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

Evolution, Communication, Observation: From Biology to Membrane Computing and Back

Memoria presentada por
Matteo Cavaliere
para optar al grado de Doctor
por la Universidad de Sevilla

Matteo Cavaliere

V.º B.º Los Directores de la Tesis

Dr. D. Gheorghe Păun

Dr. D. Mario de J. Pérez Jiménez

Sevilla, 15 de noviembre 2005

Resumen de la Tesis

EVOLUCIÓN, COMUNICACIÓN Y OBSERVACIÓN: DE LA BIOLOGÍA A LA COMPUTACIÓN CON MEMBRANAS Y VICEVERSA

Matteo Cavaliere

La *Computación Natural* es una disciplina cuyo objetivo fundamental es la simulación e implementación de procesos dinámicos que se dan en la Naturaleza y que son susceptibles de ser interpretados como procedimientos de cálculo.

La simulación que aborda la Computación Natural puede tener diversas interpretaciones a la hora de describir los nuevos modelos: que se utilice para el diseño de nuevos esquemas algorítmicos usando técnicas inspiradas en la naturaleza, o bien que sugiera la creación física de nuevos modelos experimentales en los que el medio electrónico de los ordenadores convencionales se sustituya por otro sustrato que pueda implementar ciertos procesos que aparecen en el modo de operar de la Naturaleza.

Como ejemplo de la *primera interpretación*, podemos considerar los *Algoritmos Genéticos* (o de manera más general, la *computación evolutiva*), que se basan en el proceso genético de los seres vivos a través del cual evolucionan y cuyo elemento fundamental es el principio de selección natural, y las *redes neuronales artificiales* que están inspiradas en la organización y funcionamiento del cerebro (siendo el origen de los autómatas finitos).

Como ejemplo de la *segunda interpretación*, a finales de la década de los cincuenta el premio nobel R.P. Feynman postula la necesidad de considerar operaciones *sub–microscópicas* como única alternativa revolucionaria en la carrera por la miniaturización de las componentes físicas de los ordenadores convencionales, y propone la computación a *nivel molecular* como posible modelo en el que implementar dichas operaciones. En 1987, T. Head propone explícitamente el primer modelo teórico molecular basándose en las

propiedades de las moléculas de ADN. En noviembre de 1994, L. Adleman realiza el primer experimento en un laboratorio que permite resolver una instancia concreta de un problema **NP**-completo a través de la manipulación de moléculas de ADN.

Un estudio más detallado del funcionamiento de los organismos vivos nos sugiere un nuevo nivel de computación: *el nivel celular.*

El comportamiento de una célula puede ser considerado como el de una máquina no trivial, desde el punto de vista biológico, en la que por medio de una distribución jerárquica de membranas interiores se produce el flujo y alteración de las componentes químicas que la propia célula procesa.

Esta memoria versa sobre el estudio de diversos aspectos teóricos de los sistemas celulares con membranas (que suelen denominarse sistemas P) y el análisis de sus posibles aplicaciones a la simulación y modelización de procesos biológicos.

El objetivo fundamental de la tesis es mostrar que los sistemas celulares pueden ser usados como un nuevo marco no convencional de modelización matemática de procesos biológicos, desde una óptica computacional, y que es posible manipular, modificar y adaptar dicho modelo matemático a través de ideas y sugerencias que parten de la Biología.

La interacción entre Matemática y Biología proporciona una serie de posibilidades muy atractivas desde el punto de vista de la investigación científica: la elaboración de nuevos dispositivos matemático–computacionales que pueden ser usados para modelizar procesos biológicos y, a la vez, la formulación de problemas matemáticos relacionados con nuevos paradigmas de computación.

Seguidamente, se describe la estructura y el contenido específico de la memoria que se presenta.

Capítulo Primero: El primer capítulo tiene como objetivo la introducción de todos aquellos conceptos básicos y resultados que van a ser utilizados a lo largo de la memoria ya que se pretende que, en la medida de lo posible, la tesis sea autocontenida.

En ese contexto, se presentan las nociones básicas de la teoría de lenguajes formales y la teoría de autómatas, detallándose algunos resultados que son importantes en la tesis, debido a que los resultados relativos a universalidad de modelos y a decidibilidad de cuestiones son abordadas a través del uso de técnicas y herramientas de las teorías antes citadas. También se ha considerado pertinente el resumen de una serie de nociones básicas relativas a las células y a las membranas biológicas de los organismos vivos.

La tesis se desarrolla en el marco de la *Computación Celular con Membranas.* Por ello, se introduce en este primer capítulo los primeros conceptos de los sistemas celulares, concretamente en lo relativo a dispositivos computacionales que trabajan a modo de células (*cell–like*), con símbolos

atómicos de un alfabeto finito (símbolos–objetos, como abstracción de las sustancias químicas que aparecen dentro de las células) y con dos modelos básicos que usan diferentes tipos de reglas: las de evolución (como abstracción de las reacciones químicas que se producen en las celúlas y que permiten la transformación de unas sustancias químicas en otras) y las de comunicación (reglas symport-antiport que formalizan el flujo de sustancias entre distintas membranas). En este capítulo se presentan, asímismo, algunos resultados relativos a la potencia computacional de los modelos básicos que serán usados en la memoria.

Capítulo Segundo: En los sistemas celulares se modelizan los procesos biológicos de evolución y de comunicación. Ahora bien, estos dos tipos de procesos en realidad no se producen de manera estríctamente separadas, sino que suceden de manera simultánea e independiente; es decir, mientras que unas sustancias químicas están siendo transformadas a través de reacciones químicas, en el mismo instante es posible que otras sustancias estén siendo transportadas de una membrana a otra.

Todo ello sugiere la posibilidad de diseñar nuevos dispositivos híbridos que combinen la evolución (materializada mediante reglas de evolución) y la comunicación (especificada a través de reglas symport/antiport). De esta manera se definen los modelos celulares de *evolución–comunicación*, introducidos en [46]. Este nuevo tipo de sistemas P ha sido diseñado combinando en un único modelo los mecanismos evolutivos presentes en los sistemas de membranas con símbolos-objetos y reglas de reescritura sobre multiconjuntos, así como los mecanismos de comunicación presentes en los sistemas de membranas con reglas symport/antiport. Se presenta la definición formal del modelo y se estudian distintos resultados relativos a la potencia computacional del mismo, presentándose un resultado de universalidad obtenido usando reglas de evolución no cooperativas y reglas symport/antiport de peso a lo sumo uno.

Como se ha comentado, en el modelo de evolución-comunicación los objectos evolucionan por medio de reglas de evolución y son transportados por medio de reglas symport/antiport. Ahora bien, es interesante resaltar el hecho de que las reglas de evolución se asignan de una manera fija a las regiones mientras que las reglas symport/antiport se asignan de manera fija a las membranas del sistema.

En este contexto, cabría preguntarse qué sucedería con la la situación dual; es decir, considerando que los objetos no pueden ser transportados a través de las membranas mientras que, en cambio, sí pueden serlo las reglas de evolución, a través de las membranas usando reglas symport/antiport. Este modelo, denominado *sistemas P con symport/antiport sobre reglas*, fue introducido en [50]. La idea del mismo puede ser expresada en los siguientes términos: la computación se realiza a través del movimiento de instrucciones (reglas) de una membrana a otra, en lugar del desplazamiento de datos

(objetos) usando mecanismos de comunicación bio-inspirados.

Se presentan varios resultados que demuestran cómo la capacidad generativa de estos sistemas celulares (así como el número de membranas necesario para conseguir la universalidad) dependen de la potencia de las reglas de transporte (del peso de las mismas) y de las reglas de evolución (expresadas en términos de cooperación).

Capítulo Tercero: Este capítulo está dedicado a justificar que la Computación Celular con Membranas puede ser utilizada para la modelización de algunos procesos biológicos. Concretamente, se presenta un simulador de sistemas P que implementa el modelo de evolución-comunicación enriquecido con algunos parámetros probabilísticos, inspirados en la biología celular.

Se ilustra la utilidad del simulador para la obtención de hipótesis de trabajo con implicaciones biológicas, a través del estudio y análisis de dos procesos biológicos simples pero, a la vez, importantes: la respiración en la bacteria *Escherichia coli* y la interacción entre la respiración y la fotosíntesis (es decir, entre la producción y el consumo de oxígeno) en la bacteria *Synechocystis*.

Se realiza un estudio comparativo del simulador desarrollado con datos experimentales y se analizan las distintas elecciones que se hacen durante el desarrollo del software: de esta forma se introducen nuevos conceptos en el área de la computación con membranas y, al mismo tiempo, se pueden comparar los conceptos estándares del área con aquellos presentes en biología celular. Este capítulo está basado en los resultados presentados en [18] y en el capítulo [47] del reciente libro [63] dedicado a las aplicaciones de los sistemas de membranas.

Capítulo Cuarto: En este capítulo se pone de manifiesto, nuevamente, la importancia de la interacción entre Biología y Matemáticas, en el sentido de que nuevos e interesantes modelos matemático–computacionales pueden ser diseñados a través del análisis detallado de ciertos procesos biológicos.

Se presenta, inicialmente, un nuevo un modelo de sistema celular, denominado *protón pumping P system*, que viene a ser una variante del modelo de evolución-comunicación, y en el cual se exige una condición adicional a las reglas antiport, inspirada en el sistema de inyección de protones en las células vivas (los protones serán unos símbolos distinguidos del alfabeto que no pueden ser creados ni detruidos, y sólo se admiten reglas antiport que intercambian símbolos por un único protón). Se analiza la potencia computacional del nuevo modelo y se establece su universalidad en ciertos casos.

Ahora bien, por una parte el tranpsorte de protones, así como otras reacciones químicas, necesitan cierto tiempo para ser ejecutadas. Por otra, distintos procesos biológicos necesitan, en general, diferentes tiempos de ejecución. Por tanto, los sistemas celulares *ordinarios*, en los que cada regla

es ejecutada en una unidad de tiempo, no parece tener una contrapartida biológica real. Por ello parece interesante estudiar modelos que capturen esta idea y en los que cada regla tenga asignado un tiempo de ejecución.

No obstante, el tiempo que una reacción/proceso necesita para su ejecución, puede no ser constante y depender de una serie de factores ambientales, difícilmente predecibles y controlables. Por lo tanto, parece también interesante estudiar los sistemas que no dependen de los distintos tiempos de ejecución asociados a las reacciones.

Estas ideas se pueden plasmar dentro del marco de la Computación Celular con Membranas introduciendo una clase específica de sistemas P en los que el resultado de las computaciones no dependan del tiempo de ejecución de las reglas involucradas. Para estos sistemas (que se denominan independientes del tiempo, *time-free P systems*) introducidos en [58], la sincronización de las reglas debe ser materializada usando ingredientes diferentes a un reloj global de todo el sistema. Al igual que en el mundo real es fundamental la sincronización de sucesos independientes, en el caso de los sistemas celulares independientes del tiempo hay que establecer métodos de sincronización de reglas con diferente tiempo de ejecución y que se ejecuten en paralelo, a fin de obtener una potencia computacional aceptable.

Una posible solución sería el uso de señales promotoras: como sucede en las células vivas, la sincronización de las diferentes reglas puede ser realizada usando señales que se mueven de unas regiones del sistema a otras de tal manera que pueden activar y/o promover ciertas reglas específicas.

Otra posible forma de sincronizar las computaciones en un sistema independiente del tiempo es a través de catalizadores bi-estables, o bien a través de mecanismos de transporte basados en reglas symport/antiport.

Se demuestra que la universalidad de los sistemas celulares independientes del tiempo puede ser obtenida haciendo uso de un sistema de sincronización basado, únicamente, en procesos de comunicación (con reglas symport/antiport). Y se analiza la decidibilidad del problema que consiste en dado un sistema P arbitrario, determinar si es o no independiente del tiempo.

Capítulo Quinto: En capítulos anteriores de la memoria se ha justificado la posibilidad de diseñar nuevos dispositivos computacionales *mirando* dentro de la célula y *observando* los procesos que tienen lugar en su interior. Sin embargo, parece plausible obtener nuevos paradigmas computacionales analizando el papel fundamental que el observador juega en muchos experimentos biológicos, pensando genéricamente que el observador actúa como una especie de máquina que es capaz de mirar una configuración de un sistema que está evolucionando y asociarle un cierto significado (un determinado *valor*).

El paradigma llamado evolución/observación se basa en la idea de que

es posible realizar computaciones simplemente observando la evolución de un sistema. El estudio y análisis de este paradigma, desde un punto de vista computacional, es el objetivo de este capítulo.

Con esta filosofía, se pueden diseñar dispositivos que estén compuestos de dos sistemas más simples: una especie de sistema biológico que, simplemente, *vive*, pasando de un estado a otro y produciendo un cierto comportamiento; y un segundo sistema colocado en el *exterior* del anterior que, simplemente, mira y observa el sistema biológico.

Un primer ejemplo de tal paradigma es representado por medio de sistemas P/O introducidos en [53]. Básicamente, un dispositivo P/O se compone de un sistema celular con membranas y un autómata sobre multiconjuntos, de tal manera que este último *observa* desde el exterior la evolución del sistema celular y traduce dicha observación en una salida legible. De esta forma un sistema P/O puede considerarse como un dispositivo computacional, y se demuestra que dicho modelo es universal incluso cuando el sistema celular y el observador no son dispositivos *demasiado potentes*. Se presenta un ejemplo que da una idea muy concreta del funcionamiento de este paradigma.

El paradigma de evolución/observación es muy general y puede extenderse a otras áreas de la teoría de la computación. Así por ejemplo, en la memoria se presenta una generalización a gramáticas formales, analizando la evolución de una gramática con un observador apropiado y considerando la evolución observada como la salida del proceso. Esta idea se usa en la definición de un nuevo dispositivo generador llamado *sistema gramática/observador*. Se presentan tres tipologías del funcionamiento de estos sistemas.

El marco computacional de evolución/observación puede, también, ser usado para la construcción de dispositivos de *decisión* con tal de considerar un sistema biológico, establecer un mecanismo para la introducción de una entrada debidamente codificada y observar su evolución. De tal manera que si dicha evolución es de un *tipo esperado* (por ejemplo, que siga un patrón predeterminado), entonces la entrada es aceptada y, en caso contrario, es rechazada. Es decir, un observador externo es usado para extraer el comportamiento formal de la evolución de un sistema biológico.

El paradigma evolución/observación puede sugerir, asímismo, una nueva visión de la computación molecular basada en ADN. Para ello, se consideran dispositivos reconocedores basados en splicing, que son modelos formales de recombinación de sucesiones de doble hebras de ADN bajo la acción de la enzima ligasa y de otras enzimas de restricción. La idea de estos dispositivos es que se puede computar observando la evolución de una cadena marcada en un sistema de splicing. Se pueden obtener dispositivos reconocedores que son universales incluso en el caso en que sus componentes no sean demasiado potentes (por ejemplo, regulares). Por otro lado, estos dispositivos están

bien motivados ya que recientemente se han desarrollado diferentes técnicas para observar la dinámica de una molécula de ADN.

El último capítulo de la memoria está dedicado a la presentación de algunas conclusiones que se pueden extraer de los resultados obtenidos en la tesis. Se describen varios problemas que aún permanecen sin resolver y que están directamente relacionados con los tópicos desarrollados, y se proponen una serie de ideas, sugerencias y objetivos para trabajos futuros que marcan nuevas líneas de investigación en modelos de computación no convencionales, algunas de las cuales han comenzado a desarrollarse en colaboración con algunos miembros del grupo de investigación en Computación Natural de la Universidad de Sevilla.

Conviene resaltar que al final de cada capítulo se han incluido unas observaciones relacionadas con los temas abordados y que incluyen ilustrativas consideraciones bibliográficas, así como los últimos resultados que se han obtenido y que conciernen al contenido del capítulo.

Acknowledgments

I wish to thank all the people I had the luck and the pleasure to work with and, in particular, all the collaborators of the research I have done during my PhD career.

Above all, I would like to thank Gheorghe Păun for his constant interest, encouragement and help. His contagious scientific creativity has been fundamental in these years.

My warmest thank to Mario Pérez Jiménez for his continuous support and for his precious and always present scientific help.

Thanks to all the members of the *Research Group on Natural Computing* of the University of Seville for creating such a wonderful place for working.

Thanks to my parents, Aurelia and Giuseppe and to my sister Anna for their constant support and encouragement.

Thanks to Marilena for her precious help along these years. Thanks to Marja for her special friendship.

Contents

An Introduction to the Thesis

Natural Computing: An Historical Perspective

Natural computing is a new and 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) try to construct 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.

Molecular and Cellular Computing

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 [73].

Few years later, in 1973, C. Bennett discussed how to perform computations on a molecular scale (see [27]) proposing to use RNA molecules as

physical medium for implementing computations; in a following work [28] Bennett proposed a theoretical model of a Turing machine storing information using RNA and processing them by using (imaginary) enzymes to catalyze biochemical reactions.

Almost a decade later, in 1987, T. Head presented (see [87]) *splicing systems*, a theoretical computational model based on recombination of DNA molecules under the influence of restriction enzymes and ligases.

All these ideas were implemented in 1994 with Adleman's seminal experiment, [1], where an instance of the *Hamiltonian path problem* is solved in a laboratory experiment using DNA molecules and biomolecular operations. This experiment constituted the proof of the possibility to compute by using biomolecules. After this experiment (considered the birth of molecular computing) numerous theoretical and experimental results have followed.

Several other theoretical and experimental investigations have shown that it is possible to solve hard computational problems by using molecular computing, mainly by using DNA molecules (DNA computing).

For instance, molecular algorithms for solving the *satisfiability problem* (SAT) have been published in literature (see, e.g., [91, 99, 100]). On the other hand, also experimental solutions of small instances of this problem have been presented (see [38, 140]).

In [2] it is presented probably the most impressive (potential) use of DNA computing: a molecular algorithm for attacking the *Data Encryption Standard, DES* (the authors suggest that, approximatively, one gram of DNA would be needed).

Together with such specific applications of molecular computing a more general research line tries to construct general computing devices by using biomolecules. Several authors have proved that, theoretically, using biomolecules and biomolecular operations is possible to construct very powerful computing devices (even Turing machines).

For instance, the reader can see the universality results presented in [114, 76] or the description of the implementation of a biomolecular Turing machine presented in [135].

On the other hand, also several tentatives to implement in laboratory general biomolecular computing devices have been reported.

In this respect we can distinguish two main areas of research: implementing computing devices by using DNA self-assembly and implementing computing devices by using DNA recombination and restriction enzymes.

The use of self-assembly has been widely investigated by E. Winfree that has shown how standard models of computations like formal grammars and cellular automata can be implemented by using self-assembly (see [136, 144, 145, 147, 148]). Recently self-assembly has been successfully used to construct binary counters, [20]. Several papers deal with the construction of self-assembly models that are error-free or that are able to (self) correct possible errors (see [80, 61, 60]).

Interesting and very promising results have also been obtained by using DNA recombination and restriction enzymes. In 2001 Shapiro et al. ([26]) have shown an implementation *in vitro* of finite state automata using 2 states and 2 input symbols. In fact, in [26] several different automata were constructed by changing the biomolecules initially present in the test tube; the important point of this experiment is that the computation was carried out by an autonomous processing of the input DNA molecules; in [24] it is shown that the experiment can be further simplified by removing the use of ligase.

In 2005 Keinan et al. ([142]) have improved the previous results by implementing in vitro finite state automata with 3 states and 3 input symbols. A possible extension of this experiment to implement more powerful computing devices, with unbounded memory, has been presented in [52].

In 2004 an exciting result has been presented by Shapiro et al. (see [25]). There, the authors describe an experiment where an autonomous biomolecular computer makes, in vitro, a logical analysis of the levels of messenger RNA species and in response produces a molecule capable of affecting levels of gene expression. If implemented, this device can be considered a kind of *smart drug* that operates only where the conditions of the disease hold.

A new area of molecular computing concerns computations in living cells. An abstract computational model inspired by the structure and the functioning of living cells is a *membrane system.* Membrane systems have been introduced in 1998 by Gh. Păun, see [115]. Since their introduction a large number of membrane systems models have been introduced in the literature. Several of them have been proved to be computationally complete and useful for solving in an efficient way hard computational problems (see, for instance [116]). In 2003 *Thomson Institute for Scientific Information, ISI*, has nominated membrane computing as *fast emerging research front in computer science*, see [150], with the initial paper considered *fast breaking paper.*

Several attempts to use living cells as computing devices have been reported in literature. See, for instance, the papers of R. Weiss et al., [21, 143]. Recently, a novel framework for developing a programmable in vivo finite automaton using living cells has been presented in [111].

On the other hand, another general possibility consists in investigating computational processes that take place in living cells. For instance, a fruitful line of research consists in analyzing the process of gene assembly in *ciliates* that are unicellular organisms (the reader can consult the recent monograph [69] dedicated to this research area).

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 eleventh year ([23, 22, 94, 146, 93, 85, 62, 72, 43, 139, 64])

or the monographs dedicated to the area, for instance [41, 122], as well as the Proceedings of yearly workshops on Membrane Computing ([103], [123], [102], [105], [77]) with the full information about this area available in the web-page [151].

Content of the Thesis

This Thesis deals with theoretical aspects of membrane systems (currently referred to as P systems) and with their possible applications for modeling biological processes.

The goal of the Thesis is to show how membrane systems can be used as mathematical model of cellular processes and how it is possible to change and to adapt the mathematical model by using ideas and suggestions coming from biology.

The interaction between mathematics and biology brings two important results: the construction of new mathematical devices that can be used to model cellular processes (in an always more precise way) and the creation of new challenging mathematical (often, formal languages) problems related with new bio-inspired paradigms of computation.

The Thesis starts (**first chapter**) with an introduction to the notions of formal languages and automata theory used. We also recall some basic notions concerning cellular biology.

In the same chapter we also give a brief introduction to membrane systems that are the main subject of the Thesis; we present a short overview of the field, the definitions and the main results of the two basic models (with symbol-objects and symport/antiport rules) used in the Thesis.

The **second chapter** starts with the observation that in many biological phenomena and, in particular, in many cellular phenomena, evolution and communication are processes that happen in a simultaneous and parallel way. This simple consideration constitutes the suggestion for a new model of P systems: the *evolution-communication model* originally introduced in [46]. This new kind of P systems has been created by joining, in an unique model, the evolutive mechanism present in membrane systems with symbol-objects and multiset-rewriting rules (modeling biochemical reactions) and the communicative mechanism present in membrane systems with symport/antiport rules (modeling the transport mechanisms present in living cells). We present the definition of the model together with an universality result obtained by using non-cooperative evolution rules and symport/antiport rules of weight at most one.

In the evolution-communication model objects evolve by using evolution rules and are moved by using symport and antiport rules; the evolution rules

are assigned in a fixed way to the regions while the symport/antiport rules are assigned in a fixed way to the membranes of the system. A natural question concerns the case when a dual situation is considered: the objects cannot be moved across the regions, while the evolution rules are moved across the membranes by using symport/antiport rules. This model called *P systems with symport/antiport of rules* has been introduced in [50] and is recalled here in the second chapter. The idea of this model can be expressed in the following general terms: computing by moving instructions (rules) instead of data (objects) by using biologically motivated communication mechanisms. We present several results that show how the power of the type of transport used (symport or/and antiport) and of the evolution rules used (cooperative, catalytic) influences the generative power of the system.

The evolution-communication model, enriched with some probabilistic parameters, has been implemented by a software simulator that we present in the **third chapter**.

Moreover, we show how it is possible to use the mathematical model and therefore the associated software to simulate simple but important biological processes such as respiration in *Escherichia coli* and the interaction existing in *Synechocystis* between respiration and photosynthesis. We show that the simulator can be used to infer useful results for biologists.

We also compare the simulator realized with the biological reality, discussing the choices made in the implementation: in this way we establish a link (the interaction with biology mentioned earlier) between the mathematical model and the biological reality, introducing new concepts in the membrane computing area, and at the same time, comparing the standard concepts of the membrane computing area with what we have in cell biology. The third chapter is based on the results presented in [18] and in the chapter [47] of the recent monograph [63] dedicated to applications of membrane systems.

In the fourth and fifth chapters we come back to mathematics and, in a certain sense, we give the "proof" that the interaction with biology can be really fruitful even from only a purely mathematical point of view.

In the **fourth chapter** we initially present a model of membrane systems, *proton pumping P systems*, that are, essentially based on the evolution-communication model, with a restriction (on the antiport rules) inspired by the pumping of protons present in living cells. Even with this restriction, proton pumping P systems are shown to be universal when using non-cooperative evolution rules and symport/antiport rules of weight at most one.

The movement of protons, like any other biochemical reaction/process that takes place in living cells, needs a certain time to be completed. On

the other hand, different biochemical reactions/processes may take different times to be completed.

