Cellular Programming: Efficient Resolution of NP-complete Numerical problems.* PhD thesis, University of Sevilla, 2004.

[94] Roweis, S., Winfree, E., Burgoyne, R., Chelyapov, N., Goodman, M., Rothemund, P. W. K., Adleman, L. M., A Sticker Based Architecture for DNA Computation. In [6], 1–27.

[95] Rozenberg, G., Lindenmayer, A., Developmental Systems with Locally Catenative Formulas. *Acta Inf.* 2: (1973) 214–48.

[96] Rozenberg, G., Salomaa, A., *The Mathematical Theory of L Systems.* Academic Press, New York, 1980.

[97] Rozenberg, G., Salomaa, A., eds., *Handbook of Formal Languages.* Springer, Berlin, 1997.

[98] Salomaa, A., *Formal Languages.* Academic Press, New York, 1973.

[99] Sburlan, D., *Promoting and Inhibiting Contexts in Membrane Computing.* PhD thesis, University of Sevilla, 2006.

[100] Segev, I., Schneidman, E., Axons as Computing Devices: Basic Insights Gained from Models, *J. Physiol. (Paris)*, 93 (1999), 263–270.

[101] Shepherd, G. M., *Neurobiology.* Oxford University Press, NY Oxford, 1994.

[102] Sipser, M., *Introduction to the Theory of Computation.* PWS Publishing Company, International Thomson Publishing Company, 1997.

[103] Turing, A. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Series 2, Vol. 42 (1936–37) 230–265.

[104] Winfree, E., Liu, F., Wenzler, L. A., Seeman, N. C., Design and Self-Assembly of Two-Dimensional DNA Crystals. *Nature*, 394(6693), (1998), 539–544.

[105] Winfree, E., Yang, X., Seeman, N. C., Universal Computation Via Self-Assembly of DNA: Some Theory and Experiments. In [6], 191–214.

[106] P systems web-page: `http://psystems.disco.unimib.it/`

## UNIVERSIDAD DE SEVILLA

Reunido el tribunal en el día de la fecha, integrado por los abajo firmantes, para evaluar la tesis doctoral de D. Tseren-Onolt Ishdorj
titulada Membrane Computing, Neural Inspirations, Gene Assembly in Ciliates
acordó otorgarle la calificación de

Sevilla, a 28 de Marzo de 2007

Vocal,
Rudolf Freund
Fdo.: Rudolf Freund

Vocal,
Fdo.: Marian Gheorghe

Vocal,
Fdo.: Claudio Zandron

Presidente,
Fdo.: Francesc Rosselló Llompart

Secretario,
Fdo.: José M.ª Sempere Luna

Doctorando,
Fdo.: Tseren-Onolt Ishdorj.