Actually the time that a reaction/process needs to be completed might not be a constant but may depend on many environmental factors. These factors can be difficult to predict and to control; therefore, it would be useful to have systems that behave in a certain expected way independently of the times of execution of the involved reactions/processes.

We can translate this idea in the membrane computing framework searching for a specific class of P systems where the result of the computation does not depend on the time of execution of the involved rules. For these systems, called *time-free P systems*, introduced in [58], the synchronization of the involved rules must be realized by using ingredients different from the unique global clock. A possible solution is the use of signal-promoters: like in living cells, the synchronization of different rules can be done by using signals traveling around the regions of the system that can activate/promote certain specified rules.

Another possible way of synchronizing the computation in time-free P systems is by means of bi-stable catalysts or transport mechanisms based on symport/antiport rules. A survey of different models of time-free P systems together with their computational power (some of them are shown to be universal) is given in the second part of the fourth chapter. Also the problem of deciding whether an arbitrary P system is time-free is investigated.

We can get inspirations for new bio-computing devices by looking to the biochemical processes taking place in living cells but also by looking to the procedures that biologists use during their experiments. In particular, in any biological experiment the observation process plays a crucial role.

In the **fifth chapter**, we present a paradigm of computation which stresses the role of observation. This paradigm called *evolution/observation* is based on the idea that it is possible to compute by observing the evolution of a system.

A first example of such paradigm is represented by *P/O systems* introduced in [53]. A P/O system is composed by a membrane system and a multiset finite automaton; this last one "observes", from outside, the evolution of the membrane system and then translates this evolution into a readable output. In this way a P/O system can be seen like a computing device that is universal even when the membrane system and the observer used are not so powerful and only halting computations of the membrane system are considered (we present the proof of such universality). Moreover, we show an example that gives an idea of the functioning of this computing device. We also present the case when only non-halting computations of the membrane systems are considered; in this case a comparison with classes of languages of infinite words is presented.

However, the evolution/observation paradigm is very general and can be naturally extended to other fields of the theory of computation. For instance, it can be generalized to formal grammars: watch the "evolution" of a grammar with an appropriate observer and to take this observed evolution as the output of the process. This idea is used to define a new generative device called *grammar/observer system*, in short *G/O system*, recalled also in the fifth chapter. There, three modes (*initial, free, always-writing*) of working for G/O systems are presented; G/O systems working in the initial and free modes are shown to be universal even when using simple components; on the other hand, the class of languages generated by always-writing G/O systems includes the class of context-free languages but is included in the class of context-sensitive languages.

Actually, the evolution/observation paradigm can also suggest a new way to look at DNA computing: to give an example of that, in the final part of the fifth chapter, we present *splicing recognizers*, in short *SR*, that are accepting devices whose main idea is to compute by observing the evolution of a marked string inserted in a splicing system. Universal recognizer devices can be obtained even when the components of the SRs are not so powerful (for instance, regular). On the other hand, SRs are well-motivated since several techniques for observing the dynamics of a single DNA molecule and in general of a single biomolecule have been recently developed.

During the Thesis, at the end of each chapter, the reader can find final remarks including bibliographical notes and the last results obtained in literature.

Moreover, in the final chapter, the one devoted to conclusions, we recall some of the interesting problems that are still open and we also indicate suggestions to develop further research.

The content of the Thesis is mostly based on original research done by the author (some of them in collaboration) and published in international journals and conferences. We give below some information in this respect:

Chapter 2:

1. M. Cavaliere, Evolution-Communication P Systems. In [123], pp. 134–145.

2. M. Cavaliere, D. Genova, P Systems with Symport/Antiport of Rules. *Journal of Universal Computer Science*, 10-5 (2004), pp. 540–558.

Chapter 3:

3. I.I. Ardelean, M. Cavaliere, Modeling Biological Processes by Using a Probabilistic P System Software. *Natural Computing*, 2-3 (2003),

pp. 173–197.

4. M. Cavaliere, I.I. Ardelean, Modeling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria by Using a P System Simulator. In [63], pp. 129 –158.

Chapter 4:

5. A. Alhazov, M. Cavaliere, Proton Pumping P Systems. In [102], pp. 70–88.

6. M. Cavaliere, V. Deufemia, Further Results on Time-Free P Systems. *International Journal of Foundations of Computer Science*, to appear.

7. M. Cavaliere, D. Sburlan, Time-Independent P Systems. In [105], pp. 239–258.

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

Chapter 5:

9. M. Cavaliere, N. Jonoska, P. Leupold, Recognizing DNA Splicing. In [43], pp. 6–16.

10. M. Cavaliere, P. Leupold, Evolution and Observation – A New Way to Look at Membrane Systems. In [102], pp. 70–87.

11. M. Cavaliere, P. Leupold, Evolution and Observation–A Non-Standard Way to Generate Formal Languages. *Theoretical Computer Science*, 321, 2-3 (2004), pp. 233–248.

12. M. Cavaliere, P. Leupold, Evolution and Observation – A Non-Standard Way to Accept Formal Languages. In [101], pp. 152–162.

Chapter 1

Preliminaries

In this chapter we recall the notions of formal language and automata theory used in the Thesis. We also shortly recall some basic notions of cellular biology. Finally we give a general (and short) overview about membrane computing and in particular we recall the definitions and the main results of two important and well-studied models, the one with symbol-objects and multiset rewriting rules and the one with symport/antiport rules. The reader can use this section as a reference during the reading of the Thesis.

1.1 Formal Languages Preliminaries

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

1.1.1 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 λ (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.

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 $|x|_a$. For a language $L \subseteq V^*$, the set $length(L) = \{|x| \mid x \in L\}$ is called the length set of L.

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 $L_1 \backslash L_2 = \{x \in V^* \mid yx \in L_1 \text{ for some } y \in L_2\}$;
- the *left derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is the set $\partial_x^l(L) = \{w \in V^* \mid xw \in L\}$;
- the *right derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is the set $\partial_x^r(L) = \{w \in V^* \mid wx \in L\}$.

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 $\sigma : V^* \to \mathcal{L}(U^*)$ 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 γ_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*.

We also need to recall the basics of languages of infinite words *(ω-languages)*.

When we speak about infinite words, w^ω denotes the infinite catenation of w to the right and this is the only type of infinite words we consider. Thus V^ω is the set of all infinite words (over the alphabet V), and $V^\infty := V^* \cup V^\omega$ is the set of all words. Its subsets are called ω-languages. We call an infinite word w *periodic*, if there exists a finite word u such that $w = u^\omega$; if $w = vu^\omega$, where also v is a finite word, then w has the weaker property of *ultimate periodicity*.

ω-regular languages are described by ω-regular expressions. Formally, we define them and their corresponding languages (their interpretations ϕ) as follows:

- if a is in V, then a is an expression; its interpretation is $\{a\}$;
- if e_1 and e_2 are expressions, so is $e_1 \circ e_2$; its interpretation is $\phi(e_1) \cdot \phi(e_2)$;
- if e_1 and e_2 are expressions, so is $e_1 \vee e_2$; its interpretation is $\phi(e_1) \cup \phi(e_2)$;
- if e is an expression, so is e^*; its interpretation is $\phi(e)^*$;
- if e is an expression, so is e^ω; its interpretation is $\phi(e)^\omega$.

There are no other expressions. The operator $^\omega$ is the infinite iteration for a set, which formally transforms a set X of words to $X^\omega = \{u_0 u_1 \ldots \mid i \geq 0, u_i \in X\}$. For the reader unfamiliar with this operation the following example might illustrate the difference to finite iteration.

The language $\phi(a^\omega \circ b^\omega)$ is not equal to $\phi(a^* \circ b^\omega)$, because in the first case the a is iterated infinitely often, and therefore the b is never reached, the language equals $\phi(a^\omega)$. The second language consists of all infinite words starting with finitely many a and then continuing with infinitely many b.

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 identify 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, we assume that the concatenation $w_1 w_2$ denotes the multiset obtained by the union of w_1 and w_2.

1.1.2 Chomsky Grammars

A *Chomsky grammar* 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 operations 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 $A \in N, v \in (N \cup T)^*$.
- *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 also denote by FIN the family of finite languages. Among these families, the following strict inclusions (describing the *Chomsky hierarchy*) hold:

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

We now define a variant of context-free grammars, for which the order of the symbols on the rules' right sides does not matter. Therefore we call them locally commutative context-free grammars (abbreviated as LCCF). They are defined exactly as conventional context-free grammars, only with the additional condition that for any rule $A \to u$ in the rule set, also all rules $A \to \pi(u)$ are in the rule set, where $\pi(u)$ is a permutation of u.

Instead of writing a list of all these rules, we write each rule in the form $A \to [B_1, \ldots, B_n]$, where the B_i can be terminals or non-terminals. The rule is applied by replacing A in a sentential form by a string of all the B_i in an arbitrary order. Right sides of length one can also be written without brackets. As one might expect, this variant is significantly weaker than general context-free grammars.

Theorem 1.1 *The family of locally commutative context-free languages is a proper subfamily of the family of context-free languages, i.e., $LCCF \subset CF$.*

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

We consider here some characterizations, in terms of equivalent grammars, for context-free and type-0 languages, which are useful in the proofs of the following chapters (two grammars are *equivalent* if they generate the same language).

Theorem 1.2 (λ-free form) *For each context-free grammar, one can construct an equivalent context-free grammar G' such that the right-hand sides of its productions are all different from λ except when $\lambda \in L(G)$. In this latter case, $S \rightarrow \lambda$ is the only rule with the right-side λ but S does not appear on any right-side of the rules.*

A grammar is said to be $\lambda - free$ if none of its rules has the right-hand side λ, possibly excepting $S \rightarrow \lambda$ as above.

Theorem 1.3 (Chomsky normal form) *For each λ-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 $A \rightarrow BC$ or $A \rightarrow a$, with $A, B, C \in N$ and $a \in T$.*

Theorem 1.4 (Greibach normal form) *For each λ-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 $A \rightarrow aX$ with $A \in N, a \in T$, and $X \in N^*$.*

Theorem 1.5 (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 $A \rightarrow BC, A \rightarrow a, A \rightarrow \lambda, AB \rightarrow CD$, with $A, B, C, D \in N$ and $a \in T$.*

Consider the grammar $G = (N, T, S, P)$ and consider a derivation D according to G,

$$D : S = w_0 \Longrightarrow w_1 \Longrightarrow \cdots \Longrightarrow w_n = w.$$

The workspace of w by the derivation D is:

$$WS_G(w, D) = max\{|w_i| \mid 0 \leq i \leq n\}.$$

The workspace of w is:

$$WS_G(w) = min\{WS_G(w, D) \mid D \text{ is a derivation of } w\}.$$

Observe that $WS_G(w) \geq |w|$ for all G and w.

The *workspace theorem* is a powerful tool in showing languages to be context-sensitive.

Theorem 1.6 (Workspace theorem) *If G is a type-0 grammar and if there is a nonnegative integer k such that*

$$WS_G(w) \leq k|w|$$

for all nonempty words $w \in L(G)$, then $L(G)$ is a context-sensitive language.

Finally we recall that, for context-free grammars, it is possible to prove that

Theorem 1.7 *Given a context-free grammar G then, for each word $w \in L(G)$, there is a leftmost derivation in G.*

1.1.3 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* and *programmed* grammars that are the two devices used in the proofs given in the Thesis.

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 the rule $A_i \to x_i$ 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 1.8

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

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.

Theorem 1.9 *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 1.10 *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-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 $m_{n+1} : (Z \to \lambda)$.

A context-free *programmed grammar* with appearance checking is a construct $G = (N, T, S, P)$, where N, T, S are the set of non-terminals, the set of terminals, and the start symbol, and P is a finite set of rules of the form $(b : A \to x, E_b, F_b)$, where b is a label, $A \to x$ is a context-free rule over $N \cup T$, and E_b, F_b are two sets of labels of rules of G (E_b is called the *success field* and F_b the *failure field* of the rule). If the failure field is empty for any rule of P, then the grammar is without appearance checking. We denote $Lab(P) = \{b \mid (b : A \to x, E_b, F_b) \in P\}$.

The language $L(G)$ generated by G is defined as the set of all the words $w \in T^*$ such that there is a derivation

$$S = w_0 \Rightarrow_{b_1} w_1 \Rightarrow_{b_2} w_2 \Rightarrow_{b_3} \ldots \Rightarrow_{b_k} w_k = w,$$

$k \geq 1$, and, for $(b_i : A_i \to x_i, E_{b_i}, F_{b_i})$, $1 \leq i \leq k$, one of the following conditions hold: $w_{i-1} = w'_{i-1} A_i w''_{i-1}, w_i = w'_{i-1} x_i w''_{i-1}$ for some $w'_{i-1}, w''_{i-1} \in (N \cup T)^*$ and $b_{i+1} \in E_{b_i}$ or A_i does not occur in w_{i-1}, $w_{i-1} = w_i$ and $b_{i+1} \in F_{b_i}$.

In other words, a rule $(b_i : A_i \to x_i, E_{b_i}, F_{b_i})$ is applied as follows: if A_i is present in the sentential form, then the rule is used and the next rule to be applied is chosen from those with the label in E_{b_i}, otherwise, the sentential form remains unchanged and we choose the next rule from the rules labeled by some element of F_{b_i}, and try to apply it.

By PR we denote the family of languages generated by programmed grammars without appearance checking and by PR_{ac} we denote the family of languages generated by programmed grammars with appearance checking.

The following theorem is true.

Theorem 1.11 $MAT = PR \subset MAT_{ac} = PR_{ac} = RE$.

If, for a given programmed grammar with appearance checking, we have that success and failure fields coincide for each production, then the grammar is called *with unconditional transfer*.

We recall that such grammars are able to generate the recursively enumerable one-letter languages (denoted by RE_1).

Theorem 1.12 $PR_{ut} = RE_1$.

1.1.4 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 *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 table contain 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 1.13

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

There is a list of languages which are not in given L families:

$L_1 = \{a^m b^n a^m \mid 1 \leq m \leq n\} \notin E0L$,
$L_2 = \{a^{2^n 3^m} \mid n, m \geq 1\} \notin E0L$,
$L_3 = \{a^n \mid n \text{ is a prime number}\} \notin ET0L$,
$L_4 = \{(ab^n)^n \mid n \geq 1\} \notin ET0L$,
$L_5 = \{(ab^n)^m \mid 1 \leq m \leq n\} \notin ET0L$,
$L_6 = \{(ab^m)^n \mid 1 \leq n \leq m\} \notin ET0L$,
$L_7 = \{w \in \{a, b\}^* \mid |w|_a = 2^n, |w|_b = 3^n, n \geq 0\} \notin ET0L$,
$L_8 = \{w \in \Sigma^* \mid |w| = k^n, n \geq 0\} \notin EDT0L$,
$L_9 = \{w \in \Sigma^* \mid |w| = n^k, n \geq 0\} \notin EDT0L$.

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

Theorem 1.14 *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 strengthening normal form for tabled Lindenmayer systems can also be obtained.

Theorem 1.15 *(Normal Form)*
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).

An *Indian parallel grammar* is a quadruple $G = (N, T, S, P)$, where N, T, S and P is a set of rules specified as in a context-free grammar. Let $V = N \cup T$. For two strings $x, y \in V^*$, we say that x directly derives y, denoted by $x \Longrightarrow y$, if and only if $x = x_1 A x_2 A \dots x_n A x_{n+1}$, for some $x_i \in (V - \{A\})^*, A \in N, n \geq 0, 1 \leq i \leq n+1$, $y = x_1 w x_2 w \dots x_n w x_{n+1}$ and $A \to w$ is in P. That is, a derivation step consists in the substitution of all occurrences of one non-terminal symbol according to the same production.
The language generated by G is defined as $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$.

We denote by IP the family of languages generated by Indian parallel grammars with erasing rules. The following results hold:

Theorem 1.16

- $IP \subset EDT0L$;
- the families IP and CF are incomparable.

1.1.5 Automata Theory

Automata are computing devices which work like accepting devices: they can *decide* whether the 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 and an automaton used for accepting languages of infinite words.

A *(nondeterministic) 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 $x \in V^*, a \in \Sigma$ 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 $L(A) = \{w \in V^* \mid \widehat{\delta}(q_0, w)$ contains a state in $F\}$.

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

Similarly, all ω-regular languages of infinite words are recognized by several types of automata. One of these is the Büchi automaton.

A *Büchi automaton* is a 5-tuple $A = (Q, V, \delta, I, F)$, where the set of states Q, the alphabet V, the transition function $\delta : Q \times V \times Q$, and the sets of initial and final states I and F are as in a finite automaton. The only difference is in the mode of acceptance.

To describe this mode we further define for every infinite computation c of the Büchi automaton the set $Inf(c)$ of states which are visited infinitely often. Then the set $L^\omega(A)$ of words accepted by A is the set of words which label an infinite computation c starting in an initial state and such that $Inf(c) \cap F \neq \emptyset$; that is, the computation visits at least one final state infinitely often. Such sets $L^\omega(A)$ we will call recognizable.

Büchi automata characterizes the family of subsets of V^ω that are ω-regular languages.

Because in biological simulations a great interest is attributed to the question whether a behavior is periodic, we further cite and summarize the following results concerning periodicity and recognizable languages:

Theorem 1.17 *The following statements are true:*

1. *Every nonempty recognizable subset of V^ω contains an ultimately periodic word.*
2. *Let $X, Y \subset V^\omega$ be recognizable and let $U \subset V^\omega$ be a set of ultimately periodic words. If $X \cap U \subset Y$, then $X \subset Y$. This means that X and*

Y are equal if and only if they contain the same ultimately periodic words.

Theorem 1.18 *A set of infinite words is ω-regular if and only if it is a finite union of sets of the form XY^ω, where X and Y are regular subsets of V^*.*

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.

Formally, a (*non-deterministic*) *Turing machine* is a construction $M = (Q, V, \Gamma, \flat, q_0, F, \delta)$ where Q is a finite set of states, V is the tape alphabet, $\Gamma \subseteq V$ is the input alphabet, $\flat \in V - \Gamma$ is the blank symbol, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and δ is a function from $Q \times V$ to $\mathcal{P}(Q \times V \times \{L, R\})$, with L, R symbols not in V.

The mapping δ describes the moves of the machine: if $(q', b, D) \in \delta(q, a)$ for some $q, q' \in Q, a, b \in V, D \in \{L, R\}$, then the machine reads the symbol a in the state q and changes its state to q', it replaces the symbol a with the symbol b and moves the head to the left if $D = L$, to the right if $D = R$. If $card(\delta(q, a)) \leq 1$ for all $q \in Q, a \in V$, then the Turing machine is said to be *deterministic*.

Turing machines characterize the family of languages generated by type-0 grammars and, in addition to being language acceptor, 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 *configuration* of a Turing machine M is a triple $\langle x, q, y \rangle$, where $x \in V^*$ is the string written on the tape, $q \in Q$ is the state of M, and $y \in V^*(V - \{\flat\}) \cup \{\lambda\}$ is the portion of the string whose left-end is currently scanned by the head. Note that both x and y may be empty, the blank symbol may appear in x, y but not in the last position of y. We can define a *direct transition* relation $\vdash_M$ among two configurations of M (that is, a step of the computation) in the following way:

- $\langle x, q, ay \rangle \vdash_M \langle xb, q', y \rangle$ if and only if $(q', b, R) \in \delta(q, a)$ (right move, using a symbol $a \in V$);
- $\langle x, q, \flat \rangle \vdash_M \langle xb, q', \flat \rangle$ if and only if $(q', b, R) \in \delta(q, \flat)$ (right move, using the blank symbol $\flat$);
- $\langle xc, q, ay \rangle \vdash_M \langle x, q', cby \rangle$ if and only if $(q', b, L) \in \delta(q, a)$ (left move, using a symbol $a \in V$);
- $\langle xc, q, \flat \rangle \vdash_M \langle x, q', cb \rangle$ if and only if $(q', b, L) \in \delta(q, \flat)$ (left move, using the blank symbol $\flat$),

where $x, y \in V^*, a, b, c \in V - \{\flat\}, q, q' \in Q$. We denote by $\vdash_M^*$ the reflexive and transitive closure of the relation $\vdash_M$.

The language recognized by a Turing machine M is defined as $L(M) = \{w \in \Gamma^* \mid \langle \lambda, q_0, w \rangle \vdash_M^* \langle x, q, y \rangle$ for some $x, y \in V^*, q \in F\}$, that is, the set of input strings over Γ for which the machine reaches a final state, starting from the initial state and scanning the first symbol of the input string.

There is an equivalent mode of defining the language accepted by a Turing machine: one considers the set of strings $w \in \Gamma^*$ 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 an 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 in input a string w over V, halts and accepts if the string w is the language L, otherwise halts and rejects.

When working on an input string a Turing machine is allowed to use as much as tape 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 computing devices largely used in the P systems area: *register machines.*

An n-register machine is a construct $M = (n, R, l_0, l_h)$ where n is the number of registers (each register stores an arbitrary natural number); R (the *program* of the machine) is a finite set of labeled instructions of the form $(l_1 : op(r), l_2, l_3)$ where $op(r)$ is an operation on register r of M, $l_1, l_2, l_3 \in Lab(R)$ (where $Lab(R)$ denotes the set of labels of the instructions from R); l_0 is the initial label; l_h is the final label.

The operations allowed by an n-register machine are:

• $(l_1 : \text{ADD}(r), l_2, l_3)$ – increment the value stored into register r and proceed, in a non-deterministic way, to the instruction labeled l_2 or to the instruction labeled l_3 ($l_2 = l_3$ for the deterministic variant and then the instruction is written in the form $(l_1 : \text{ADD}(r), l_2)$);

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

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

A register machine $M = (n, R, l_0, l_h)$ accepts a vector $(r_1, \cdots, r_{n-2}) \in \mathbb{N}^{n-2}$ iff, starting from the instruction labeled l_0, with register j having value r_j for $1 \leq j \leq n-2$, and the contents of registers $n-1$ and n being empty, the machine halts.

It is known that deterministic register machines accept exactly the family of Turing computable sets of vectors of natural numbers ($PsRE$).

We also will use *two-counter automata.* These are automata with an input tape, which they can read from left to right only; they have two counters, the operations for which are increment by one, decrement by one or remain unchanged. At each step, the automaton is in a certain state of Q, a symbol in V is read from the input tape, and it checks if the two counters are zero or greater than zero. Then the automaton assumes a new state, moves the input head and the values of the two counters are changed, if required. An input word is accepted if it is completely read, and the automaton stops in a final state with both counters being empty. Then the transitions are elements of a mapping

$$\delta : Q \times V \times \{z, nz\} \times \{z, nz\} \mapsto Q \times \{\text{stay}, R\} \times \{+1, 0, -1\} \times \{+1, 0, -1\}.$$

This is the format we will refer to, when simulating a two-counter machine. The meanings of the components is the following: Q is the set of states, and V is the input alphabet. z and nz mean respectively that the counter is zero or not zero, *stay* and R (right) are the movement orders for the head of the input tape, $+1, 0, -1$ are the orders for the changing of the value of the counters. These machines characterize the family RE.

Automata have been introduced also as accepting devices for multisets. An MFA (*multiset finite automaton*) is a structure

$$A = (Q, V, \delta, q_0, F)$$

of a finite non-empty set of states Q, an input alphabet V, an initial state q_0, and a set of final states F; δ is the transition mapping $\delta : Q \times V \mapsto \mathcal{P}(Q)$. The automaton has an input bag, in which the input multiset is placed. The multiset is accepted if the computation of the automaton halts in a final state and the bag is empty, very much like for conventional finite automata.

An MFA is deterministic if its transition function fulfills the following conditions:

1. $|\delta(q, x)| \leq 1$ for all $q \in Q$ and $x \in V$, and
2. if $|\delta(q, x)|$ is non-empty for some $x \in V$, then it is empty for all other letters.

Such deterministic MFAs are abbreviated $DMFA$s. Finally, both types of device can be equipped with the ability to check whether a symbol is (still) contained in the bag. For this the alphabet V is extended by a barred copy $\overline{V} = \{\overline{x} \mid x \in V\}$. The barred letters are not contained in the multiset in the automaton's bag, rather a transition reading a barred letter can be applied only if the corresponding unbarred letter is currently not contained in the multiset. This ability is signified by adding a D for detection to the end of the automaton's abbreviation providing us with the classes $DMFAD$ and $MFAD$. For these automata the second condition for being deterministic is changed to

> (ii.d) if $|\delta(q, x)|$ is non-empty for some $x \in V$, then it is empty for all other letters and their barred counterparts.

We now summarize the relations between the four classes of automata defined.

Theorem 1.19 $DMFA \subset DMFAD \subset MFAD = MFA = PsREG$.

Here $PsREG$ is the family of Parikh sets of regular languages.

1.2 Cells and Membranes

In this section we recall some basic notions concerning living cells and cell membranes. The reader can use this section as short reference for the biological arguments used through all the Thesis.

According to the standard cell biology all living things are composed of one or more cells. 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).

The cell membrane functions as a semi-permeable barrier, allowing a very few molecules across it while fencing the majority of organically produced chemicals inside the cell. Electron microscopic examinations of cell membranes have led to the development of the lipid bilayer model (also referred to as the fluid-mosaic model). The most common molecule in the model is the *phospholipid*, which has a polar (hydrophilic) head and two nonpolar (hydrophobic) tails. These phospholipids are aligned tail to tail so the nonpolar areas form a hydrophobic region between the hydrophilic heads on the inner and outer surfaces of the membrane. See Figure 1.1.

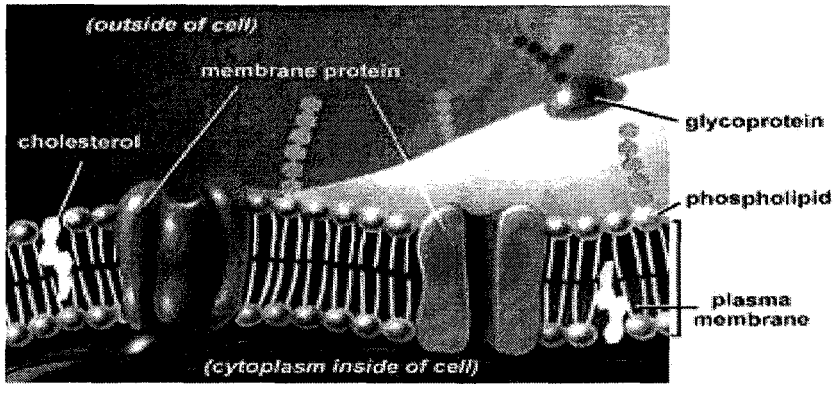

Figure 1.1: Cell Membrane. From [149].

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 will, in exchange for a "price", 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.

Bacteria have cell walls containing peptidoglycan. Plant cells have a variety of chemicals incorporated in their cell walls. Cellulose is the most common chemical in the plant primary cell wall. Some plant cells also have lignin and other chemicals embedded in their secondary walls. The cell wall is located outside the plasma membrane. Plasmodesmata are connections through which cells communicate chemically with each other through their thick walls. Fungi and many protists have cell walls although they do not contain cellulose, rather a variety of chemicals (chitin for fungi).

The compartments of an animal cell is depicted in Figure 1.2.

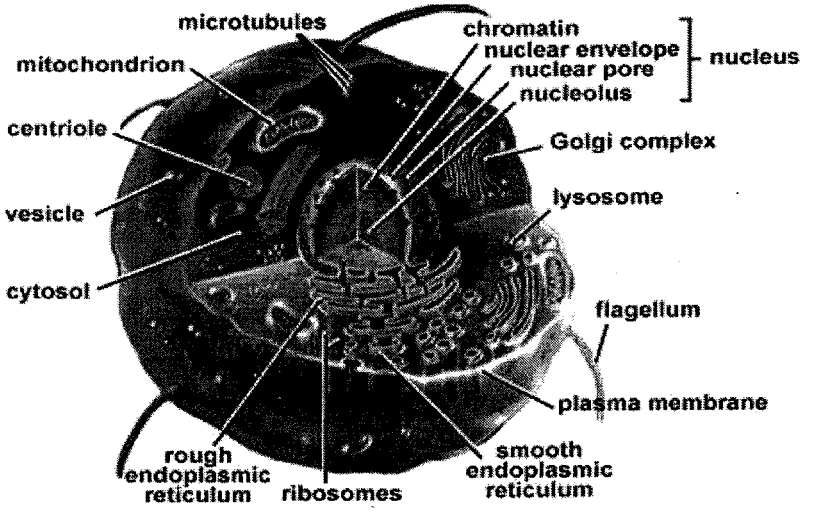

Figure 1.2: The Structure of an animal cell. From [149].

As mentioned earlier, the cell membrane functions as a semi-permeable barrier, allowing a very few molecules across it while fencing the majority of organically produced chemicals inside the cell.

Anyway, transport of chemical substances across the regions is crucial for the life of cells. There are two kinds of transport of chemicals through a membrane: *passive* and *active*.

Passive transport requires no energy from the cell. Examples include the diffusion of oxygen and carbon dioxide, osmosis of water, and facilitated diffusion. Passive transport takes place when certain molecules are present in different concentration in two compartments divided by a membrane. In this case the molecules pass from the compartment with higher concentration to the one with lower concentration.

Active transport requires the cell to spend energy, usually in the form of ATP. In all cells, the energy storage and utilization process involves the *proton-motive force* in some step. This can be described as the storing of energy as a combination of a proton and voltage gradient across a membrane.

Examples of active transports include the moving of large molecules (non-lipid soluble) and the sodium-potassium pump. Active transport is performed by *carrier proteins*, where each protein transports a specific class of molecules. Some carrier proteins transport a chemical from one side of the membrane to the other and they are called *uniporters*.

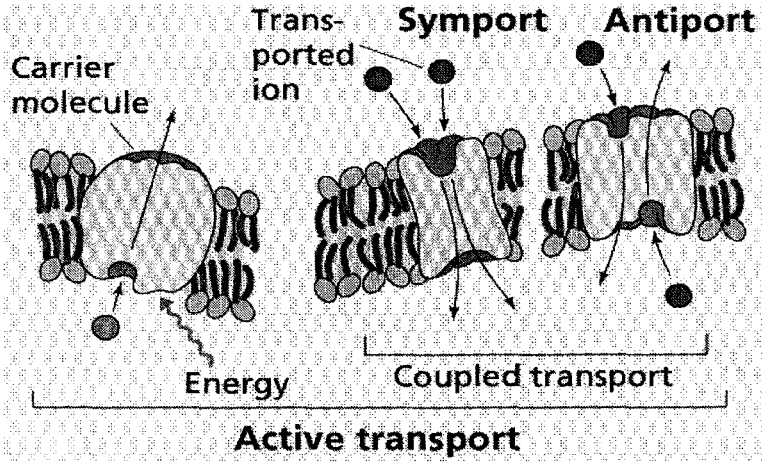

Figure 1.3: Types of active transports. From [149].

Other carrier proteins function as coupled carriers, that means the transfer of one chemical is coupled with the transport of another one. Coupled transport involves either the simultaneous of a second chemical in the same direction (*symport*) (one can be a proton, previously moved) or the simultaneous exchange in the opposite direction (*antiports*) (also in this case one can be a proton previously moved).

The different types of active transports are described in Figure 1.3.

1.3 Membrane Computing Preliminaries

Membrane systems (P systems) have been introduced by Gh. Păun in 1998 and since then a great number of papers (currently almost 700) have been published in the field, with the introduction of several different models. In this section we give a general (and short) overview about membrane systems, recalling the definitions and the main results of two important and well-studied types of this model of computation, the one with symbol-objects and multiset rewriting rules and the one with symport/antiport rules.

As mentioned before, membrane systems are a new model of computation inspired by the cellular structure and functioning.

The main component of this new model of computation is the membrane structure consisting of membranes hierarchically embedded in the outermost skin membrane. Each membrane delimits a region and each region contains a multiset of objects and possibly other membranes. Each region has an associated set of operators working on the objects contained by such region and these operators can be of a different type and they can have different features. As a general idea, each operator works on the multiset of objects in the region and it can "change" the objects (in this case the operator is usually called "evolution rule") and/or transfers the objects from a region to another one (in this case the operator is referred to as "transport rule"). At the beginning of a computation the regions of a P system contains some multisets of objects and some operators and, under given operations, the contents of the regions change and these changes define a computation. There could be different ways to consider a result of a computation; for instance, it could be the number of objects stored in a certain specified region at the end of the computation.

Several types of P systems have been considered, which have introduced in the area several features, like priority, membrane thickness, membrane dissolution and membrane division; many types of P systems have been shown to be universal.

1.3.1 Membrane Systems with Symbol-Objects

We recall now the 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 is present an unique external membrane, called the *skin membrane*. An *elementary membrane* is a membrane without any membrane inside, a *region* enclosed by a membrane is the space between a membrane and the immediately inner membranes (if any), and to each membrane is associated a *label*. Because there is a one-to-one correspondence between membranes and (enclosed) regions we can also understand 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 1.4.

Definition 1.1 *A P system with symbol-objects and of degree $m \geq 1$ is defined as*

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

where

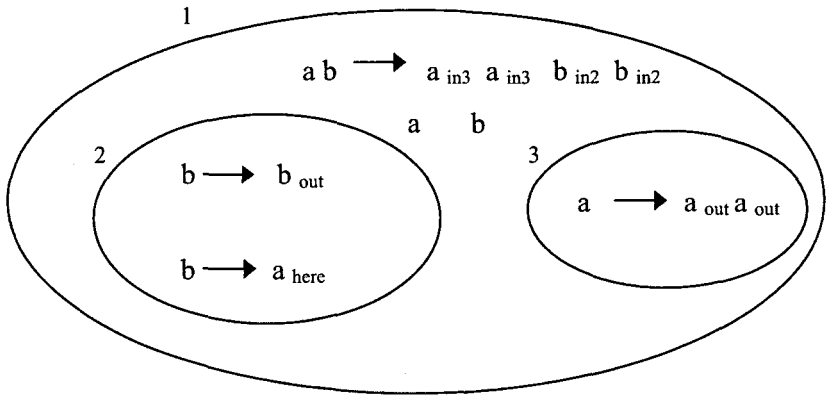

Figure 1.4: A membrane system with symbol-objects and evolution rules. Membranes 2 and 3 are elementary. Membrane 1 is non elementary and represents the skin membrane.

- *O is an alphabet and its elements are called* objects*;*
- *μ 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 μ;*
- *R_i, $1 \leq i \leq m$, are finite sets of* evolution rules *over O; R_i is associated with the region i of μ; 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 $u \rightarrow v$ 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: $a \rightarrow v$ and $ca \rightarrow cv$ (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 Π is constituted by the membrane structure μ and the multisets represented by the strings $w_i, 1 \leq i \leq m$. In general, we call a *configuration* of the system the $m - tuple$ of multisets of objects present at any time in the m regions of the system.

A transition between configurations is executed using the evolution rules in a *non-deterministic maximally parallel manner* at each step (we suppose that a global clock exists, marking the time of each step for the whole system). This means that the objects are assigned with 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 non-deterministic way.

The application of an evolution rule $u \rightarrow v$ 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 $u \rightarrow v$ cannot be applied).

In other words, by using the evolution rule $u \rightarrow v$ in a region i that contains a multiset identified by m, we subtract the multiset identified by u from the multiset identified by m, 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 Π 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 Π; the set of

numbers computed (or generated) in this way by the system Π, 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 in the output region in the halting configuration. In this case $Ps(\Pi)$ denotes the set of vectors of numbers generated by Π, considering all the successful computations.

When we consider a class of P systems then 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 symbol-objects 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 α.

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/or 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.

Here 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 1.20 $NOP_*(ncoo, tar) = NOP_1(ncoo) = NCF$.

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 1.21 $NOP_*(coo, tar) = NOP_m(coo, tar) = NRE$ *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 1.22 $NOP_2(cat_2, tar) = NRE$.

Special 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 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 1.23 $NOP_1(2cat_1, tar) = NRE$.

1.3.2 Membrane Systems with Symport/Antiport Rules

Another well-known model of P systems is the one using *symport/antiport* rules, where the only available operation is the movement of objects through the membranes of the system. In this model specific groups of objects can pass (together) through a membrane either in the same direction (in this case the operation is called *symport*) or in opposite direction (in this case the operation used is called *antiport*).

The idea of this new model is directly inspired by the biological mechanisms used in the cell to transport chemicals through the membranes as described in Section 1.2.

Now we recall the definition of the model and we present the main results obtained in the literature.

Definition 1.2 *A P system with symport/antiport rules and of degree* $m \geq 1$ *is defined as*

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

where

- O *is an alphabet and its elements are called* objects*;*
- μ *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 the regions* $1, 2, \cdots, m$ *of* μ*;*
- $E \subseteq O$ *is the set of objects which are supposed to appear in the environment in arbitrarily many copies;*

- $R_1, \cdots, R_m$ *are finite sets of* symport/antiport rules *over* O *associated with the membranes* $1, 2, \cdots, m$ *of* μ*; a* symport rule *is of the form* (x, in) *or* (x, out)*, while an* antiport rule *is of the form* $(x, in; y, out)$*, where* x, y *are strings representing multisets of objects of* O*. For a symport rule* (x, in) *or* (x, out) *we say that* $|x|$ *is the* weight *of the rule. In case of an antiport rule* $(x, in; y, out)$ *the* weight *of a rule is defined as* $\max\{|x|, |y|\}$*;*

- $i_0 \in \{1, 2, \cdots, m\}$ *is the label of an* elementary *membrane that encloses the* output region.

In a P system with symport/antiport rules, a configuration is represented by the membrane structure μ and by the m-tuple of multisets of objects present in the m regions of the system.

In particular, the *initial configuration* is constituted by the membrane structure μ and the multisets represented by the strings $w_i, 1 \leq i \leq m$. The objects of E are initially present in the environment in an infinite and unbounded supply (their number remain infinite irrespective of how many copies have been introduced in the system, during the computation).

A transition between two configurations is executed by using the symport/antiport rules in a *non-deterministic maximally parallel manner* at each step (we suppose that a global clock exists, marking the time of each step for the whole system).

The application of the *symport rule* (x, in) to membrane i means to move in region i the objects represented by x, from the region (or from the environment) surrounding the region i. If the symport (x, out) is applied to membrane i, then the objects represented by the multiset x move out from region i to the region (or to the environment) that surrounds the region i.

If the *antiport rule* $(x, in; y, out)$ is applied to membrane i, then the objects represented by multiset x pass into the region i from the region surrounding it, while, at the same time, the objects represented by the multiset y move in the opposite direction.

A sequence of transitions is called a *computation*, and a computation is considered *successful* (or *halting*) if it starts in the initial configuration and if it ends in a *halting configuration* (a configuration where no rule can be applied, for any membrane).

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

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

When we consider a class of P systems then we denote by $NPP_m(j,k)$ $[PsPP_m(j,k)]$ the family of subsets of [vectors of] numbers defined by P systems with symport/antiport rules having at most m membranes (if the degree is not bounded, then the subscript m becomes $*$), symport rules of weight at most j, and antiport rules of weight at most k.

The following results are of interest for the work present in this Thesis.

Every recursively enumerable set of vectors of natural numbers can be generated by P systems using symport and antiport rules of weight one and three membranes.

Theorem 1.24

$$PsPP_3(1,1) = PsRE.$$

Universality can be obtained also by using only symport rules of weight at most two and three membranes.

Theorem 1.25

$$PsPP_3(2,0) = PsRE.$$

On the other hand, the following result is true.

Theorem 1.26 $NPP_1(2,0)$ *consists of finite sets only.*

1.4 Final Remarks

The reader can find further details concerning formal languages in the classical books, for instance, [88], [138], and [141].

In particular, Theorems 1.2, 1.3, 1.4, and 1.7 can be found in [88] while Theorem 1.5 can be found in the fourth chapter of the second volume of [138]. The workspace theorem, due to N.D. Jones, can be found in the cited books or in the original paper [90].

Locally commutative grammars have been introduced in [54] where the reader can also find the proof of Theorem 1.1.

The reference for the notions and results from the regulated rewriting area is the monograph [68]. The series of results for matrix grammars given in Theorem 1.8 are from [68].

Theorem 1.9 concerning the binary normal form for matrix grammars is from [68]. The Z-binary normal form for matrix grammars (and Theorem 1.10) has been introduced in [104], recalled in [117], and then used in several proofs in the area.

The proof of the universality of programmed grammars with appearance checking given in Theorem 1.11 can be found in [68]. The universality proof for programmed grammars with unconditional transfer, Theorem 1.12, can be found in [71].

Our presentation of ω-regular languages follows closely the introduction of Berstel and Pin [32]. Further details as well as a survey of finite automata

for regular languages of finite words can be found in their book. The proof that Büchi automata characterize ω-regular languages can be found in [32], likewise the proofs of Theorems 1.17 and 1.18.

The general reference for the parallel rewriting section is the book [137] and the dedicated chapter of [138]. The results concerning Lindenmayer systems given in Theorem 1.13 are from [137], and the same for the normal-form Theorem 1.14. The proof of Theorem 1.15 can be found in [9]. The results concerning Indian parallel grammars of Theorem 1.16 can be found in the preliminaries chapter of [33] and in [34].

The notions of automata theory given in Section 1.1.5 can be found in any introductory book to theory of computation, for instance, [88]. Multiset finite automata have been introduced in [66] where the reader can also find the proof of Theorem 1.19.

More information about register machines and the proof of their universality can be found in [109].

Section 1.2 with preliminaries of cellular biology is a short survey of what the reader can find in fifth chapter of the on-line biology book, [149].

If the reader wish to have more details, we suggest to consult the standard biological books, such as [3] and the specific papers, [15, 16], underlying the relevance of biological membranes for the membrane systems area.

Membrane systems have been introduced by Gh. Păun in 1998 in [115]. After less than seven years a huge number of papers can be found in the literature. For a complete overview of the membrane computing field we refer the reader to the monograph [117], and for an update bibliography of the topic the reader can have a look the web page [151]. For a friendly introduction to membrane systems the reader can consult the guide [121].

The definition of P systems with symbol-objects given in Section 1.3.1, together with the proofs of Theorems 1.20 and 1.21 can be found in the book, [117]. The universality of catalytic systems, Theorem 1.22, can be found in [75]. The universality result for systems with bi-stable catalysts, Theorem 1.23, is from [8] (actually there the result is more general, for strings and not set of numbers).

P systems with symport/antiport rules were introduced in [113] where it was also proved a first universality result. The proof of Theorems 1.24 and 1.25 concerning universality of systems using antiport/symport rules of weight one or symport rules of weight at most two can be found in [13]. The proof of Theorem 1.26 concerning systems with only one membrane and symport rules of weight at most two can be found in [79].

A recent survey on the computational power of P systems with symport and antiport rules is the paper [12].

Chapter 2

A First Suggestion from Biology: the Evolution-Communication Model

In the previous chapter the models of P systems with symbol-objects and with symport/antiport rules have been recalled. Both the models have been inspired by biological processes that happen in living cells: evolution of symbol-objects (application of biochemical reactions) and application of symport/antiport rules (movement, across the membranes, of chemical substances).

Actually, in reality, these two processes of evolution and communication are not strictly separated, but they happen together, in a simultaneous and independent way. In fact, while some chemicals are modified by means of biochemical reaction, at the same time, other chemicals are transported across the membranes of this cell, by means of symport/antiport rules (the reader can refer to the preliminaries of cellular biology presented in Section 1.2).

Using these suggestions, a natural idea is to create a P system that combines the evolution (in the form of evolution rules) and the communication (in the form of symport/antiport rules); this is the philosophy of the *evolution-communication model* that we present in this chapter.

In this "hybrid" model the evolution rules lose their target indication (considered not so realistic from a biological point of view) and the system is embedded in an *empty environment* (and *not infinite* as in the case of P system with symport/antiport rules).

The work of an evolution-communication P system is divided in two phases: the *evolution* of the symbol-objects (that consists in the application of the evolution rules) and the *communication* between the regions of the

system (that consists in the application of the symport/antiport rules).

In evolution-communication P systems there can be different approaches to define a computation; in this chapter we prove that evolution- communication P systems are universal when using the *mixed approach* – the computation takes place as a *mixed* "sequence" of evolution rules and symport/antiport rules.

The evolution-communication model is based on the assumption that objects are moved, while rules are assigned in a fixed way to regions (evolution rules) or to membranes (symport/antiport rules). Actually, the situation might be different. In cells, reactions are catalyzed/driven/controlled by chemicals which can move through membranes. In a certain sense the migration of a chemical is like the migration of the reaction catalyzed by such chemical. Therefore it is natural to consider also P systems with migration of rules.

Such model can be obtained following the philosophy of the evolution-communication model: we combine, this time in a more "exotic" way, P systems with evolution rules and P systems with symport/antiport rules.

We present *P systems with symport/antiport of rules* that are P systems with evolution rules used to evolve the symbol-objects in the classical way (without communication targets) and symport/antiport rules used to move the evolution rules across the membranes.

We recall the definition of this model, some examples that illustrate its functioning and the current results present in literature.

2.1 Evolution-Communication P Systems

The evolution-communication model is constructed by combining evolution rules without communication targets (or, in other words, with all the communication targets fixed as "here") called *simple evolution rules* and symport/antiport rules. The formal definition for this model follows.

Definition 2.1 *An* evolution-communication P system *(in short,* an EC P system*), of degree $m \geq 1$, is defined as*

$$\Pi = (O, \mu, w_1, w_2, \cdots, w_m, R_1, \cdots, R_m, R'_1, \cdots, R'_m, i_0),$$

where:

- *O is the alphabet of objects;*
- *μ is a membrane structure with m membranes (and hence m regions) injectively labeled with $1, 2, \cdots, m$;*
- *w_i are strings which represent multisets over O associated with the regions $1, 2, \cdots, m$ of μ;*

- *R_i, $1 \leq i \leq m$, are finite sets of simple evolution rules over O; R_i is associated with the region i of μ; a simple evolution rule is of the form $u \to v$, where u and v are strings over the alphabet O;*

- *R'_i, $1 \leq i \leq m$, are finite sets of symport/antiport rules over O; R'_i is associated with the membrane i of μ;*

- *$i_0 \in \{0, 1, 2, \cdots, m\}$ is the label of a region which is designed as the output region; if $i_0 = 0$, then the output region is the environment.*

As usual, the m-tuple of multisets of objects present at any moment in the regions of Π and the membrane structure μ represent the configuration of the system at that moment; so, the m-tuple $(w_1, \cdots, w_m)$ and the structure μ represent the *initial configuration* (the environment is empty). A transition between configurations is governed by the mixed application of the evolution rules and of the symport/antiport rules.

The rules from R_i are applied to objects in region i and the rules from R'_i govern the communication of objects through membrane i. There is no difference between evolution rules and communication rules (*mixed approach*): they are chosen and applied in the *non-deterministic maximally parallel manner.*

The system continues parallel steps until there is no applicable rules (evolution rules or symport/antiport rules) in any region of Π; the system halts and the configuration reached is *halting.*

In this case the computation is *successful* and the number of objects contained in the output region i_0 in the halting configuration is the result of the computation of Π. We can also consider as result of the computation the vector of numbers representing the multiplicities of the objects contained in the output region in the halting configuration.

The following notation

$$NECP_m(i, j, \alpha), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}$$
$$[PsECP_m(i, j, \alpha), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\}]$$

is used to denote the family of sets of numbers [the family of sets of vectors of numbers] generated by EC P systems with at most m membranes (as usually, $m = *$ if such a number is unbounded), using symport rules of weight at most i, antiport rules of weight at most j, and simple evolution rules that can be cooperative (coo), non-cooperative ($ncoo$), or catalytic (cat_k), using at most k catalysts.

2.1.1 Computational Power

Now we recall the main results obtained for EC P systems. In particular, we present an universality proof for the model; the proof is interesting, not only

for the result obtained but also for the idea of the proof: it shows how a new model of computation, biologically inspired, can simulate a very old model of computation like programmed grammars with appearance checking.

A first result for EC P systems is obtained by observing that communication targets can, obviously, simulate the control of communication as done by a symport rule of weight 1; therefore

$$NECP_m(1,0,\alpha) \subseteq NOP_m(\alpha, tar),$$

for all $\alpha \in \{coo, ncoo\} \cup \{cat_k \mid k \geq 0\}$.

Moreover, in the case of $NOP_1(ncoo, tar)$, the role of the target indications is only to send out (this means to "lose") some symbol-objects, using target indications of the form a_{out}, where a is a symbol-object. We can simulate the evolution rules which contain such a target indication by replacing each a_{out} with λ; in this way, we can avoid to use target indications. Using the result presented in Theorem 1.20, we obtain:

$$NOP_1(ncoo, tar) \subseteq NECP_1(0,0,ncoo) \subseteq NECP_1(1,0,ncoo).$$

Then we can write

Theorem 2.1 $NECP_1(1,0,ncoo) = NCF.$

The equality shown in the last theorem is no longer valid if more powerful ingredients are used: symport/antiport rules of weight one (that can be considered "realistic" from a biological point of view as described in Section 1.2). In this case the evolution-communication model becomes universal (it can generates any recursive enumerable set of natural numbers).

The idea of the proof is to simulate a programmed grammar with appearance checking and unconditional transfer.

Using Theorem 1.12 we can prove the following result.

Theorem 2.2 $NECP_3(1,1,ncoo) = NRE.$

Proof. Consider a programmed grammar $G = (N, \{a\}, S, P)$ with appearance checking and with unconditional transfer. We add to P the triple $(0 : U \to S, E_0, F_0)$ with $E_0 = F_0 = l(S)$ where U is a new non-terminal and $l(S) = \{k \in Lab(P) \mid (k : S \to x, E_k, F_k) \in P\}$. In this way, we obtain a new grammar, $G' = (N', \{a\}, U, P')$, with $N' = N \cup \{U\}$; clearly, $L(G) = L(G')$ and each derivation in G' starts with the rule with label 0.

For each $x \in N'$ we consider a new symbol $\bar{x}$, we denote by $\bar{N}$ the set of such symbols, and we define the morphism $\gamma : (N' \cup T)^* \longrightarrow (\bar{N} \cup T)^*$ by $\gamma(x) = \bar{x}$, if $x \in N'$, and $\gamma(x) = x$, if $x \in T$.

We construct the EC P system

$$\Pi = (O, \mu, w_1, w_2, w_3, R_1, R_2, R_3, R'_1, R'_2, R'_3, i_0),$$

with:

$$
\begin{aligned}
O &= N' \cup \bar{N} \cup \{d_i, d_i'', d_i''', e_i, e_i' \mid i \in Lab(P')\} \\
&\cup \{a, p_1 \cdots p_5, c, h, \bar{h}, c_1, c_2, \cdots, c_7, c', \#, F, F', f\}; \\
\mu &= [_1[_2[_3\ ]_3]_2]_1; \\
w_1 &= \lambda; \\
w_2 &= FfUh; \\
w_3 &= c; \\
R_1 &= \{e_i \to e_i' \mid i \in Lab(P')\} \cup \{Y \to \# \mid Y \in N'\} \cup \{\# \to \#\}; \\
R_2 &= \{\bar{Y} \to Y \mid Y \in N'\} \cup \{\bar{h} \to h, F \to F, \\
&\quad c \to c_1, c_1 \to c_2, \cdots, c_6 \to c_7\} \\
&\cup \{d_i \to d_i', d_i'' \to d_i' e_i \mid i \in Lab(P')\}; \\
R_3 &= \{X \to \gamma(x) d_k \mid (k : X \to x, E_k, F_k) \in P'\} \\
&\cup \{h \to \bar{h} d_i'' p_1, d_i \to \#, d_i'' \to \# \mid i \in Lab(P')\} \\
&\cup \{p_1 \to p_2, p_2 \to p_3, \cdots, p_4 \to p_5, c_7 \to c, c \to c', F \to F'\} \\
&\cup \{\# \to \#\}; \\
R_1' &= \{(a, out)\}; \\
R_2' &= \{(e_i', in; Q, out), (e_i, out), (e_i', in; p_5, out) \mid \\
&\quad (i : Q \to z, E_i, F_i) \in P'\} \\
&\cup \{(F', out), (a, out)\} \cup \{(F', in; Y, out) \mid Y \in N'\}; \\
R_3' &= \{(\bar{Y}, out), (Y, in; c, out) \mid Y \in N'\} \\
&\cup \{(\bar{h}, out), (F', out), (c', out; F, in), \\
&\quad (c, out; h, in), (c_7, in), (a, out)\} \\
&\cup \{(d_j', in; d_i, out), (d_j', in; d_i'', out) \mid i \in E_j\}; \\
i_0 &= 0.
\end{aligned}
$$

The system Π simulates the programmed grammar G' in the following way. In region 3 (the inner one) we simulate the application of the context-free rules of G', using the simple evolution rules

$$X \to \gamma(x) d_k, \text{ for } (k : X \to x, E_k, F_k) \in P',$$

while the symport/antiport rules

$$(d_j', in; d_i, out), (d_j', in; d_i'', out), \text{ for } i \text{ and } j \text{ such that } i \in E_j,$$

associated with membrane 3, take care of the application in the correct order of the rules of G'.

During the simulation, using the symport/antiport rules

$$(Y, in; c, out), \text{ for } Y \in N',$$

associated with membrane 3, we move from region 2 to region 3 the non-terminal that must be rewritten, and after performing such a rewriting, we store in region 2 the objects obtained. Finally, we use region 1 to implement the appearance checking mechanism of G', using the antiport rules

$$(e'_i, in; Q, out), \text{ for } (i : Q \to z, E_i, F_i) \in P',$$

associated with membrane 2.

The evolution rules $F \to F, F \to F', c \to c'$ present in region 2 and 3, together with the antiport rule $(F, in; c', out)$ associated with membrane 3, and with the symport rule (F', out) associated with membranes 3 and 2, are used to make sure that, when the computation halts, there are no objects from N' in region 2.

Now we pass to describe in more details the work of the system Π.

After applying an antiport rule

$$(Z, in; c, out), \text{ for some } Z \in N',$$

associated with membrane 3, the object Z arrives in region 3 and the evolution rule

$$Z \to \gamma(z)d_m, \text{ with } (m : Z \to z, E_m, F_m) \in P',$$

is applied.

The non-terminal objects of $\gamma(z)$ are sent to region 2, using the symport rules

$$(\bar{Y}, out), \text{ for } Y \in N',$$

associated with membrane 3. The object $d_{\bar{m}}$ indicates that the rule of G' with label m has been applied; as soon as such an object has been produced it must leave region 3 (because of the evolution rule that produces the trap symbol #). The object d_m can exit only if in region 2 there exists the "right" predecessor ("right" in the sense of the order defined by the success/failure fields of G'). In fact, to leave region 3 the object d_m must use (one of) the antiport rules associated with membrane 3 of the form

$$(d'_j, in; d_m, out), \text{ for } m \in E_j.$$

Such an antiport rule can be applied only if in region 2 one of the predecessors of d_m is present (notice that the simulated grammar has unconditional transfer and then $E_j = F_j$).

When the object d_m goes in region 2, it is changed in d'_m and this object will store the new predecessor and the computation continues.

If a rule cannot be applied, then we must simulate the appearance checking mechanism of the grammar: we introduce in region 3 the special object

h using the antiport rule $(h, in; c, out)$, associated with membrane 3, and, after that, we can apply the simple evolution rule

$$h \to \bar{h} d''_q p_1, \text{ for some } q \in Lab(P'),$$

present in region 3. The application of this rule means that the context-free rule of G' with label q cannot be applied. The application of this evolution rule produces the objects d''_q, $\bar{h}$, and p_1. The object $\bar{h}$ is immediately sent out to region 2 and the object d''_q must go to region 2, but, as in the previous case, it can pass to region 2 only if the "right" predecessor is there. In fact, to exit from region 3, the object d''_q must use (one of) the antiport rules associated with membrane 3

$$(d'_j, in; d''_q, out), \text{ for } q \in E_j.$$

In this case, when the object d''_q arrives in region 2, the evolution rule $d''_q \to d'_q e_q$, is applied and it creates the new predecessor d'_q and the object e_q that will check if the rule with label q was skipped correctly.

After producing e_q, this object is sent out to region 1, using the symport rule (e_q, out), associated with membrane 2.

As soon as the object e_q arrives in region 1, it is changed to e'_q and, at the next step, the antiport rule

$$(e'_q, in; Q, out), \text{ for } (q : Q \to z, E_q, F_q) \in P',$$

associated with membrane 2, might be applied.

If this happens, then the computation never halts, because the evolution rule $Q \to \#$, present in region 1, is applied (this means that the skipping of the rule with label q was not correct). If this antiport rule is not applied, then the skipping of the rule was correct and, now, the only thing to do is the cleaning of region 1 of the object e'_q. To this aim we use the object p_5 (it will arrive in region 2 only after checking the correctness of skipping a rule), together with the antiport rule

$$(e'_q, in; p_5, out),$$

associated with membrane 2.

After removing the object e'_q from region 1, the simulation of the rules from G' can be repeated. Finally, we observe that the objects $p_1, p_2, \cdots, p_5$ are used to send at the "right" time the object p_5 to region 1, and the objects $c_1, c_2, \cdots, c_7$ are used to "keep busy" the special object c.

In each step, each produced terminal is sent out by the symport rules associated with membrane 1, 2, and 3, and we collect the output in the environment.

Thus, the system Π sends out exactly the symbols of the strings of $L(G) = L(G')$, and, hence, it generates the length set of $L(G)$. □

2.2 P Systems with Symport/Antiport of Rules

The basic idea of P systems with symport/antiport of rules is that, during the computation, objects can be rewritten but not moved, while evolution rules can be moved but only by means of symport/antiport rules. The definition of this model follows closely the one given for the evolution-communication model.

Definition 2.2 *A* P system with symport/antiport of rules *(*a CR P system *in short, where CR comes from "communication of rules") of degree* $m \geq 1$ *is defined as follows:*

$$\Pi = (O, R, l, \mu, w_1, w_2, \cdots, w_m, R_1, \cdots, R_m, R'_1, \cdots, R'_m, i_0),$$

- O *is the alphabet of objects;*
- R *is a finite set of simple evolution rules;*
- l *is an injective labeling of the rules in* R*; let* $L = \{l(r) \mid r \in R\}$*;*
- μ *is a membrane structure with* m *membranes (and hence* m *regions) injectively labeled with* $1, 2, \cdots, m$*;*
- w_i *are strings which represent multisets over* O *associated with the regions* $1, 2, \cdots, m$ *of* μ*;*
- $R_i \subseteq R$, $1 \leq i \leq m$ *are finite sets of simple evolution rules over* O*;* R_i *is associated with the region* i *of* μ*; a simple evolution rule is of the form* $u \rightarrow v$*, where* u *and* v *are strings over the alphabet* O*;*
- R'_i, $1 \leq i \leq m$, *are finite sets of symport/antiport rules;* R'_i *is associated with the membrane* i *of* μ*. A symport rule is of the form* (x, in) *or* (y, out)*, while an antiport rule is of the form* $(x, in; y, out)$*, where* x, y *are strings that represent sets of elements in* L*;*
- $i_0 \in \{0, 1, 2, \cdots, m\}$ *is the label of the membrane that encloses the output region; if* $i_0 = 0$*, then the output region is the environment.*

In a P system with symport/antiport of rules objects never pass through membranes, but they can change to other objects using the simple evolution rules. R is the set of all possible simple evolution rules that the system can use. Each rule in R is labeled with an unique label. During the computation, symport/antiport rules constructed over the set of labels L associated with the rules are used to move the simple evolution rules across the membranes.

A configuration is described by using the $m - tuple$ of multisets of objects, the structure μ, and the evolution rules present in the m regions of the system. So, the *initial configuration* is represented by the $m - tuple$,

$w_1, w_2, \cdots, w_m$, the structure μ, and the rules $R_1, R_2, \cdots, R_m$ associated to the regions.

The evolution rules are present in the regions in the *set sense*, i.e., we cannot have more than one copy of a rule in one region, unlike the case of objects, where the multiplicity is essential.

A transition between two configurations is governed by the *mixed application* of the evolution rules and of the symport/antiport rules. All objects which can evolve through evolution rules should evolve and all evolution rules that can be moved by symport/antiport rules should be moved. However, if an evolution rule acts on an object in a region i, then it cannot be moved in the same step using the symport/antiport rules associated with membrane i. On the other hand, if the evolution rule is moved, then it cannot act on the objects of region i in the same step. Essentially, evolution rules and symport/antiport rules have the same "priority": they are chosen and applied in a *non-deterministic maximally parallel way.*

Evolution rules act on symbol-objects in the standard way; symport and antiport rules work in a standard way, as well, except that they move rules (using their labels) and not objects. If a symport rule (x, in) associated to membrane i is applied, then the evolution rules represented by the string x pass into the region i from the surrounding region. If the symport (x, out) is applied to membrane i, then the evolution rules represented by string x move up from region i to the surrounding region (or to the environment).

Finally, if the antiport rule $(x, in; y, out)$ is applied to membrane i, then the evolution rules represented by x pass into region i from the region surrounding it, while at the same time the evolution rules represented by y move in the opposite direction.

A sequence of transitions is called a *computation* and a computation is considered *successful* (or *halting*) if it starts in the initial configuration and ends in a halting configuration – a configuration where no evolution rule and no symport/antiport rule can be applied in any region and in any membrane, respectively.

The result of a successful computation is the number of objects present at the end of the computation in the output region.

We use the following notation

$$PsCRP_m(i, j, \alpha), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\},$$

to denote the family of sets of vectors of numbers generated by CR P systems with at most m membranes using symport rules of weight at most i, antiport rules of weight at most j (as usually, $m = *, i, j = *$ if such a number is unbounded), and simple evolution rules that can be cooperative (coo), non-cooperative ($ncoo$), or catalytic (cat_k) using at most k catalysts.

For simplicity, in what follows we use the following *notation*: If S is a set, then $x = \langle S \rangle$ means that x is a string representing S (i.e., x is a string

obtained by concatenating elements of S in an arbitrary order, such that each element appears exactly once).

2.2.1 Computing with CR P Systems: Two Examples

In this section we illustrate the computation of a CR P system using two simple examples. In particular, we show how to construct a CR P system generating the Parikh image of the non-semilinear language $\{a^{2^n} \mid n \in \mathbb{N}\}$. In Example 2.1, only antiports of weight 1 are used, while in Example 2.2 we use only symports of weight 2.

Example 2.1 Using Antiport of Rules

We construct the following CR P system of degree 2.

$$\Pi = (O, R, l, \mu, w_1, w_2, R_1, R_2, R_1', R_2', 2),$$

where:

$$\begin{aligned}
O &= \{A, a\};\\
R &= \{A \to a, A \to AA\};\\
&\quad l \text{ is an injective labeling of the rules in } R;\\
&\quad \text{we use } l_1 = l(A \to a), l_2 = l(A \to AA); L = \{l_1, l_2\};\\
\mu &= [_1\, [_2\,]_2\,]_1;\\
w_1 &= \emptyset,\ w_2 = A;\\
R_1 &= \{A \to a\};\\
R_2 &= \{A \to AA\};\\
R_1' &= \emptyset;\\
R_2' &= \{(l_1, in; l_2, out)\}.
\end{aligned}$$

The system Π *generates the set of natural numbers* $\{2^n \mid n \geq 0\}$, *or equivalently the Parikh image of the language* $\{a^{2^n} \mid n \geq 0\}$. *At the beginning of the computation only the symbol-object* A *is present in region* 2. *Moreover, the rule* $A \to AA$ *with label* l_2 *is present in region* 2 *and the rule* $A \to a$ *with label* l_1 *is present in region* 1. *The rule* $A \to AA$ *is applied an arbitrary number of times in region* 2. *At some point of time we move the rule* $A \to a$ *from region* 1 *to region* 2 *and the rule* $A \to AA$ *from region* 2 *to region* 1 *using the antiport* $(l_1, in; l_2, out)$ *associated with membrane* 2. *Once the rule* $A \to a$ *is inside region* 2 *all copies of* A *are transformed into* a*'s and no rule can be applied anymore in any region. Then the computation halts and we obtain the result in region* 2.

Example 2.2 Using Symport of Rules

Consider the following CR P system of degree 2:

$$\Pi = (O, R, l, \mu, w_1, w_2, R_1, R_2, R_1', R_2', 2),$$

where:

$$\begin{aligned}
O &= \{A, a, C\};\\
R &= \{A \to a, A \to AA, C \to C\};\\
&\quad l \text{ is an injective labeling of the rules in } R;\\
&\quad \text{let } l_1 = l(A \to a), l_2 = l(A \to AA),\ l_3 = (C \to C)\\
&\quad \text{and } L = \{l_1, l_2, l_3\};\\
\mu &= [_1\, [_2\,]_2\,]_1;\\
w_1 &= \lambda,\ w_2 = A;\\
R_1 &= \{A \to a\};\\
R_2 &= \{A \to AA, C \to C\};\\
R_1' &= \emptyset;\\
R_2' &= \{(l_3 l_2, out), (l_3 l_1, in)\}.
\end{aligned}$$

At the beginning of the computation only one copy of the symbol-object A is present in region 2. The rules $A \to AA$ and $C \to C$ are, also, present in region 2. In this region, the rule $A \to AA$ is applied an arbitrary number of times (notice that the rule $C \to C$ cannot be applied). At some point of time using the symport $(l_3 l_2, out) \in R_2'$ the rules $A \to AA$ and $C \to C$ move from region 2 to region 1. After this step all evolution rules (namely $A \to AA, C \to C$, and $A \to a$) are in region 1. Since the symport rules are applied in a maximally parallel manner, the symport $(l_3 l_1, in) \in R_2'$ is applied and the rules $A \to a$ and $C \to C$ move from region 1 to region 2. Note that, because of the dummy rule $C \to C$, this symport rule can be applied only after the rule $A \to AA$ exits from region 2. When the rule $A \to a$ enters region 2 all symbol-objects A are changed to a and the computation halts. The result (i.e., the Parikh image of the language $\{a^{2^n} \mid n \geq 0\}$) is then obtained in region 2.

2.2.2 Using Non-Cooperative Evolution Rules

We present now several results proved for CR P systems using evolution rules that are non-cooperative; the results are obtained simulating parallel grammar devices (for the basic notions on this topic the reader can consult the short survey given in Section 1.1.4).

If we use antiport rules with unbounded weight, then the direct simulation of ET0L systems is rather simple, as shown in the following theorem.

Theorem 2.3 $PsET0L \subseteq PsCRP_2(0, *, ncoo)$.

Proof. Given an extended tabled Lindenmayer system $G = (V, T, w, h)$ with 2 tables ($h = \{h_1, h_2\}$) in the normal form described in Theorem 1.15, we construct a CR P system Π generating the Parikh set of L (actually, we remove the trivial productions from h_1 and h_2). Let us denote $N = V - T$.

Consider

$$\Pi = (O, R, l, \mu, w_1, w_2, R_1, R_2, R_1', R_2', 2),$$

where:

$$\begin{aligned}
O &= V \cup \{\#\};\\
R &= h_1 \cup h_2 \cup \{\# \to \#\} \cup R'';\\
&\quad \text{where } R'' = \{Y \to \# \mid Y \in N\};\\
&\quad \text{l is an injective labeling of the rules in } R;\\
&\quad \text{Let}\\
&\quad L_1 = \{l(r) \mid r \in h_1\};\\
&\quad L_2 = \{l(r) \mid r \in h_2\}, \text{ and } L_3 = \{l(r) \mid r \in R''\};\\
\mu &= [_1\, [_2\,]_2\,]_1;\\
w_1 &= \lambda, w_2 = w;\\
R_1 &= h_1 \cup R'';\\
R_2 &= h_2 \cup \{\# \to \#\};\\
R_1' &= \emptyset;\\
R_2' &= S_1 \cup S_2,\\
&\quad \text{where}\\
S_1 &= \{(x, in; y, out) \mid x = \langle L_1 \rangle \text{ and } y = \langle L_2 \rangle\}\\
&\cup \{(x, in; y, out) \mid x = \langle L_2 \rangle \text{ and } y = \langle L_1 \rangle\},\\
S_2 &= \{(y, in; x, out) \mid y = \langle L_3 \rangle \text{ and } x = \langle L_1 \rangle\}\\
&\cup \{(y, in; x, out) \mid y = \langle L_3 \rangle \text{ and } x = \langle L_2 \rangle\}.
\end{aligned}$$

The system works as follows. The rules of the two tables h_1 and h_2 are moved between regions 1 and 2 using the antiports in S_1. This simulates the application of the productions of one of the two table over the symbol-objects contained in region 2 (in the beginning the symbol-objects corresponding to the axiom w of G are present in region 2). These antiports guarantee that it is not possible to apply rules of the first table mixed with rules of the second table.

The role of the antiports in S_2 is to stop the computation (in particular, the movements of evolution rules across membrane 2). The purpose of these antiports is twofold: the rules of R'' are moved into region 2 and in the opposite direction the rules of the table h_1 (or of the table h_2) are moved into region 1. If there are still symbol-objects corresponding to non-terminals of

G in region 2, then one of the rules in R'' will produce the trash symbol # and then the computation will never halt, because of the presence of the rule $\# \to \#$ in region 2. Therefore, the computation halts iff all symbol-objects corresponding to terminals of G are obtained in region 2 and then the system Π generates exactly the Parikh set of $L(G)$. □

The previous result was obtained using antiport of rules of unbounded weight; in particular, the proof of Theorem 2.3 indicates that the weight of the antiport rules used is the maximum of the cardinality of the two tables of the ET0L system simulated (and this number is unbounded).

We present now a way to decrease (and to bound) the weight of the antiport rules by using a priority among the transport rules of the system.

The priority used here is just like the classical *weak priority* defined in the P systems area. The difference is that here the priority is defined among transport rules in charge of moving evolution rules.

Given two sets of transport rules R_1 and R_2, we indicate that R_1 has weak priority over R_2 by writing $R_1 > R_2$. This means that in the process of assigning transport rules to "objects" (which in our system are evolution rules) first the transport rules in R_1 are assigned in a non-deterministic, maximally parallel manner, and then the rules in R_2 are assigned to the remaining objects (in our case evolution rules) again in a non-deterministic, maximally parallel manner.

In order to distinguish between this priority and classical priority among evolution rules we use the notation pri_{tran}.

The use of the weak priority among transport rules makes possible to simulate an ET0L system using antiports of weight 2 and 5 membranes.

Theorem 2.4 $PsET0L \subseteq PsCRP_5(0, 2, ncoo, pri_{tran})$.

Proof. Given an extended tabled Lindenmayer system $G = (V, T, w, h)$ with 2 tables ($h = \{h_1, h_2\}$), in the normal form described in Theorem 1.15, generating L, we construct a CR P system Π generating the Parikh image of L. Let $N = V - T$.

After removing the trivial productions from h_1 and h_2, construct another table h_3 composed by the rules $\{Y \to \# \mid Y \in N\}$, where # is a new symbol not in V. Let table h_1 have m rules, table h_2 have k rules, and table h_3 have n rules. Let $max\{m, k, n\} = p$. Then, if one of the three tables has less than p rules, we add $D \to D$, for $D \notin V$, as many times as needed to increase the number of rules in that table to p. Then we can assume that the cardinality of the three tables so adjusted is exactly p. Moreover, we need different rules in each one of the three tables (more precisely, we need rules with different labels). Therefore, we substitute every rule $X \to x$ in table h_1 with the rule $X \to xd_1$, where d_1 is a new symbol not in V. Similarly, we substitute every rule $X \to x$ in table h_2 with the rule $X \to xd_2$ where

d_2 is a new symbol not in V. In what follows, the tables h_1 and h_2 changed in the above described way are called h'_1 and h'_2. For the table h_3, we do not change the rules because they are already different from the ones in the other two tables.

Consider the system

$$\Pi = (O, R, l, \mu, w_1, w_2, w_3, w_4, w_5, R_1, R_2, R_3, R_4, R_5, R'_1, R'_2, R'_3, R'_4, R'_5, 3),$$

where:

$$\begin{aligned}
O &= V \cup \{\#, D\} \cup \{d_1, d_2\};\\
R &= h'_1 \cup h'_2 \cup h_3 \cup \{\# \to \#\};\\
&\quad \text{l is an injective labeling of the rules in } R;\\
&\quad \text{let } l_1 = l(\# \to \#);\ \text{ let } l'_i, l''_i, l'''_i,\ i \in \{1, \cdots, p\},\\
&\quad \text{be the labels associated with the} i - th \text{ rule in the tables}\\
&\quad h'_1, h'_2 \text{ and } h_3, \text{ respectively}\\
&\quad \text{(the cardinality of the three tables is } p\text{);}\\
&\quad \text{let } L_1 = \{l(r) \mid r \in h'_1\} = \{l'_i \mid i \in \{1, \cdots, p\}\};\\
&\quad L_2 = \{l''_i \mid i \in \{1, \cdots, p\}\},\ L_3 = \{l'''_i \mid i \in \{1, \cdots, p\}\};\\
\mu &= [_1\, [_2\, [_3\, [_4\, [_5\,]_5\,]_4\,]_3\,]_2\,]_1;\\
w_1 &= \lambda,\ w_2 = \#,\ w_3 = w,\ w_4 = \#,\ w_5 = \lambda;\\
R_1 &= \{\# \to \#\};\\
R_2 &= h'_2;\\
R_3 &= h'_1 \cup \{d_1 \to \lambda, d_2 \to \lambda, \# \to \#\};\\
R_4 &= h_3;\\
R_5 &= \{\# \to \#\};\\
R'_1 &= \emptyset;\\
R'_2 &= \{(l_1, in; ij, out) \mid i \in L_1, j \in L_2\};\\
R'_3 &= \{(l'_i, in; l''_i, out) \mid i \in \{1, \cdots, p\}\}\\
&\cup\ \{(l''_i, in; l'_i, out) \mid i \in \{1, \cdots, p\}\};\\
R'_4 &= \{(l'_i, in; l'''_i, out) \mid i \in \{1, \cdots, p\}\};\\
R'_5 &= \{(ij, in; l_1, out) \mid i \in L_1, j \in L_3\};\\
&\quad R'_5 > R'_4; R'_2 > R'_3.
\end{aligned}$$

The system Π works as follows. Initially, the symbol-objects corresponding to the axiom w of G, along with the rules of table h'_1 are present in the output region 3. These rules simulate the application of a rule of h_1 over the symbol-objects present in region 3 (the dummy symbols d_1 produced by such rules are immediately deleted by the rule $d_1 \to \lambda$).

At the beginning the rules of h'_2 are in region 2, while the rules of h_3 are in region 4.

In order to pass from table 1 to table 2 (and viceversa) we use the antiports in R'_3. These antiports exchange each rule of one table with a rule in the other table. We need to guarantee a full"swap" of the rules (i.e. that all rules from one table are exchanged with all rules of the other table). Ensuring that the passage from one table to the other is "complete" is essential, since we need to avoid having rules of table h'_1 mixed with rules of table h'_2 in region 3. That is checked by the antiport rules in R'_2. In fact, if the passage from one table to the other one is not correct then rules from both tables will be present in region 2 simultaneously. In that case, one of the antiports in R'_2 is applied (it has higher priority over the antiports in R'_3) and then the rule $\# \rightarrow \#$ is imported in region 2 (because the symbol $\#$ is present in region 2 this leads to a non-halting computation). Thus, such a construction ensures that the passage from one table to the other is made in a correct way.

The presence of rules of table h'_2 in region 3 simulates the application of a rule from h_2 over the symbol-objects present in that region (the dummy symbols d_2's produced by such rules are immediately deleted by the rule $d_2 \rightarrow \lambda$). The change from table h'_2 to table h'_1 is made similarly to the above described way for the passage from table h'_1 to h'_2.

Finally, we need to guarantee that the computation halts only when all symbol-objects in region 3 have been transformed into terminals of G. To halt the computation we have to stop the movements of rules made possible by the antiports in R'_3. This can be made using the antiports in R'_4 that move the rules of table h'_1 into region 4 and the rules of table h_3 into region 3. These antiports exchange each rule of one table with the corresponding rule in the other table. After applying such antiports, the antiports in R'_3 cannot be applied anymore (the rules of h'_1 and h'_2 are in region 4 and 2 respectively). The complete passage from table h'_1 to table h_3 is guaranteed by antiports in R'_5 that work similarly to the ones described above for antiports in R'_2 (notice that $R'_5 > R'_4$). In particular, if it happens that rules from h'_1 and from h_3 are simultaneously present in region 4, then the rule $\# \rightarrow \#$ is moved to region 4 using one of the antiports in R'_5 and this leads to a non-halting computation. Therefore, the passage from table h'_1 to h_3 must be made in a "complete" way, as well.

On the other hand, the rules of h_3 (that are of the kind $Y \rightarrow \#, Y \in N$) check that there are only terminal symbol-objects in region 3: if this is not the case, then the symbol $\#$ is produced in region 3 and using the rule $\# \rightarrow \#$ present in that region a non-halting computation is obtained.

Therefore, the computation halts iff all symbol-objects corresponding to terminals of G are obtained in region 3. Thus, the system Π generates (in region 3) exactly the Parikh set of $L(G)$. □

If we use antiports with bounded weight and no priorities among the transport rules, then CR P systems can generate at least the family of

Parikh images of the languages generated by Indian parallel grammars.

Theorem 2.5 $PsIP \subseteq PsCRP_2(0,2,ncoo)$.

Proof. Given an Indian parallel grammar $G = (N, T, S, P)$, we construct a new Indian parallel grammar $G' = (N' = N \cup \{S'\}, T, S', P' = P \cup \{S' \to S\})$. It is obvious that $L(G') = L(G)$. In addition, we assume that P does not contain trivial rules of the kind $X \to X, X \in N$.

We construct the CR P system

$$\Pi = (O, R, l, \mu, w_1, w_2, R_1, R_2, R_1', R_2', 2),$$

where:

$$\begin{aligned}
O &= N' \cup T \cup \{Z, D\} \text{ with } Z, D \notin N' \cup T;\\
R &= P' \cup \{D \to D, Z \to Z\} \cup \{X \to X \mid X \in N'\};\\
&\quad \text{l is an injective labeling of the rules in } R;\\
&\quad \text{let } l_1 = l(D \to D) \text{ and } l_2 = l(Z \to Z);\\
\mu &= [_1\, [_2\,]_2\,]_1;\\
w_1 &= \lambda,\ w_2 = S';\\
R_1 &= \{S' \to S', Z \to Z\} \cup h;\\
R_2 &= \{X \to X \mid X \in N'\} \cup \{S' \to S, D \to D\};\\
R_1' &= \emptyset;\\
R_2' &= R_2'' \cup R_2''' \cup R_2^{iv}, \text{ where}\\
&\quad R_2'' = \{(x, in; y, out) \mid x = l_1 l(X \to X), y = l(X \to w)\\
&\quad \text{with } X \to w \in P'\},\\
&\quad R_2''' = \{(x, in; y, out) \mid x = l(X \to w), y = l_1 l(X \to X)\\
&\quad \text{with } X \to w \in P'\},\\
&\quad R_2^{iv} = \{(x, in; y, out) \mid x = l_2 l(X \to X), y = l(X \to w)\\
&\quad \text{with } X \to w \in P'\}.
\end{aligned}$$

The system Π works in the following way. The symbol-objects that correspond to a sentential form generated by the grammar G' are present in region 2. At the beginning, only the symbol-object S' is present in region 2. The basic idea is to simulate a derivation of the grammar G' in region 2. The antiports in R_2' are used to bring in region 2, one at a time, the rules of the grammar G'. In particular, the antiports in R_2''' are used to bring inside region 2 a new rule of G' to be applied, while the antiports in R_2'' are used to bring back the rule of G' that has been used in region 2. The antiports in R_2^{iv} are used to stop the computation. The rules $X \to X, X \in N'$, are used to check that no symbol-objects corresponding to non-terminals of G' are present in region 2 at the time the computation halts.

In particular, when one of the antiports in R_2''' is applied, then a rule $X \to w$ moves from region 1 to region 2 and, at the same time, the rules $D \to D$ and $X \to X$ move in the opposite direction. A rule from P' can move from region 1 to region 2 only by using one of the antiports in R_2''' and then only when the dummy rule $D \to D$ is present in region 2. The role of the dummy rule $D \to D$ is to guarantee that only one rule of P' is present in region 2 at any particular moment of time. The rule $X \to X$ is, also, moved from region 2 to 1 to avoid a "mixed" application of the rules $X \to w$ and $X \to X$ in region 2.

After a rule $X \to w$ enters region 2, it can be applied or it can exit without being applied. Without loss of generality we can assume that the rule is applied and in the next step it exits or is being re-applied. After it has been applied a certain number of times (0, 1 or more), then the rule must exit and a new rule from P' must enter region 2.

To achieve this, the antiports in R_2'' must be used. Using one of the antiports in R_2'' the rule $X \to w$ can move from region 2 to 1. At the same time, in the opposite direction, the dummy rule $D \to D$ and the rule $X \to X$ move back from region 1 to 2. At this point a new rule from h' can be moved inside region 2 using one of the antiports in R_2''' as described above. As mentioned earlier, only one rule of G' can be present in region 2 at each moment, because of the dummy rule $D \to D$ in the antiports of R_2'''. In fact, only one copy of the rule $D \to D$ is present in the system and such copy moves between regions 2 and 1.

To halt the computation the movement of rules between regions 2 and 1 should be stopped. This can be achieved by applying one of the antiports in R_2^{iv}. The last rule $X \to w$ from P' used in region 2 is moved out and, at the same time, the dummy rule $Z \to Z$ and the rule $X \to X$ are moved to region 2. Notice that the dummy rule $D \to D$ is not moved back in this case. After applying such an antiport, the movement of rules between regions 2 and 1 stops. At that time, the dummy rule $D \to D$ is in region 1 and there are no rules from P' in region 2. Thus, no antiports from R_2''' can be applied anymore.

On the other hand, the presence of the rules $\{X \to X \mid X \in N'\}$ guarantees that no symbol-objects corresponding to non-terminals of G' are present in region 2 when the computation halts.

Therefore, the system Π generates exactly the Parikh set of $L(G')$. □

2.2.3 Using One Catalyst: Universality

If we use catalysts, then we can inhibit the parallelism in the application of the evolution rules and we can simulate sequential grammars. In this case CR P systems become universal. In this section we present an universality result where CR P systems simulate matrix grammars with appearance

checking using one catalyst, antiports of weight 2, and 3 membranes.

Because of the way CR P systems work and in particular since it is not possible to move objects, inhibiting the parallelism without using catalysts seems a very hard task.

Theorem 2.6 $PsCRP_3(0,2,cat_1) = PsRE$.

Proof. The idea is to simulate matrix grammars with appearance checking in the Z-binary normal form. We start with a matrix grammar with ac in Z-binary normal form $G = (N,T,S,M,F)$ in the standard notation with $N_1 = \{X_1, X_2, \cdots, X_{m'}\}$ and $N_2 = \{A_1, A_2, \cdots, A_{m''}\}$. We construct the CR P system

$$\Pi = (O, R, l, \mu, w_1, w_2, w_3, R_1, R_2, R_3, R_1', R_2', R_3', 2),$$

where:

$$\begin{aligned}
O &= N \cup T \cup \{d, g, g', g'', g''', g^{iv}, \#\} \cup \{i \mid 1 \leq i \leq n\};\\
R &= S_1 \cup S_2 \cup S_3 \cup S_4 \cup S_5\\
&\cup \{i \to \lambda \mid 1 \leq i \leq n\} \cup \{\# \to \#\}, \text{ where}\\
&\quad S_1 = \{X \to iYd, cA \to icxd \mid\\
&\quad m_i = (X \to Y, cA \to cxd) \in M, 1 \leq i \leq k\};\\
&\quad S_2 = \{X \to iYd, A \to i\# \mid\\
&\quad m_i : (X \to Y, A \to \#) \in M, k+1 \leq i \leq n\};\\
&\quad S_3 = \{Z \to d \mid m_{n+1} : (Z \to \lambda)\}\\
&\quad \cup \{g'' \to g'', g''' \to g''', g^{iv} \to g^{iv}\};\\
&\quad S_4 = \{g \to g, d \to \lambda\};\\
&\quad S_5 = \{g' \to g', d \to \#\};\\
&\quad l \text{ is an injective labeling of the rules in } R;\\
&\quad \text{let } r_{i,1} = l(X \to iYd) \text{ for each } X \to iYd \in S_1 \cup S_2,\\
&\quad r_{i,2} = l(cA \to icxd) \text{ for each } cA \to icxd \in S_1,\\
&\quad r_{i,2} = l(A \to i\#) \text{ for each } A \to i\# \in S_2,\\
&\quad r_f = l(Z \to d),\\
&\quad r_0 = l(g \to g), r_0' = l(g' \to g'), r_0'' = l(g'' \to g''),\\
&\quad r_0''' = l(g''' \to g'''), r_0^{iv} = l(g^{iv} \to g^{iv});\\
\mu &= [_1\, [_2\, [_3\,]_3\,]_2\,]_1;\\
w_1 &= \lambda;\\
w_2 &= cX_{init}A_{init};\\
w_3 &= cZX_1X_2 \cdots X_{m'}A_1A_2 \cdots A_{m''};\\
R_1 &= S_1 \cup S_2 \cup S_3;
\end{aligned}$$

$$
\begin{aligned}
R_2 &= S_4 \cup \{i \to \lambda \mid 1 \leq i \leq m\} \cup \{\# \to \#\};\\
R_3 &= S_5 \cup \{\# \to \#\};\\
R'_1 &= \emptyset;\\
R'_2 &= S'_1 \cup S'_2\\
&\cup \{(r_f r''_0, in; r_0, out), (r'''_0, in; r''_0, out), (r^{iv}_0, in; r'''_0 r_f, out)\}, \text{ where}\\
&\quad S'_1 = \{(r_{i,1} r_{i,2}, in; r_0, out) \mid 1 \leq i \leq n\},\\
&\quad S'_2 = \{(r_0, in; r_{i,1} r_{i,2}, out) \mid 1 \leq i \leq n\};\\
R'_3 &= S''_1 \cup S''_2 \cup \{(r_f, in; r'_0, out)\}, \text{ where}\\
&\quad S''_1 = \{(r_{i,j}, in; r'_0, out) \mid 1 \leq i \leq k, j \in \{1,2\}\},\\
&\quad S''_2 = \{(r_{i,1}, in; r'_0, out) \mid k+1 \leq i \leq n\}.
\end{aligned}
$$

The system Π works in the following way. Initially, the evolution rules corresponding to the matrices of the matrix grammar are present in region 1. In particular, S_1 is the set of rules corresponding to the matrices of type 2, S_2 is the set of rules of type 3. Moreover, the final rule $Z \to d$ is, also, present. Each one of these rules has been modified to generate a special symbol d that will be used to check if a rule has been applied. The purpose of the symbol i produced by a rule when applied (except by the final rule) is used to distinguish between the rules (otherwise, two identical rules in two different matrices get the same label). Such symbol i is immediately deleted when generated.

The sentential form is stored in region 2, the output region of the system. At the beginning of the computation only the objects X_{init}, A_{init} and the catalyst c are present in region 2. The idea is that, step after step, the productions of the matrix grammar are applied over the symbol-objects present in region 2. If something goes wrong during the application of the rules of the matrix grammar, the symbol # is generated in region 3 or region 2 and the computation will never halt because of the presence of the rule $\# \to \#$.

We now show in greater detail how the computation proceeds. Using the antiports in S'_1 two rules of the same matrix i, with labels $r_{i,1}, r_{i,2}$ are moved from region 1 into region 2. Once the rules with labels $r_{i,1}, r_{i,2}$ are in region 2, they can be applied or they can exit together moving from region 2 to 1 using one of the antiports in S'_2 (and in this case a new pair of rules can be introduced in region 2). If both rules are applied then, in the next step, they can be reapplied or they can exit together, as in the previous case. Because of the maximality of the parallelism and because the two rules can only exit together, if they can be applied, both are applied or both come back to region 1 without being applied. The "dummy" rule with label $r_0 : g \to g$ is used only to ensure that only one pair of rules moves inside region 2 (i.e., when r_0 is in region 1 then no other rules can enter region 2).

Now suppose that only one of the two rules can be applied and let us

assume that the rules are of a matrix of type 2. Without loss of generality suppose that the first rule $r_{i,1}$ can be applied and it is applied in region 2 at step j. In the same step j, the other rule $r_{i,2}$ cannot be applied because the corresponding non-terminal object is not present. In this case, because $r_{i,2}$ cannot come back "alone" to region 1 and because of the maximality of the parallelism, such rule moves from region 2 to region 3, using one of the antiports of S_1''. Once the rule $r_{i,2}$ is in region 3, the computation will never halt, because the symbol # will be generated (in region 3 the moved rule is applied producing d; therefore in the next step $d \rightarrow \#$ is executed). In this way we make sure that both rules of the same matrix of type 2 have to be applied to get a successful computation.

Suppose now that the two rules $r_{i,1}$ and $r_{i,2}$ that are inside region 2 correspond to a matrix of type 3 (with ac); the label $r_{i,2}$ corresponds to the rule used in ac mode.

If the rule $r_{i,2}$ is applied, then # is generated in region 2 and the computation never halts because of the rule $\# \rightarrow \#$ (independently from the fact that rule $r_{i,1}$ is applied).

If the rule $r_{i,1}$ is applied at step j but not the other one $r_{i,2}$, then this last rule simply "waits" that the first rule is applied. In fact, such rule cannot be moved neither to region 1 where the two rules can move only together, nor to region 3, because there are no antiports in S_2'' that can move rules used in ac mode. After the rule $r_{i,1}$ is applied, the two rules can move together to region 1, using antiports in S_2', or they can remain in region 2 again. In this way the ac mode is respected.

In order to halt the computation (i.e., to terminate the symport/antiport movements of rules) it is necessary to bring the rules with labels r_f and r_0'' inside region 2 using the antiport $(r_f r_0'', in; r_0, out)$ in R_2' at some step j. If Z is not present in region 2, then in step $j+1$ the antiport $(r_f, in; r_0', out)$ in R_3' is applied and this leads to a non halting computation because the symbol # is generated in region 3 (rule $Z \rightarrow d$ and then $d \rightarrow \#$ are applied). In case that Z is present in region 2, the rule r_f is applied in the step $j+1$ and in the same step the antiport $(r_0''', in; r_0'', out)$ in R_2'' is applied (then r_0''' comes inside region 2). Therefore, in the step $j+2$ the rule r_f can move from region 2 to region 1, together with the rule r_0''' using the antiport $(r_0^{iv}, in; r_0''' r_f, out)$ in R_2'. Then the computation halts (no evolution rules and no symport/antiport rule can be applied in any region and any membrane, respectively). In this way, the system Π generates exactly the Parikh set of $L(G)$. □

2.3 Final Remarks

The evolution-communication model has been originally proposed and studied in [46]. Another approach to avoid the use of target indications in the

evolution rules and having communication mixed with evolution has been studied in [31]; however, while in [31] one changes the structure of the evolution rules, in the evolution-communication model one just puts together two already known types of P systems. Another model of P systems with both evolution and communication based on symport and antiport rules has been introduced in [29]. This model is inspired by the Na^+K^- pump, a typical phenomenon of active transport that happens in living cells.

The results (and the proofs) of Section 2.1.1 have been recalled from [46].

While here universality (in terms of set of numbers) of EC P systems has been proved by using symport/antiport rules of weight one, non-cooperative evolution rules, and three membranes, this result has been improved in [6], where universality has been proved by using only two membranes and has been generalized to set of vectors of numbers. The proof is still based on the simulation of programmed grammars with appearance checking. A similar result has been obtained in [96], where universality has been shown by using non-cooperative evolution rules, two membranes, and symport rules of weight at most two. In [6] accepting EC P systems have been introduced and shown to be universal by using non-cooperative evolution rules, two membranes, symport/antiport rules of weight one, and also using promoters. In [7] universality has been proved for (deterministic) accepting EC P systems by either using three membranes, non-cooperative evolution rules, and symport/antiport rules of weight one, or by using three membranes, non-cooperative evolution rules, three membranes, and symport rules of weight at most two. Another variant of EC P systems proved to be universal is called proton pumping P system and has been introduced in [9]; we discuss this model in details in Chapter 4.

More general results concerning time-free EC P systems (where the result of the computation is independent from time of execution of the rules) have been obtained in [11]; time-free P systems are presented in details in Section 4.2.

On the other hand, the evolution-communication model has been also used in [18] to model biological processes; this topic is discussed in the following chapter.

The idea of moving rules instead of object has been presented in [118]. There, the author suggested a model of P systems where the rules are moved across the membranes rather than the objects processed by these rules. In the model presented in [118] the migration of the simple evolution rules is governed by metarules existing in all regions. There, it has been shown that this class of P systems, using catalysts, can strictly generates the family of Parikh images of languages generated by matrix grammars without appearance checking. Actually, P systems with moving rules were already investigated by R. Freund and his co-workers, in a more general setup, where

both the objects and the rule can move (universality results were obtained – references can be found in [117]).

CR P systems have been introduced in [50]. The definition, the results, and the examples given in Section 2.2 have been recalled from this paper.

A possible variant of CR P systems has been introduced in [78] and it is obtained by dividing the computation into two substeps that are applied in an interleaved way: first all possible communication rules are applied, and then, the evolution rules are applied, in a non-deterministic, maximally parallel manner. In this case, using non-cooperative evolution rules, the systems generate exactly the family of Parikh images of ET0L languages. Moreover, in the same paper, universality has been obtained by using only one catalyst.

Chapter 3

Using EC P Systems to Model Biological Processes

We have presented two basic models of P systems (with symbol-objects and with symport/antiport rules) and the evolution-communication model; until now, we have discussed only the mathematical aspects of such systems while the biological aspects and biological motivations have not been deeply investigated.

However, as we have already discussed in the Introduction to this Thesis, the interaction with biology is fundamental and one of the goals of membrane computing can be considered the mathematical modeling of biological processes. For this reason we dedicate this chapter to show how the membrane computing framework can be used to help biology, for instance, by modeling some very common biological processes.

In particular, we present a P system simulator that implements the evolution-communication model, enriched with some probabilistic parameters inspired by cell biology.

At the beginning we discuss the biological motivations of the implemented model and we present the functioning of the software-simulator realized.

Moreover we discuss the choices made during the implementation of the software: we think that this is a good way to establish a link between the mathematical framework and the biological reality, introducing new concepts in the P system area and, at the same time, comparing the standard concepts of membrane computing with what we have in cell biology.

We conclude the chapter showing how it is possible to use the mathematical model (and the associated software) to simulate some cellular processes and how the simulator can be used to infer useful results for biologists; in particular, we simulate the process of respiration in *Escherichia coli*, the corresponding proton pumping by cytochrome *c* oxydase in *Anacystis nidulans*, the interplay between oxygen consumption and oxygen production by

photosynthesis *II* (*PSII*) in *Synechocystis PCC*6803.

3.1 The Software

The software realized is able to simulate the functioning of an EC P system. Moreover, the simulator solves the conflicts that can appear when the rules choose the symbol-objects, using the *weak priority* approach; the weak priority can be seen as a competition of the rules for each single occurrence of the objects.

The simulator takes as an input the rules of the P system to simulate (evolution rules and symport/antiport rules), the structure of the model (actually, the structure is not limited to be a tree, as usual defined in the P systems area, but it is permitted to be a graph), and the occurrences of the symbol-objects present at the beginning of the computation in the regions of the P system.

Also, we must specify for each rule r two kinds of probabilities: *the probability to be available* (we call it Pav_r) and the *probability to win a conflict* (we call it $Pwin_r$); actually, as we will see below, the simulator computes this second probability using an integer coefficient, $Cwin_r$, fixed for each rule r.

The simulation takes place in the following way.

At each step, the simulator decides which are the available rules in that step, and this decision is taken by using the probability Pav_r fixed for each rule r. These probabilities are independent of each others and they express the probability for each rule to be *available* for the application in a step.

After this choice, the simulator solves the conflicts present among the *available* rules, using the probabilities $Pwin_r$ that indicates the probability for the rule r to win a conflict over a symbol-object with other rules; in this case the probability $Pwin_r$ is calculated for each conflict and depends on the coefficient $Cwin_r$ associated with r and on the coefficients of the other rules involved in the same conflict: in this way, for each conflict, a probability distribution among the rules involved in that conflict is produced. If $r_i, i = 1, \cdots, k$, are the rules involved in a conflict over a symbol-object, then $Pwin$ is calculated, for each $r_i, i = 1, \cdots, k$, as follows:

$$Pwin_{r_i} = Cwin_{r_i}/(Cwin_{r_1} + \cdots + Cwin_{r_k}), \text{ for } i = 1, \cdots, k.$$

Finally, when every conflict has been solved, the rules are applied in parallel in each region, as usual defined in the P systems area.

3.1.1 How the Software Works: A Simple Example

The best way to explain the working of the software is to discuss a simple example.

Suppose we have a simple P system composed of one membrane (labeled with 1) and in this region there are the evolution rules $r_1 : a \rightarrow aa$, $r_2 : b \rightarrow a$, and $r_3 : aa \rightarrow a$;

The first rule r_1 says that one occurrence of the symbol-object a is replaced by two occurrences of the same symbol-object, r_3 is exactly the opposite, while the second rule means that one occurrence of the symbol-object b is transformed in one occurrence of the symbol-object a. In region 1 there are (at the beginning of the computation) 10 occurrences of the symbol-object a and 1 occurrence of the symbol-object b.

Moreover, we suppose that $Pav_{r_1} = 0.8, Pav_{r_2} = 0.2, Pav_{r_3} = 0.6$, and, $Cwin_{r_1} = Cwin_{r_3} = 100$ and $Cwin_{r_2} = 40$.

At each step, the simulator chooses the available rules in region 1 and the choice is made using the probabilities $Papp_{r_i}, i = 1, 2, 3$.

This probability implies that, in some steps, the available rules might be different from the previous steps and there might be rules that are chosen more often than the others (their Pav is bigger than the probability of the other rules). After constructing the list of the available rules, the simulator must solve the conflicts that, at this point, might be present in the region.

Suppose that, at some step, the list of available rules is composed of $r_1 : a \rightarrow aa$ and $r_3 : aa \rightarrow a$.

At this point, the simulator searches for possible conflicts on the occurrences of the symbol-objects present in region 1.

In our case, there are conflicts because both the rules can use the symbol-object a. Then, the simulator starts to assign each occurrence of a's present in region 1 to one of the two rules involved in the conflict.

For this goal one uses the probability $Pwin_{r_i}$ calculated for each rule from the coefficient $Cwin_{r_i}$ associated with the rule r_i; actually, in our case, the probabilities are computed in the following way:

$$Pwin_{r_i} = Cwin_{r_i}/(Cwin_{r_1} + Cwin_{r_3}), i \in \{1, 3\}.$$

In this example, the probability is the same (0.5) for both the rules involved in the conflict. Using such probabilities, each occurrence of object a is assigned. When a rule *wins* the other, it takes the number of occurrences of objects that it needs (2 for r_3 and 1 for r_1). In this way, when there are no more conflicts, the rules can be applied in a parallel way. For example, in our case, suppose that r_3 wins the first conflict against r_1, then r_3 can take the two occurrences of a that needs; in this way, the $free$ occurrences to assign are 8. Now, there are still conflicts to solve so the process continues in the same way.

Suppose that, at the end, 6 occurrences of a are assigned to r_1 and 4 to r_3; after applying the rules in the parallel way, the new number of occurrences of a's in region 1 is 14.

To clarify the idea of the working of the simulator, we discuss the previous example step by step starting from the initial configuration. The P system is (when modeling the evolution of a biological process the output region is not necessary, hence we do not specify it):

$$\Pi = (O, \mu, w_1, R_1),$$

with

$$\begin{aligned} O &= \{a, b\}, \\ \mu &= [_1\,]_1, \\ w_1 &= a^{10}b, \\ R_1 &= \{r_1 : a \to aa, \quad r_2 : b \to a, \quad r_3 : aa \to a\}. \end{aligned}$$

Moreover, we have that

$$\begin{aligned} &Pav_{r_1} = 0.8, \; Pav_{r_2} = 0.2, \; Pav_{r_3} = 0.6, \\ &Cwin_{r_1} = Cwin_{r_3} = 100, \; Cwin_{r_2} = 40. \end{aligned}$$

1: *List of rules available*

The simulator decides which are the available rules *in this step*. Suppose that, using the probabilities Pav, the simulator chooses the rules r_1 and r_3. Then, the list of available rules in this step of computation is $L = \{r_1, r_3\}$.

2: *Searching for conflicts in region 1*

The number of occurrences of a's is 10. There are conflicts among the rules in the list L (in this case, both the rules can use the symbol-object a).

3: *Solving the conflicts*

The simulator solves the conflicts by assigning, using the probabilities $Pwin$, the occurrences of a's to the rules involved in the conflicts.

In this case, $Cwin_{r_1} = Cwin_{r_3} = 100$ and hence $Pwin_{r_1} = Pwin_{r_3} = 100/200 = 0.5$; therefore, in this case the two rules have the same probability to win a conflict in which they are involved.

Suppose that, for example, the rule r_3 wins the first conflict; therefore, 2 free occurrences of a's are assigned to this rule. Now, there are further 8 *free* occurrences of object a to assign. There are still conflicts so the simulator repeats, in similar way, step 3 until all the conflicts are solved.

4: *All the conflicts in region 1 have been solved*

Suppose that all the conflicts in region 3 have been solved and suppose, for example, that 6 occurrences of a's are assigned to r_1 and 4 to r_3.

5: *Execute the rules*

When the objects have been assigned, then the simulator executes the rules present in the membrane, in the parallel way, according to the assigned objects. In particular, after the execution of the rules, we will have 14 occurrences of object a and 1 occurrence of the object b.

6: *Repeat the same process* for a new step of computation (it starts again from 1, where a new list L of available rules is created).

In this example, we had only one membrane but if the P system to simulate has many membranes, then the same algorithm is applied in each region of the system, and only when each occurrence of each symbol-object is assigned (or, at least, the simulator has tried to assign it), the rules are executed in parallel in each region, as usual in P systems.

3.2 A First Probabilistic Simulation: Is Life Unpredictable?

In this section, we show the results of a first experiment, consisting in the simulation of a very simple EC P system.

This simple experiment clarify the working of the simulator and it is interesting because of the *unexpected* results and because it can remind many others systems (in different fields) where similar phenomena appears.

The P system is defined in the following way:

$$\Pi = (O, \mu, w_1, R_1),$$

with

$$\begin{aligned} O &= \{a\}, \\ \mu &= [_1\,]_1, \\ w_1 &= a, \\ R_1 &= \{r_1 : a \to aa, \quad r_2 : aa \to a\}. \end{aligned}$$

This P system is composed by one region (with label 1) and by two simple evolution rules, r_1 and r_2. The first rule duplicates the occurrences of symbol-objects a, while the second rule makes the opposite: reduces by half the number of occurrences of a's.

Moreover, we fix Pav_{r_1} and $Pav_{r_2} = 0.8$: this means that r_1 and r_2 have the same probability to be available at each step of the simulation. Also, we take $Cwin_{r_1}$ and $Cwin_{r_2} = 100$; this means that, if there is a conflict between r_1 and r_2, then both the rules have the same probability to *win* the conflict.

The two rules of the system can be considered as describing *associations* ($r_1 : a \to aa$) and *dissociations* ($r_1 : aa \to a$) of molecules, and this reminds the importance of such phenomena for the origin of life on Earth, [108]. Thus, models and software package, as considered above can also be related to this important issue of origin of life (whatever "life" means in this framework). Moreover, from a *formal language* point of view, it is natural to ask which is the link between systems as above and some kinds of Lindenmayer systems with probabilistic behavior.

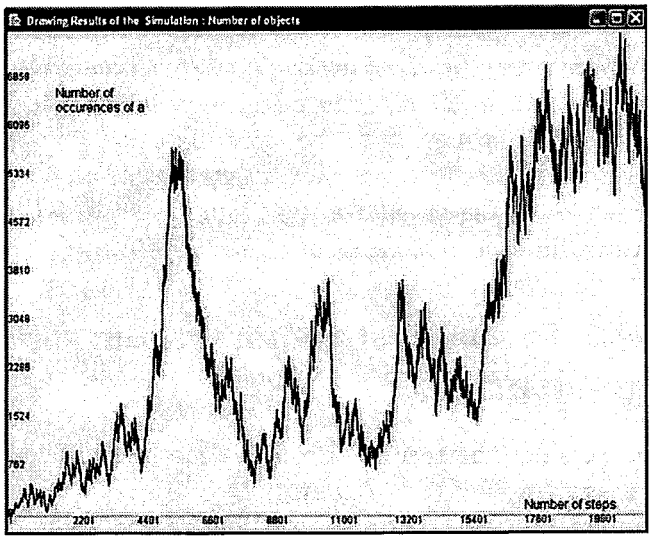

Figure 3.1: Running the software: the simulation of a simple probabilistic EC P system.

Even if the system constructed could seem very *stable* (all the probabilities are equal), the results (the number of occurrences of symbol-object a present in the system during the evolution – we can say, metaphorically, during the *life* of the system) of this experiment are quite surprising. The behavior of the system seems very *unstable*, and we notice big variations of the number of a's during the computation, with many big jumps (see Figure 3.1).

Roughly speaking, we can think that, for this simple system, with these coefficients and probabilities, we can expect any kind of behavior; this makes very hard to make predictions on the behavior. The picture in Figure 3.1 represents very well the controversial results of this experiment (the number of a's is indicated on the axis y and the number of the steps on the axis x. Moreover, it has been noticed that, lower we fix the probability for the two rules to be available, more unstable (with bigger jumps) the system becomes. We think that the study of this (simple) system could be a good starting point to understand more about the dynamics of probabilistic P systems.

3.3 Modeling Some Biological Processes

In this section we describe how some important cellular processes, such as the respiration process in *Escherichia coli* and the interactions between respiration and photosynthesis in cyanobacteria, can be modeled and translated in the P systems framework, and, how, in this way, using the simulator, it

is possible to obtain results that can be of interest for biologists.

Also, we intend to illustrate with a realistic example what we have discussed in the previous section.

3.3.1 Modeling a First Biological Process: The Respiration in Bacteria

Respiration is the biological process that allows the cells (from bacteria to humans) to obtain energy. In short, respiration promotes a flux of electrons from electron donors to a final electron acceptor, which in most cases is molecular oxygen.

In *Escherichia coli*, as well as in other bacteria, the cell ability to consume molecular oxygen during the respiration is determined by the presence of two different enzymes that catalyze the final step of respiration.

In *Escherichia coli*, these two terminal oxydases (enzymes) – called *terminal* because they are the last components of the respiratory process – are *cytochrome bd* and *cytochrome bo*. The occurrence of multiple (two or even more) types of terminal oxydases enables the cell to modulate its respiration, in agreement with its energy requirements and depending on the availability of chemicals in the environment [133].

For example, in *Escherichia coli*, cytochrome bd has a high affinity for oxygen and is involved in energy conversion with a medium efficiency: more exactly, for every electron (passed through the cytochrome bd to molecular oxygen) one proton (one atom of bound hydrogen without its electron) is transported from inside the cell to outside the cell.

Thus, because of its higher affinity for oxygen, the cytochrome bd works "more" at relative low oxygen concentration in the growing medium.

On the other hand, the cytochrome bo oxidase has a lower affinity for oxygen. Thus, cytochrome bo works at higher oxygen concentration in the growing medium; however it has an higher efficiency in energy conversion.

Recently, Alexeeva et al. [4, 5] studied the relationship between the oxygen concentration in the growing medium and the *activity rate* of the two terminal oxidases, including the flux of electrons to molecular oxygen through each of the two pathways.

Simply, but correctly (for more details the reader can consult the above cited papers), we can say that, at low oxygen concentration in the growing medium (lower than about 40% of oxygen saturation) the cytochrome bd oxydase is responsible for the entire respiratory activity of the cells; in other words, the flux of electrons to molecular oxygen proceeds 100% through the cytochrome bd oxydase. At high oxygen concentration in the growing medium (this means in between 90% and 100 % of oxygen saturation), the cytochrome bo oxydase is responsible for almost the entire respiratory activity of the cells. Furthermore, in between 40% and 90%, the two types of terminal oxidases contribute together to the respiration of cell.

Now, we show how to *translate* this biological reality into the P systems framework such that it is possible to use the simulator.

We know that, in *Escherichia coli*, the consumption of molecular oxygen is made using two different chemical reactions catalyzed by two different kinds of enzymes: *bo* and *bd* (for simplicity of notation we call *bo* as e'' and *bd* as e'). The activity rate of these two enzymes is different, according to the percentage of saturation of molecular oxygen: this means that the activity rate of the enzymes determines how many chemical reactions catalyzed by such enzymes occur in a fixed unit time.

We can represent the two reactions in the following (informal) way:

$$e'ab \rightarrow ce',$$
$$e''ab \rightarrow ce'',$$

where e'' and e' are the two enzymes that catalyze the two reactions, b represents the molecular oxygen (consumed by the two reactions), a the hydrogen and c is the molecular water produced.

As we can see, both the two chemical reactions consume oxygen and hydrogen but the activity rate of the two reactions is different, according to the concentration of oxygen (substrate of the reaction) in the cell, because it depends on the activity rate of the enzymes e' and e'' (which depends on the concentration of oxygen in the cell, as discussed before). Moreover, only the occurrences of an enzyme that are active can be used to catalyze the reaction, and the number of active occurrences change according to some parameters (in this case, the concentration of oxygen).

We have to observe that at a low concentration of oxygen there could be conflicts among the two reactions for using this chemical; these conflicts must be solved by using the different affinity of the two enzymes with respect to the oxygen and this means to use the *probability to win* associated to the two rules, in the way previously discussed.

We now show how one can use the simulator to model the process of respiration in *Escherichia coli* described above and how, in this way, we can have some useful results for biologists. Initially we consider only one of the two reactions that take place in *Escherichia coli* during respiration, and we try to detail it and to simulate this reaction using the software.

In particular we consider a system where only the first enzyme e' (*cytochrome bd*) is present. It is know, see, e.g., [124], that the activity of an enzyme can be expressed as the activity of each *unit* enzyme and the quantity of substances involved in the reactions can be given in *nanomols*. Then, using units of enzyme and nanomols, the chemical reaction introduced before,

$$e'ab \rightarrow ce'$$

must be rewritten in a more precise way:

$$E' A^k B^j \rightarrow C^i E'. \tag{3.1}$$

This means that one unit of the enzyme E' (*cytochrome bd*) catalyzes a reaction that consumes k nanomols of hydrogen (A) and j nanomols of oxygen (B), producing i nanomols of water (C). Actually, for the purpose of this experiment, we are only interested in the consumption of oxygen, and then we can see the reaction in a the following simpler way:

$$E' B^j \rightarrow E'.$$

From [124] we know that 1 unit of enzyme E' consumes (more precisely, we should say "catalyzes the reaction that consumes") 42 nanomols of oxygen in 1 hour.

We must specify that this is the activity rate of the enzyme when the concentration of oxygen (substrate) is lower than 40% of saturation and we suppose this as true for our experiment. Moreover, we suppose that our system is closed (no oxygen is introduced from the environment) and we maintain constant the level of concentration of oxygen.

Using these data we can rewrite the chemical reaction in the following way:

$$E' B^{42} \rightarrow E'.$$

In this way we have made the translation from the biological reality to the mathematical model: now each occurrence of the symbol-object E' represents one unit of enzyme *cytochrome bd*, each occurrence of the symbol-object B represents one nanomol of oxygen and each step of the P system is one hour (that is, the time unit of the activity rate of the enzyme).

Now we can simulate the consumption of oxygen in the respiration process of *Escherichia coli*, using the software on the following simple P system:

$$\Pi = (O, \mu, w_1, R_1) \tag{3.2}$$

with:

$$\begin{aligned} O &= \{E', B\}, \\ \mu &= [_1\,]_1, \\ w_1 &= (E')^k B^j, \\ R_1 &= \{E' B^{42} \rightarrow E'\}, \end{aligned}$$

where *Papp* is 1.0 for the rule in R_1 (we suppose the reaction is always available and the *Pwin* is not relevant here because there is only one rule).

In the formal definition of the P system we have left as parameters the number of occurrences of nanomols of oxygen (j) and the number of

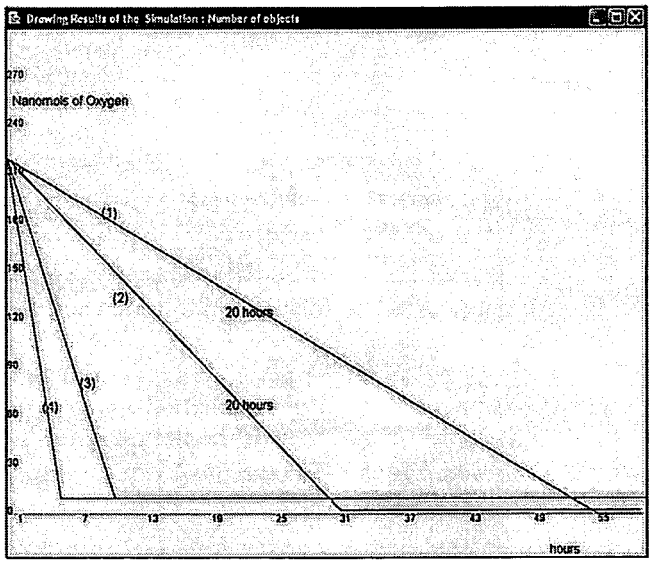

Figure 3.2: Consumption of oxygen in *Escherichia coli* using only *cytochrome bd.*

unit enzymes (k) present at the beginning of the experiment; in this way, changing the values of this parameters, we can run different experiments for different biological situations.

An immediate application for our simulator is to have diagrams of the consumption of oxygen, changing the quantity k (expressed in unit) of enzyme *cytochrome bd* used.

In other words, we are interested in looking how fast the oxygen decreases, using different quantities of enzymes. In this way we can check in a real experiment which quantity of enzymes we have used by looking in what extent the consumption obtained in laboratory is "similar" to the diagrams calculated by the simulator. Of course, the checking of the "similarity" can be automated, using the software itself and making, in this way, an on-line checking of the quantity of enzymes used in an experiment. We illustrate this application with a simple practical example.

In Figure 3.2 it is shown the simulation of the P system described before, where the consumption of oxygen is reported, when the quantity of enzyme used is 1/10 of unit of enzyme (1), 1/6 of unit of enzyme (2), 1/2 of unit of enzyme (3) and exactly 1 unit of enzyme (4).

On the x axis one reports the time of the experiment (expressed in hours), while on the y axis one reports the quantity of oxygen (in nanomols) presents in the cell; the initial quantity of oxygen is 220 nanomols (for this first step and for the goal of our experiment we can take, as approximation, that the activity rate of the enzymes used is constant in the time but we

have to remark that, in practice, the activity rate of the enzyme changes according to the concentration of oxygen).

Now we want to consider the consumption of oxygen in *Escherichia coli*, where only the second enzyme e'' (*cytochrome bo*) is present.

Using units to express the quantity of enzyme *cytochrome bo* and nanomols for the quantity of oxygen and hydrogen, the reaction for the consumption of oxygen is:

$$E''A^kB^j \rightarrow C^iE''.$$

This formula is interpreted exactly as the formula described in (3.1) except the fact that here the enzyme used is E'' indicating the *cytochrome bo*.

Again, because we are only interested in the consumption of oxygen, the reaction can be simplified and written as:

$$E''B^j \rightarrow E''.$$

From [124] we know that 1 unit of enzyme E'' (*cytochrome bo*) consumes (i.e., catalyzes the reaction that consumes) 66 nanomols of oxygen in 1 hour. In this experiment we suppose that the concentration of oxygen (substrate) is higher than 90% of the saturation.

Also we suppose that enough oxygen is introduced from the environment to the system considered (to maintain the same level of saturation).

Using these data we can rewrite the reaction for the consumption of oxygen in the following way:

$$E''B^{66} \rightarrow E''.$$

Each occurrence of the symbol-object E'' represents one unit enzyme of *cytochrome bo*, each occurrence of the symbol-object B represents one nanomol of oxygen and each step of the P system is one hour.

The consumption of oxygen in the respiration process of *Escherichia coli*, when only *cytochrome bo* is used, can be studied running the simulator on the following P system:

$$\Pi = (O, \mu, w_1, R_1) \tag{3.3}$$

with:

$$\begin{aligned} O &= \{E'', B\}, \\ \mu &= [_1\,]_1, \\ w_1 &= (E'')^kB^j, \\ R_1 &= \{E''B^{66} \rightarrow E''\}, \end{aligned}$$

where *Papp* is 1.0 for the rule in R_1 (we suppose the reaction is always available and the *Pwin* is not relevant here because there is only one rule).

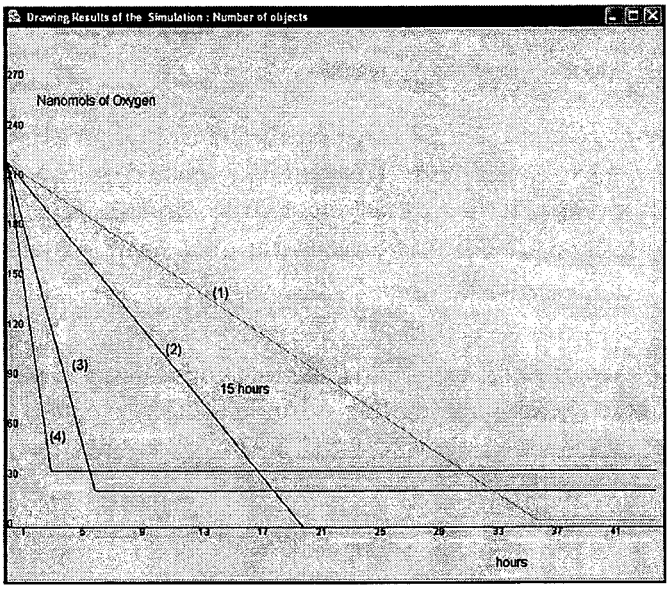

Figure 3.3: Consumption of oxygen in *Escherichia coli* using only *cytochrome bo*.

As in the case of *cytochrome bd* considered before, in Figure 3.3, we draw the diagram representing the consumption of oxygen, when the quantity of (the unique) enzyme *cytochrome bo* used is 1/10 of unit of enzyme (1), 1/6 of unit of enzyme (2), 1/2 of unit of enzyme (3) and exactly 1 unit of enzyme (4), and starting from an initial amount of oxygen that is 220 nanomols.

Now we consider the consumption of oxygen in *Escherichia coli* when both the enzymes cytochrome *bo* and cytochrome *bd* oxydase are present.

In this case the two enzymes are, somehow "concurrent": both the enzymes catalyze the consumption of oxygen. Because the oxygen is a common resource for the two enzymes then the ability of an enzyme to catalyze the consumption of oxygen is related with its affinity, as discussed in Section 3.3.1.

From [4, 5] we can find out that, when the oxygen present is 220 nanomols, then the affinity with respect to oxygen of the enzyme *bo* is almost 8 times bigger than the affinity of the enzyme *bd*.

Now we can simulate the consumption of oxygen in the respiration process of *Escherichia coli*, using the software on the following simple P system that is obtained by "composing" the two P systems previously considered in (3.2) and (3.3):

$$\Pi = (O, \mu, w_1, R_1), \tag{3.4}$$

where

$$O = \{E', E'', B\},$$

$$\begin{aligned}\mu &= [_1\,]_1,\\ w_1 &= (E')^k(E'')^pB^j,\\ R_1 &= \{r_1 : E'B^{42} \to E', r_2 : E''B^{66} \to E''\},\end{aligned}$$

where *Papp* is 1.0 for the rules in R_1 (we suppose the reactions are always available).

The two rules r_1 and r_2 compete for the use of the oxygen B. Such conflicts are solved using the *probability to win* that simulates the affinity of an enzyme and that works in the way described in Section 3.1. Therefore we fix the coefficients C_{win} of the two rules in R_1 such that, in case of conflict, the probability of r_2 to win a conflict is 8 times bigger the probability to win of the rule r_1.

We fix the parameters k, p, j and then we run the simulator with initial amount of oxygen fixed at $j = 220$ nanomols; we obtain Figure 3.4 that represents how the consumption of oxygen changes if only 1/5 of unit of the enzyme *bo* is used (3), only 1/5 of unit of the enzyme *bd* is used (1), or, (2) the enzymes *bd* and *bo* are used together (in quantity 1/10 of unit, each one). Notice that the difference between the cases when the enzymes are used separately and when together is not so strong; when the two enzymes are used together then the speed of the consumption of oxygen is around the half respect to the case when only the enzyme *bo* is used and the double respect to the case when only the enzyme *bd* is used. In practice, a stronger difference would be noticed when the quantity of oxygen available is very low and then the affinity of the two enzymes becomes a fundamental parameter.

3.3.2 Pumps in Escherichia coli

As already discussed in the Chapter 1.2 the proton pumping mechanism is fundamental for biological energy conservation.

In *Escherichia coli* the consumption of molecular oxygen is related to the translocation (pumping out) of protons outside the cell.

Specifically, *cytochrome bd* oxydase translocates one proton for every electron transported to molecular oxygen. Following the data presented in [4, 5], we conclude that 42 nanomols of protons are pumped out, using one unit of enzyme bd oxydase.

In the case of *cytochrome bo* oxidase, two protons are translocated for every electron transported to molecular oxygen. This means that, following the data presented in [4, 5], 132 nanomols of protons are pumped out, in one hour, using one unit of enzyme bo oxidase.

Formally, the following P system represents the situation where only consumption of molecular oxygen and translocation of protons are taken in consideration and where only the enzyme *cytochrome bo* oxidase is present.

$$\Pi = (O, \mu, w_1, R_1, R_1') \tag{3.5}$$

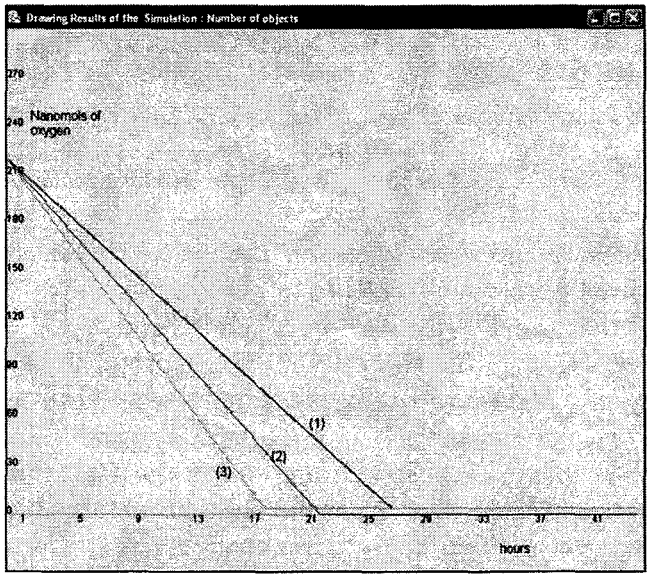

Figure 3.4: Consumption of oxygen in *Escherichia coli* using *cytochrome bo* and *bd*.

with:

$$\begin{aligned} O &= \{E'', O, P\}, \\ \mu &= [_1\,]_1, \\ w_1 &= E''^k O^j, \\ R_1 &= \{E''O^{66} \rightarrow E''P^{132}\}, \\ R_1' &= \{(P, out)\}, \end{aligned}$$

where $Papp = 1$ (the reaction is always available; $Pwin$ is not relevant); each step of this system corresponds to 60 minutes. Each occurrence of the symbol-object O corresponds to one namomol of molecular oxygen; each occurrence of the symbol-object P corresponds to one nanomol of protons and each occurrence of the symbol-object E'' corresponds to one unit of enzyme cytochrome bo oxidase. The consumption of molecular oxygen is simulated by using the evolution rule in R_1 while the translocation of protons toward the environment is simulated by using the symport rule present in R_1'.

Figure 3.5 represents the accumulation of protons in the environment when 1/10 of unit of enzyme is used and 220 nanomols of molecular oxygen are initially present in the cell.

The diagrams presented in Figures 3.2, 3.3, 3.4 and 3.5 can be used to obtain biological significant results. For instance, suppose that for an experiment we take exactly 220 nanomols and a certain quantity of the enzyme

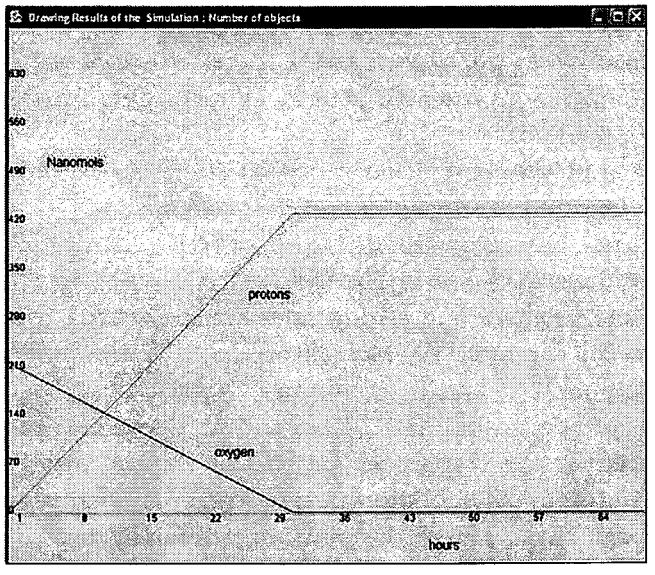

Figure 3.5: Translocation of protons in *Escherichia coli* using *cytochrome bo*.

cytochrome bd. Suppose we observe that, after 20 hours of respiration, the *Escherichia coli* still contains (around) 140 nanomols of oxygen (this can be measured in lab). Then, looking at the diagram represented in Figure 3.2 we can understand that the quantity of enzymes present in the cell is around 1/10 of unit (the same idea can be applied to the other diagrams obtained).

In this case the calculation is quite easy, but the same idea, together with an improved version of the simulator, can be adapted to calculate, on-line, the quantity of enzymes used also when the activity rate of each enzyme changes during the experiment, according to the concentration of the oxygen substrate.

3.3.3 Respiration-Photosynthesis Interaction in Cyanobacteria

Cyanobacteria are the largest and most diversified, important group of prokaryotes [128], defined by the ability to carry out both oxygenic photosynthesis (within the thylakoid membranes) and respiration (within the plasma (cell) membranes) [128], [129]. In this section we will use, for our simulations, *Synechocystis PCC 6803* that is a typical kind of cyanobacteria.

Shortly, the overall process of photosynthesis consists in using electrons from water to ultimately reduce carbon dioxide, producing some chemicals (for example, carbohydrates). This process is essential for the life on Earth, being the main food source for almost all living cells and it is the only source

of molecular oxygen needed for respiration.

The first reaction in photosynthesis is the splitting of water (at the expense of light energy, not presented here for simplicity) to produce molecular oxygen, protons and electrons [86].

The last step of the respiration process is the opposite reaction of the first reaction that occurs in photosynthesis; this mean that, in this case, water is produced by the consumption of oxygen during its combination with protons and electrons, as in *Escherichia coli.*

In cyanobacteria there is a strong interaction between respiration and photosynthesis; for example, the oxygen produced in the photosynthesis in the inner membrane (thylakoid membrane) is used in the cell membrane for the process of respiration, [127, 129], and at the same time the carbon dioxide produced by the respiration process in the cell membrane is used (recycled) in the inner membrane for the process of photosynthesis. For simplicity, we can say that the oxygen production (by the photosynthesis process) is catalyzed by only one type of enzymes (actually, this is called oxygen-evolving complex, [86]) and the last step of respiratory oxygen consumption in cyanobacteria is catalyzed by the enzyme cytocrome c oxydase [129].

Following the data presented in [16], the photosynthetic oxygen production is of 26 nanomols of oxygen per 1 microgram of chlorophyll in 10 minutes, while oxygen consumption in respiration is of 5 nanomols of oxygen per 1 microgram chlorophyll in 10 minutes. The carbon dioxide production in respiration is of 5 nanomols per 1 microgram chlorophyll (present in the cyanobacteria) in 10 minutes, while the carbon dioxide consumption in photosynthesis is of 26 nanomols of carbon dioxide per 1 microgram chlorophyll in 10 minutes. We have to notice that more the data are detailed, more the simulation results are near to the biological reality; for our purpose, in this case, the data expressed in nanomols and micrograms are of a good precision. The system considered here is closed: there is no exchange of chemicals with the environment and this means that oxygen and carbon dioxide are not introduced in the system. We suppose, for simplicity, that we have inexhaustible quantities of water and light.

Then, in a formal simplified way, we can represent the process of respiration and photosynthesis in *Synechocystis* with the following EC P system:

$$\Pi = (O, \mu, w_1, w_2, R_1, R_2, R_1', R_2') \tag{3.6}$$

with:

$$\begin{aligned} O &= \{c, O, D\}, \\ \mu &= [_1\, [_2\,]_2\,]_1, \\ w_1 &= c^k O^j, \\ w_2 &= c^l D^m, \end{aligned}$$

$$
\begin{aligned}
R_1 &= \{cO^5 \to cD^5\},\\
R_2 &= \{cD^{26} \to cO^{26}\},\\
R_1' &= \emptyset,\\
R_2' &= \{(D, in), (O, out)\},
\end{aligned}
$$

where $Papp = 1$ for each rule in R_1, R_2, R_2' (we suppose the reaction is always available and the $Pwin$ is not relevant here because there are no conflicts between the rules) and each step of this system corresponds to 10 minutes.

The system is composed of two membranes labeled with 1 and 2 that represent, the cell membrane and the thylakoid membrane, respectively. In the two regions there are the symbol-object c (each occurrence of c represent 1 microgram of chlorophyll), the symbol-object D (each occurrence of this symbol-object represents 1 nanomol of carbon dioxide) and the symbol-object O (each occurrence of this symbol-object represents 1 nanomol of oxygen).

In region 1, between membranes 1 and 2, we have the simple evolution rule $cO^5 \to cD^5$ that represents the last step of the respiration that occurs in the cell membrane. In particular, the rule says that, in one step, 5 occurrences (nanomols) of O (oxygen) are consumed (using 1 occurrence (microgram) of chlorophyll c) and 5 occurrence (nanomols) of D (carbon dioxide) are produced. In region 2, there is the simple evolution rule $cD^{26} \to cO^{26}$ that represents the oxygen production by photosynthesis that happens in the thylakoid membrane. In particular, the rule says that in one step, 26 occurrences (nanomols) of D (carbon dioxide) are consumed (using 1 occurrence (microgram) of chlorophyll c) and 26 occurrences (nanomols) of O (oxygen) are produced.

In our general model we did not fix the quantity of O (oxygen), D (carbon dioxide) and c (chlorophyll) that is present in the cell at the beginning of the simulation, and in this way we can apply the same model for different experimental data.

In the first simulation we take, at the beginning, 1 microgram of chlorophyll, 300 nanomols of oxygen in the cell membrane and 0 in the thylakoid membrane, 500 nanomols of carbon dioxide in the thylakoid membrane and 0 in the cell membrane. The result of the simulation of this system is given in Figure 3.6.

We can notice that, after some time (in this case after 23 steps = 230 minutes), the oxygen stops to accumulate at around 770 nanomols; around 80 minutes we have the same amount of oxygen in the cell membrane and of carbon dioxide in the thylakoid membrane.

In the second simulation we consider a system that does not contain any oxygen at the beginning and we want to see how it is able to produce oxygen and when the accumulation of oxygen stops. In particular, we take at the beginning 1 microgram of chlorophyll, 0 nanomols of oxygen in the

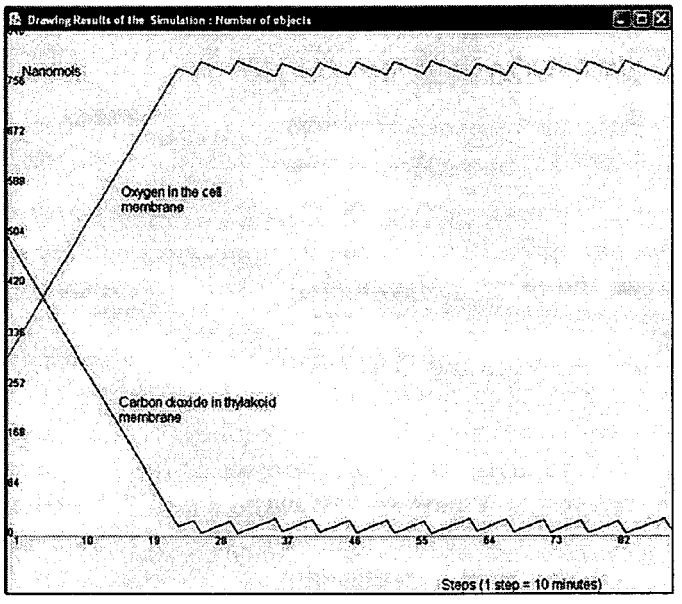

Figure 3.6: Oxygen (cell membrane) - carbon dioxide (thylakoid membrane) in cyanobacteria (first simulation).

cell membrane and 0 in the thylakoid membrane, 450 nanomols of carbon dioxide in the thylakoid membrane and 0 in the cell membrane.

The result of the simulation of this system is given in Figure 3.7.

We can notice that after some time (around 21 steps = 210 minutes) the accumulation of oxygen stops at around 440 nanomols; around 110 minutes we have the same amount of oxygen in the cell membrane and of carbon dioxide in the thylakoid membrane.

The simulation presented in Figure 3.7 corresponds to the situation when the cyanobacteria are in a medium without molecular oxygen and this situation is a practical interest because it that can easily occurs in laboratory or in natural environments, in conditions of prolonged darkness. When the light is turned on (step 0) then in the cyanobacteria the photosynthetic activity starts. The results presented in Figure 3.7 permit us to predict the time after which the oxygen stops to accumulate (in absence of oxygen at the beginning) and the quantity of molecular oxygen in the cell.

We want to stress again the fact that the systems considered in the two previous experiments are closed and oxygen is accumulated both in the liquid phase and in the gaseous one (for simplicity, in our simulation we did not take into account the well-know inhibition of photosynthesis by the increasing of oxygen concentration).

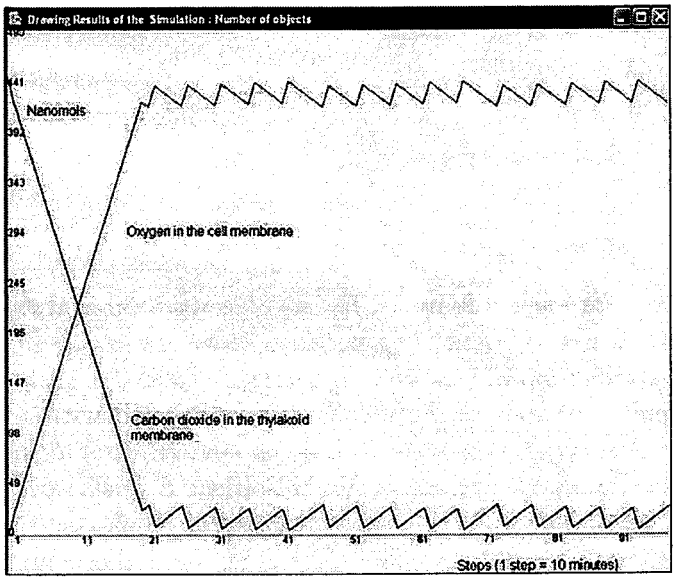

Figure 3.7: Oxygen (cell membrane) - carbon dioxide (thylakoid membrane) in cyanobacteria (second simulation).

3.3.4 Pumps in Cyanobacteria

Now we use the simulator to study the process of *proton translocation* in the case of *Anacystis nidulans* that is another kind of cyanobacteria. The proton translocation consists in the pumping of protons, outside the cell, when oxygen is consumed. Therefore the decrease of oxygen concentration inside the cell corresponds to an increase of protons concentration outside the cell.

From [127] we can get the experimental data relating oxygen consumption and protons translocation. In particular considering 1 mg of dry weight of *Anacystis nidulans*, using one unit of enzyme, in 2 minutes 5 nanomols of oxygen are consumed and, at the same time, 21 nanomols of protons are sent out (translocated) from the cell and accumulated in the environment.

We can simulate this mechanism using the software.

In a formal simplified way, considering only the reaction involving oxygen and protons, we can represent the process of pumps in *Anacystis nidulans* with the following EC P system:

$$\Pi = (O, \mu, w_1, w_2, R_1, R_2, R_1', R_2') \tag{3.7}$$

with:

$$\begin{aligned} O &= \{E, O, P\}, \\ \mu &= [_1\, [_2\,]_2\,]_1, \end{aligned}$$

$$
\begin{aligned}
w_1 &= E^k O^j, \\
w_2 &= \lambda, \\
R_1 &= \emptyset, \\
R_2 &= \{EO^5 \rightarrow EP^{21}\}, \\
R_1' &= \emptyset, \\
R_2' &= \{(P, out)\}.
\end{aligned}
$$

where $Papp = 1$ for each rule in R_1, R_2, R_2' (the reaction is always available and the $Pwin$ is not relevant here because there are no conflicts between the rules) and each step of this system corresponds to 2 minutes.

The system is composed of two membranes labeled with 1 and 2 that represent the *Anacystis nidulans* cell and the environment, respectively.

In the two regions there are the symbol-object E (each occurrence of E represent 1 unit of enzyme), the symbol-object O (each occurrence of this symbol-object represents 1 nanomol of oxygen) and the symbol-object P (each occurrence of this symbol-object represents 1 nanomol of protons).

The consumption of oxygen takes place in region 2 and using the symport rule presents in R_2' the protons are sent out and accumulated in region 2.

Running the simulator over the P system described in 3.7, with $k = 1$, $j = 220$, we obtain the diagram of Figure 3.8 describing the quantity (in nanomols) of oxygen contained in the cell (2), and the quantity (in nanomols) of protons accumulated in the environment (1).

3.4 Final Remarks

In the last years many research efforts have been dedicated to the investigation of membrane systems as modeling tools for biological systems and biological processes. In this framework membrane systems are used as a modeling mechanism, instead of computing mechanism. The usual way these investigations proceed is the following: one examines a piece of biological reality, one models it by using an appropriate class of membrane systems, and one constructs a simulator modeling this class; in this way several experiments and simulations can be done in a much cheaper with respect to laboratory experiments. For instance, using a simulator, one can look how, by changing some parameter, one can change the evolution of the system (for instance, the amount of certain produced objects). There are now in literature several computer programs simulating various types of P systems. A survey of these simulators, together with their specific features, can be find in the chapter [83] of the volume [63]. A general reference for applications of membrane systems in biology are the specific chapters of the volume [63].

The software and the experiments presented in this chapter have been recalled from [18] and [47]. We do not give here further biological references

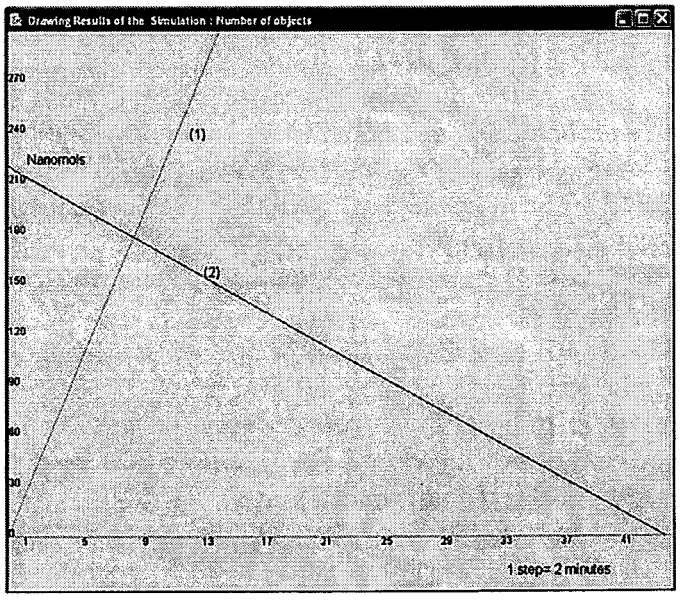

Figure 3.8: Consumption of oxygen - translocation of protons in *Anacystis nidulans*.

since we have already given them while presenting the experiments.

In the literature there are several other papers dealing with cellular processes modeled by membrane systems.

For instance, the problem of modeling photosynthesis has been also investigated in [112].

The activity of mechanosensitive channels has been modeled by membrane systems as described in [33]; an extended version of this work can be found in the corresponding chapter, [17], of the volume [63].

In [125] and [126] *continuous P systems* have been used to model the EGFR signaling cascade. Continuous membrane systems use real numbers instead of discrete mathematics and seem very powerful devices to model many interesting biological processes.

In [36] P systems have been used to model and simulate the circadian rhythms of *Drosophila melanogaster*; there has been also done a comparison of the obtained results with the data present in literature, usually obtained by methods based on differential equations. A survey of the possible use of membrane systems for modeling biological dynamics can be found in the chapter [35] of the volume [63].

In [30] the quorum sensing regulatory networks of the bacterium *Vibro fisceri* is modeled by using a (cell-like) P system with a type of boundary rules introduced in [31] and with each rule having associate a certain rate of application. The approach used seems independent from the particular

choice of P system model and can be adopted to model other biological processes, maybe by adapting other P systems models.

In [130] it is introduced a dynamical probabilistic P systems, that is a P system where probabilities associated to the rules change during the evolution of the system, according to the current form of the multiset (amount of certain objects). The model is used to describe the evolution of complex systems, in particular in the paper the authors show an application to the *Brusselator*, [132], a well-known and simplified scheme which describe the *Belousov-Zhabotinskii* reaction.

We should also mention that the use of discrete mathematics for modeling cellular processes has been explored also in other frameworks, for instance, the recent introduced *brane calculi*, based on process algebra, [44]. Recently brane calculi and membrane systems have also been joined, [45], and further papers are in preparation, [39].

Chapter 4

Proton Pumping and Time-Free P Systems

In this chapter we come back to mathematics showing how new mathematical devices together with new interesting mathematical problems can be obtained by looking to the biological processes that take place in living cells.

In other words, as already mentioned before, this chapter wants to be one more "proof" of the importance of the interaction existing between mathematics and biology. In fact, we present here two topics that are directly inspired by biological facts.

Initially, we discuss a model called *proton pumping P system* that is, essentially, an evolution-communication model with a restriction (on the antiport rules) inspired by the pumping of protons present in living cells.

As we have seen in Section 1.2, in many bacteria, the antiports available are those that can exchange a proton (previously moved/pumped) with some chemical objects; then, a natural step is to add such restrictions to the evolution-communication model.

Therefore, a proton pumping P system is an evolution-communication P system with a set of special objects called *protons* that are never created and never destroyed and where only antiport rules that can exchange some symbol-objects (also protons among them) for a single proton are admitted (inspired by biology, such antiport rules are called *proton pumping rules*).

The movement of protons as well as the other biochemical reactions that happen in living cells need a certain time to be completed. Therefore, the classical definition of membrane systems where each rule is executed in one clock-step does not seem to have a biological counterpart. It is then natural to search for a class of P systems where to each rule is assigned a certain time of execution.

On the other hand, the time of execution of biochemical reactions and processes may depend on many factors, some of them not easily predictable;

therefore, we are interested in a special class of P systems, called *time-free*, producing always the same result independently from the time of execution of the rules; in biological terms, this means that we are interested in systems that are stable with respect to (possible and, maybe, unpredictable) changes of the time that the reactions need to be completed.

A possible mathematical formalization of this idea is the following one: we associate to each rule r a certain integer value $e(r)$ (by using a time-mapping e) representing the execution time of the rule; then we consider systems that generates (or accepts) the same family of vectors of natural numbers, independently of the values assigned to the parameter $e(r)$, for any rule r present in the system.

As in real world where synchronization of independent events is fundamental, also in the case of time-free P systems a similar problem must be considered: in order to get time-free systems that are computationally enough powerful it is important to synchronize rules with different execution times and that run in parallel. In this chapter we present several approaches to this problem.

Initially, we show how signals can be useful to tackle this problem; in particular, several results on time-free P systems with signal-promoters are presented. Signal-promoters are inspired from biological signals that in living cells control the enzyme activation/deactivation and in general the synchronization of biochemical processes.

On other hand, synchronization can also be obtained by using a restricted kind of cooperative evolution rules like bi-stable catalytic rules.

Finally, we show how universal time-free systems can be obtained by making use of synchronization based only on communication (symport and antiport rules). Moreover, also the problem of deciding whether an arbitrary P system is time-free is investigated.

4.1 Proton Pumping P Systems

We introduce in this section the idea and the definition of a proton pumping P system and we present an universality proof for this model.

The definition is similar to the one given for EC P systems but with restricted type of antiport rules; the similarity between *catalysts* and protons should also be noticed. In fact, in both cases such special objects are never created and never destroyed; while catalysts are used to help some evolution rule to be applied, the protons are used to help some communication rules to be applied.

We show that proton pumping P systems are universal, providing that they can use *sufficient* types of *different* protons during the computation.

First we recall the definition of a proton pumping P system.

Definition 4.1 *A proton pumping P system of degree $m \geq 1$ is defined as*

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

where $(O, \mu, w_1, \cdots, w_m, R_1, \cdots, R_m, R_1''' \cup R_1'' = R_1', \cdots, R_m''' \cup R_m'' = R_m', i_0)$ is an evolution–communication P system, $P \subseteq O$ is the set of protons, R_i''' are the sets of symport rules and R_i'' are the sets of antiport rules (proton pumping rules) of the form $(p, in; x, out)$ or $(x, in; p, out)$ where $x \in O^+$ and $p \in P$. Every evolution rule is of the form $u \to v$, where $u \in (O - P)^+, v \in (O - P)^*$.

The computation of a proton pumping P system evolves like in the case of an evolution–communication P system, hence a transition between configurations is governed by the mixed application of the evolution rules and of the symport/antiport rules.

All objects which can be the subject of the rules from the sets R_i, R_j''', R_j'', $1 \leq i \leq m, 1 \leq j \leq m$, have to evolve by such rules. As usual, the rules from R_i are applied to objects in region i and the rules from R_i''' and R_i'' govern the communication of objects through membrane i. There is no difference between evolution rules and communication rules (symports and proton pumping rules): they are chosen and applied in the non-deterministic maximally parallel manner. The system continues parallel steps until there remain no applicable rules (evolution rules or symport/antiport rules) in any region of Π. Then the system halts, and we consider the multiplicities of objects contained in the output region i_0, at the moment when the system halts, as the result of the computation of Π.

The following notation is used:

$$PsProP_m^k(i, j, \alpha), \alpha \in \{ncoo, coo\} \cup \{cat_k \mid k \geq 0\},$$

denotes the family of sets of vectors of numbers generated by proton pumping P systems with at most m membranes (as usually, $m = *$ if such a number is unbounded), k different types of protons (i.e., k is the cardinality of the set P), using symport rules of weight at most i, antiport rules of weight at most j, and evolution rules that can be cooperative (coo), non-cooperative ($ncoo$), or catalytic (cat_k), using at most k catalysts.

4.1.1 An Universality Result

In this section we present an universality result for proton pumping P systems. The universality has been obtained by simulating programmed grammars with appearance checking. The constructed system uses an unbounded number of protons, but their number can be bounded (for a reference, see the Final Remarks section).

Theorem 4.1 $PsProP_3^*(1, 1, ncoo) = PsRE$.

Proof. Let $L \in RE$. Then there exists a programmed grammar with appearance checking $G = (N, T, S, P)$ generating L, where $Lab(P) = \{i \mid 1 \leq i \leq m\}$ and $N = \{X_i \mid 1 \leq i \leq n\}$. Moreover $N' = N \cup \{h\}$, $\overline{N'} = \{\overline{X} \mid X \in N'\}$ and we define the morphism $\gamma : (N' \cup T)^* \longrightarrow (\overline{N'} \cup T)^*$ by $\gamma(x) = \overline{x}, x \in N'$, and $\gamma(x) = x$, $x \in T$. We construct the following system:

$$\Pi = (O, P', \mu, w_1, w_2, w_3, R_1, R_2, R_3, R_1''', R_2''', R_3''', \emptyset, R_2'', R_3'', 0),$$

where:

$$\begin{aligned}
O &= P' \cup T \cup N' \cup \tilde{N}' \cup \overline{N'} \cup C \cup A \cup M \text{ for :} \\
P' &= \{p_i, q_i \mid i \in Lab(P)\} \cup \{r_j \mid X_j \in N\} \cup \{p_0, c, F\}, \\
\tilde{N}' &= \{\tilde{X} \mid X \in N'\}, \\
C &= \{b_i^{(j)}, d_i^{(j)} \mid i \in Lab(P), 1 \leq j \leq 5\} \cup \{l, g, H\}, \\
A &= \{e_j, e_j' \mid X_j \in N\} \cup \{f_j \mid 1 \leq j \leq 5\}, \\
M &= \{t_j \mid -2 \leq j \leq 5\} \cup \{\#, K, s\}, \\
\mu &= [_1\, [_2\, [_3\,]_3\,]_2\,]_1, \\
w_1 &= Fsr_1r_2 \cdots r_n,\ w_2 = p_0cK,\ w_3 = Shp_1p_2 \cdots p_mq_1q_2 \cdots q_m, \\
R_1 &= \{\tilde{X} \to \gamma(x)d_i^{(5)} \mid (i : (X \to x, E_i, F_i)) \in P\} \\
&\cup \{\tilde{h} \to \overline{h}b_i^{(5)}e_jf_5 \mid (i : (X_j \to x, E_i, F_i)) \in P\} \\
&\cup \{\tilde{X} \to \# \mid X \in N\} \cup \{f_j \to f_{j-1} \mid 2 \leq j \leq 5\} \\
&\cup \{b_i^{(5)} \to \#, d_i^{(5)} \to \# \mid i \in Lab(P)\} \cup \{\# \to \#\}, \qquad (4.1) \\
R_2 &= \{X \to \tilde{X}t_5 \mid X \in N'\} \cup \{t_j \to t_{j-1} \mid -1 \leq j \leq 5\} \\
&\cup \{b_i^{(5)} \to b_i^{(4)}lg, d_i^{(5)} \to d_i^{(4)}lg \mid i \in Lab(P)\} \\
&\cup \{b_i^{(j)} \to b_i^{(j-1)}, d_i^{(j)} \to d_i^{(j-1)} \mid i \in Lab(P), 2 \leq j \leq 4\} \\
&\cup \{e_j \to e_j' \mid X_j \in N\}, \qquad (4.2) \\
R_3 &= \{\overline{X} \to X \mid X \in N'\} \cup \{g \to H, K \to K\}, \qquad (4.3) \\
R_1''' &= \{(a, out) \mid a \in T\}, \qquad (4.4) \\
R_2''' &= \{(\tilde{X}, out), (\overline{X}, in) \mid X \in N'\} \cup \{(e_j, in) \mid X_j \in N\}, \qquad (4.5) \\
R_3''' &= \{(\overline{X}, in) \mid X \in N'\} \cup \{(g, in)\}, \qquad (4.6) \\
R_2'' &= \{(d_i^{(5)}, in; p_j, out), (b_i^{(5)}, in; p_j, out) \mid i \in E_j\} \\
&\cup \{(d_i^{(5)}, in; q_j, out), (b_i^{(5)}, in; q_j, out) \mid i \in F_j\} \\
&\cup \{(d_i^{(5)}, in; p_0, out), (b_i^{(5)}, in; p_0, out) \mid i \in Lab(P)\} \\
&\cup \{(p_j, in; l, out) \mid j \in Lab(P) \cup \{0\}\} \\
&\cup \{(q_j, in; l, out) \mid j \in Lab(P)\} \\
&\cup \{(r_j, in; e_j', out), (f_1, in; r_j, out) \mid X_i \in N\} \\
&\cup \{(s, in; c, out), (F, in; s, out)\}, \qquad (4.7)
\end{aligned}$$

$$
\begin{aligned}
R_3'' &= \{(c, in; X, out) \mid X \in N'\}, \\
&\cup \{(d_i^{(1)}, in; p_i, out), (b_i^{(1)}, in; q_i, out) \mid i \in Lab(P)\} \\
&\cup \{(p_i, in; H, out) \mid i \in Lab(P) \cup \{0\}\} \\
&\cup \{(q_i, in; H, out) \mid i \in Lab(P)\} \\
&\cup \{(r_j, in; X_j, out), (K, in; r_j, out) \mid X_j \in N\} \\
&\cup \{(t_{-2}, in; c, out), (F, in; X, out), (K, in; F, out)\}. \qquad (4.8)
\end{aligned}
$$

The goal of this proof is to show that the proton pumping P system is universal when it can use *sufficient* types of protons during the computation (we recall that the number of types of protons is the cardinality of the set P' of the P system).

To prove the theorem we just need to show how the constructed P system Π simulates the programmed grammar with appearance checking G.

Because the proof is rather technical, to help the reader, we denote with the label (nx) the rule (evolution or antiport/symport rule) present in equation n and corresponding to the alphabetical place x (for example: $(12b)$ means "the second rule in equation (12)").

The idea of the proof is simple: the protons store the labels of the rewriting rules that are applied (or skipped) and the antiports (the proton pumping rules) check that the rewriting rules are applied in a correct order (correct in the sense of the programmed grammar to simulate). Such idea is very similar to what has been made to show the universality of EC P systems (Theorem 2.2) but here we must take care of the fact that the "protons" cannot be created, but only "moved" and "used" in proton pumping rules.

The protons serve the role of remembering the label of the rule that has been previously applied (p_i) or skipped (q_i), appearance checking (r_j), p_0 is the starting point of the control sequence, c is used to sequentialize the application of the rules, and F checks that all non-terminals were rewritten at the end.

We also use objects associated with each terminal (T), associated with each non-terminal (N'), including also the rule application failure symbol (h), their "first" versions ($\overline{N'}$), their intermediate versions ($\tilde{N'}$), control sequence symbols ($b_i^{(j)}$ if the rule with label j was skipped and $d_i^{(j)}$ if the rule with label j was applied), appearance checking symbols (e_j, e_j'). Objects f_j are used to return the appearance checking protons, l, g, H are used to return the control sequence proton, t_j are delaying c for synchronization, K helps F and r_i to block the computation, s is used to end the simulation of derivation, switching to checking that no more non-terminals are in region 3, and # is the trap symbol.

In region 1 (the outer one) we simulate the application of the context-free rules of G, using simple evolution rules (4.1a), while the rules (4.7a-4.7f) enforce the "control sequence", i.e., transitions from an application of (or from the failure to apply) a rule of G to the next one. During the simula-

tion, the non-terminals to be rewritten are brought into region 1 by rules (4.8a,4.2a,4.5a) and the result is returned by rules (4.5b,4.6a), except the terminal symbols, ejected into the environment by (4.4). Rules (4.3a) remove the bars. The failure to apply a rule is simulated by (4.1b). The appearance checking is enforced by the rules (4.5c,4.2g,4.7i,4.8f). If appearance checking fails (4.8f was applied), then (4.8g) is applied and (4.3c) blocks the computation. If the appearance checking succeeds, then (4.7j) is applied to return the proton. Rules (4.7k,4.7l) make sure that the computation halts, and that when it does, no non-terminals are present in region 1, because otherwise the computation is blocked by (4.8i,4.8j,4.3c).

The following rules are auxiliary. (4.1c) blocks the computation if (4.1a) is not applied, and this could happen if there is no rule to rewrite non-terminal X. Rules (4.1d) provide a delay, so that if appearance checking is successful, the proton can be returned to region 1. Rules (4.1e,4.1f) block the computation if the control sequence is invalid, i.e., if the next rewriting rule is not in the success, or failure field of the previous rule applied, or skipped, respectively. (4.1g) helps (4.1c,4.1e,4.1f) to block. Rules (4.2b) organize the delay, so that the new non-terminal will be brought for rewriting after the previous one has been processed. Rules (4.2e,4.2f) organize the delay so that the proton, corresponding to the rule applied/skipped enters region 2 after the proton, corresponding to the previous rule applied/skipped, returns to region 3 using rules (4.2c,4.2d,4.7g,4.7h,4.6b,4.3b,4.8d,4.8e). Rules (4.8b,4.8c) replace the "current" label of rule by the "previous" label of rule. Finally, (4.8h) returns c to region 2, so that the next rule can be applied or skipped, or to finish the computation.

We will now proceed to a more detailed explanation of why this construction works; in what follows we will indicate with $E_n x$ the evolution rule present in the set R_n at (alphabetical) place x and with $\overline{E_n x_1 \cdots x_k}$ one of the rules present in the set of evolution rules R_n in position x_1 or x_2 or $\cdots x_k$ (the same idea is used for the antiport rules, $A_n x$, and symport rules, $S_n x$).

Now we pass to the description of valid computations. Applying or skipping a rule of G consists of 10 transitions of Π. Suppose that at some step the configuration of Π is

$$
\begin{array}{rcl}
 & & [_1[_2[_3 w X h \pi q \tau_3]_3 p c K \tau_2]_2 F s \rho r_i \tau_1]_1 v, \text{ where:} \\
alph(\pi) & \cup & \{p, q\} = \{p_i, q_i \mid i \in Lab(P)\}, \\
alph(\rho) & \cup & \{r_i\} = \{r_j \mid X_j \in N\}, \\
\tau_3 & \in & (\{t_{-2}\} \cup \{b_i^{(1)}, d_i^{(1)} \mid i \in Lab(P)\})^*, \\
\tau_2 & \in & \{H, f_1\}^*, \\
\tau_1 & \in & (\{l\} \cup \{e_j \mid X_j \in N\})^*,
\end{array}
$$

where the first two unions are disjoint. This corresponds to the simulation of derivation of G and having the sentential form in $Perm(wXv)$, where

$wX \in N^*$ and $v \in T^*$, and the last rule applied or skipped was j if $p = p_j$ or $p = q_j$, respectively.

Suppose that we would like to apply the rule $i : (X \to x, E_i, F_i)$ and $q = p_i$. Let $x \in Perm(yz)$, $y \in N^*, z \in T^*$.

Step	Region 3	Region 2	Region 1	Env	Rules
0-10	$wh\pi\tau_3$	$K\tau_2$	$Fs\rho r_i\tau_1$	v	
0	Xq	pc			A_3a
1	cq	pX			E_2a
2	cq	$p\tilde{X}t_5$			E_2b, S_2a
3	cq	pt_4	$\tilde{X}$		E_2b, E_1a
4	cq	pt_3	$\gamma(x)d_i^{(5)}$		$E_2b, S_2b, S_1, \overline{A_2ace}$
5	cq	$t_2\gamma(y)d_i^{(5)}$	p	z	E_2bd, S_3a
6	$cq\gamma(y)$	$t_1d_i^{(4)}lg$	p	z	$E_2bf, E_3a, \overline{A_2gh}, S_3b$
7	$cqyg$	$t_0d_i^{(3)}p$	l	z	E_2bf, E_1b
8	$cqyH$	$t_{-1}d_i^{(2)}p$	l	z	E_2bf, A_3de
9	$cpqy$	$t_{-2}d_i^{(1)}H$	l	z	$\overline{A_3bc}, A_3h$
10	$ypt_{-2}d_i^{(1)}$	qcH	l	z	like 0

So, starting from step 0, after 10 transitions, X will be replaced by y, v by vz, p by q, τ_3 by $\tau_3t_{-2}d_i^{(1)}$, τ_2 by τ_2H, τ_1 by τ_1l. The only variant not considered is if $p = q$, i.e. the last time the rule with label i was applied, and we choose to apply it again in case it is in its own success field. Note that p comes to region 3 before q leaves it, so in that case the computation will have no q in steps 0-8, no p in step 10, and the claim remains valid.

Again we start with $[_1[_2[_3wXh\pi q\tau_3]_3pcK\tau_2]_2Fs\rho r_i\tau_1]_1v$. Suppose that we would like to skip rule $i : (X_l \to x, E_i, F_i)$, and $q = q_j$.

Step	Region 3	Region 2	Region 1	Env	Rules
0-10	$wX\pi\tau_3$	$K\tau_2$	$Fs\rho\tau_1$	v	
0	hq	pc	r_i		A_3a
1	cq	ph	r_i		E_2a
2	cq	$p\tilde{h}t_5$	r_i		E_2b, S_2a
3	cq	pt_4	$\tilde{h}r_i$		E_2b, E_1b
4	cq	pt_3	$\overline{(h)}b_i^{(5)}e_jf_5r_i$		$E_2b, S_2bc, \overline{A_2bdf}, E_1d$
5	cq	$t_2\overline{(h)}b_i^{(5)}e_j$	pf_4r_i		E_2bcg, S_3a, E_1d
6	$cq\overline{(h)}$	$t_1b_i^{(4)}lg$	pr_if_3		$E_3a, E_2be, \overline{A_2gh}, A_2i, E_1d$
7	$cqhg$	$t_0b_i^{(3)}pr_i$			E_3b, E_2be, E_1d
8	$cqhH$	$t_{-1}b_i^{(2)}pr_i$			$\overline{A_3de}, E_2be, A_2j$
9	$cpqh$	$t_{-2}b_i^{(1)}f_1H$	r_i		A_3ch
10	$hpt_{-2}b_i^{(1)}$	$qcHf_1$	$r_ile'_j$		like 0

So, starting from step 0, after 10 transitions, p will be replaced by q, τ_3 by $\tau_3 t_{-2} b_i^{(1)}$, τ_2 by $\tau_2 H f_1$, τ_1 by $\tau_1 l e'_j$. Terminating computation (no non-terminals are present in region 3, $w = \varepsilon$):

Step	Region 3	Region 2	Region 1	Env	Rules
0-2	$hqX\pi\tau_3$	$pK\tau_2$	$\rho\tau_1 r_i$	v	
0		c	Fs		$A_2 k$
1		sF	c		$A_2 l$
2		F	cs		NONE

Consequently, $\Psi_T(L(G)) = Ps(\Pi)$ □

4.2 Time-Free P Systems

Time-free P systems are, as described earlier, systems where the results of the computations do not depend on the time of execution of the rules. Time-free P systems still work in the standard maximally parallel way but there is no assumption on the time of completion of a started rule.

The definition of time-free P systems we give here is for a general class of systems with signal-promoters and bi-stable catalysts.

Definition 4.2 *A P system* Π *of degree* $m \geq 1$, *with signal-promoters and bi-stable catalysts, is a construct*

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

where:

- O *is the alphabet of* Π*; its elements are called objects;*
- $C \subseteq O$ *is the set of bi-stable catalysts;*
- $D \subseteq O$ *is the set of signal-promoters;*
- μ *is a membrane structure consisting of* m *membranes injectively labeled with* $1, 2, \cdots, m$*;*
- w_i, $1 \leq i \leq m$, *specifies the multiset of objects present in the corresponding region* i *at the beginning of the computation;*
- R_i, $1 \leq i \leq m$, *are finite sets of evolution rules over* O *associated with regions* $1, 2, \ldots, m$ *of* μ*; there are rules of two types: (1) non-cooperative, that are of the form* $a \to v$*; (2) catalytic (using bi-stable catalysts) that are of the forms* $ca \to cv, ca \to \overline{c}v, \overline{c}a \to \overline{c}v$, *and* $\overline{c}a \to cv$*; a is an object from* $O - (C \cup D)$, *v is a string over* $\{a_{here}, a_{out} \mid a \in O - (C \cup D)\} \cup \{a_{in_j} \mid a \in O - (C \cup D), 1 \leq j \leq m\}$, *and* $c \in C$*;*

- R_i^s, $1 \leq i \leq m$, *are finite sets of signaling rules over* D *associated with regions* $1, 2, \ldots, m$ *of* μ*; the signaling rules are of the form* $a \to v|_z$ *or* $ca \to cv|_z$, *where* a *is an object from* $O - (C \cup D)$, v *is a string over* $\{a_{here}, a_{out} \mid a \in O-(C \cup D)\} \cup \{a_{in_j} \mid a \in O-(C \cup D), 1 \leq j \leq m\}$, z *is a string representing a subset of* $\{(p, here), (p, out) \mid p \in D\} \cup \{(p, in_j) \mid p \in D, 1 \leq j \leq m\}$, *and* $c \in C$*;*

- $i_0 \in \{0, 1, \cdots, m\}$ *is the label of the* output region*; if* $i_0 = 0$, *then the output region is the environment.*

Given a computable mapping

$$e : R_1 \cup \cdots \cup R_m \cup R_1^s \cup \cdots \cup R_m^s \longrightarrow \mathbb{N}$$

and a system Π as defined above, it is possible to construct a *timed P system* as

$$\Pi(e) = (O, C, D, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, R_1^s, \ldots, R_m^s, i_0, e)$$

working in the following way.

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

The m-tuple $(w_1, \cdots, w_m)$ and the structure μ represent the *initial configuration* of the P system. A timed P system starts the *computation* from the initial configuration.

At each time, all the *active rules* that can be applied (started) in each region, must be applied (started).

Evolution rules are always active. Signaling rules are promoted by the signal-promoters specified in the string z; a signaling rule in region i is active if the signal-promoters specified by z are present in region i.

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

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

Like for evolution rules where the produced objects are moved according to their target indications, in the case of a signaling rule $u \to v|_z$, also the signal-promoters specified by z are moved to the regions according to their target indications (notice that signal-promoters cannot be created/rewritten but only moved during the computation). Moreover, in every region, the signal-promoters are present in the *set sense*, i.e., in each region there cannot be more than one copy of the same signal-promoter.

When a rule r *(either evolution or signaling) is started at time* j, *then its execution terminates at time* $j + e(r)$ *(from this time the objects produced as well as the signal-promoters moved by the rule can be used).*

Because the rules have a certain time of execution then, at each time, in the regions of the system there are (possibly) rules in execution and rules not in execution.

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

The computation halts (and then it is considered *successful*) when no rule can be applied in any region and there is no rule in execution (the last configuration reached is called *halting*).

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

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

Definition 4.3 *A P system* $\Pi = (O, C, D, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, R_1^s, \ldots, R_m^s, i_0)$ *is* time-free *if and only if every timed system in the set*

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

where $R = R_1 \cup \cdots \cup R_m \cup R_1^s \cup \cdots \cup R_m^s$, *produces the same set of vectors of numbers.*

We use the notation

$$fPsP_m(\alpha, j), \alpha \in \{ncoo, coo\} \cup \{2cat_k, cat_k \mid k \geq 0\},$$

to denote the family of sets of vectors of numbers generated by *time-free* P systems with at most m membranes, at most j signal-promoters, evolution rules and signaling rules that can be non-cooperative ($ncoo$), or catalytic ($cat_k/2cat_k$), using at most k catalysts/k bi-stable catalysts (as usual, $*$ is used if the corresponding number of membranes, signal-promoters or catalysts/bi-stable catalysts is not bounded).

A remark concerning signal-promoters is worth mentioning.

Using signal-promoters, two situations can cause conflicts.

If (at least one of) the signal-promoters that activate a rule r is moved out from the region where r is present before rule r is terminated then the execution of r cannot continue (the rule is not active anymore) and the computation is considered not successful.

On the other hand, if two or more signaling rules, promoted by the same signal-promoters but with different targets terminate their execution at the same time, then, also in this case, the computation is considered not

successful (a conflict over the destination of signals is present). However, in the Thesis these conflicts will never appear.

In what follows, to simplify the notation, in the definition of a P system that does not use signal-promoters we will not write the set of signal-promoters and of signaling-rules (instead of adding them as empty sets).

4.2.1 Preliminary Results

Given a timed P system using only non-cooperative evolution rules is possible to construct an equivalent P system having only one object in one region in the initial configuration (this unique object can generate, possibly in several consecutive steps, the initial configuration of the original system).

Then each computation of a timed P system using only non-cooperative evolution rules can be described by a tree where each node represents an object indexed with the label of the region where the object is present (following the idea of derivation tree, defined for context-free grammars). The tree stores on (part of) the yield the result of the represented computation; the length of the edges encodes the time of execution of the corresponding rules. Because the yield of the tree does not depend on the length of the edges then it is clear that:

Theorem 4.2 *Every P system using only non-cooperative evolution rules is time-free.*

We recall that $PsOP_m(ncoo, tar)$ denotes the family of sets of vectors of numbers generated by symbol-objects P systems of degree at most m, using non-cooperative evolution rules and target communications of the type $\{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$. We have $PsOP_m(ncoo, tar) = PsCF$, for all $m \geq 1$, where $PsCF$ denotes the family of Parikh images of context-free languages.

Therefore, as a corollary of Theorem 4.2, we obtain the following result:

Corollary 4.1 *$fPsP_m(ncoo, 0) = PsCF$ for all $m \geq 1$.*

Theorem 4.2 does not hold for P systems using catalytic evolution rules (even with only one catalyst) as it is shown in the following theorem.

Theorem 4.3 *There exists a P system using only non-cooperative and catalytic evolution rules that is not time-free.*

Proof. Consider the following P system with one catalyst

$$\Pi = (O, C, \mu, w_1, w_2, R_1, R_2, i_0),$$

where:

$$
\begin{aligned}
O &= \{B', B'', c, X, D, b, a\},\\
C &= \{c\},\\
\mu &= [_1[_2]_2]_1,\\
w_1 &= B'B''c,\\
w_2 &= \lambda,\\
R_1 &= \{r_1 : B' \to b_{in_2}X,\ r_2 : cB'' \to c,\ r_3 : X \to D,\\
&\quad\ r_4 : cX \to c,\ r_5 : D \to a_{in_2}\},\\
R_2 &= \emptyset,\\
i_0 &= 2.
\end{aligned}
$$

The system Π is not time-free.

In fact it generates two different outputs by using two different time-mappings e' and e''.

First, consider the time-mapping e' defined in the following way:
$e'(r_1) = 1$,
$e'(r_2) = 2$,
$e'(r_i) = k'$, for some $k' \in N$, for $i = 3, 4, 5$.

Consider now the time-mapping e'' defined in the following way:
$e''(r_1) = 2$,
$e''(r_2) = 1$,
$e''(r_i) = k''$, for some $k'' \in N$, for $i = 3, 4, 5$.

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

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

In fact, rules r_1 and r_2 are started in parallel as in the previous case, but because of the time-mapping e'', rule r_1 ends after rule r_2. Then, after two steps, X is produced and b is sent to the output region; in the next step X can be rewritten in two possible ways. In the first way the catalyst c can be used to apply rule r_4 and then the computation halts producing as output the vector $(1, 0)$. In the second case the rule r_3 is used, and then rule r_5,

that produces and sends the object a to the output region. Therefore, in this case the output of the system is $\{(1,1),(1,0)\}$.

Notice that for any time-mapping e, $Ps(\Pi(e)) \neq \emptyset$. □

While by using time-free systems with only non-cooperative evolution rules one can get only "context-free" power (Corollary 4.1) it is easy to construct more powerful time-free P systems by using non-cooperative signaling rules, as shown in the following example.

Example 4.1 *We consider the system*

$$\Pi = (O, C, D, \mu, w_1, w_2, R_1, R_2, R_1^s, R_2^s, i_0),$$

where:

$$\begin{array}{ll} O = \{a, b, p\}, & R_1 = \emptyset, \\ C = \emptyset, & R_2 = \emptyset, \\ D = \{p\}, & R_1^s = \{b \rightarrow b|_{(p,in)},\ b \rightarrow b|_{(p,out)}\}, \\ \mu = [_1\ [_2\ ]_2\ ]_1, & R_2^s = \{a \rightarrow aa|_{(p,out)}\}, \\ w_1 = bp, & i_0 = 2, \\ w_2 = a. & \end{array}$$

The rule $a \rightarrow aa$ is activated by the signal-promoter p which is present at the beginning of the computation in region 1. The rule is applied an arbitrary number of times in the maximally parallel manner. Every time that the signal-promoter p is sent to region 1, one of the rules present in that region is applied. If rule $b \rightarrow b|_{(p,in)}$ is applied, then the process can be iterated. If rule $b \rightarrow b|_{(p,out)}$ is applied, then the signal-promoter is sent to the environment and the computation halts with a number of objects in region 2 that is a power of 2. It is trivial to see that the system generates the Parikh image of $\{a^{2^n} \mid n \geq 0\}$ independently from the execution times of the rules and, therefore, the system Π is time-free.

Using signals, as presented in Example 4.1, it is possible to synchronize in an easy way the rules of the system and to get time-free systems with enough computational power.

In what follows we recall a series of results for time-free P systems using signal-promoters.

Initially, we consider time-free P systems using only non-cooperative evolution and signaling rules. As soon as we use bi-stable catalysts, then we obtain universality even without using signaling rules (in this case the synchronization is obtained by making use of the short term memory of the bi-stable catalysts). On the other hand, the short memory of bi-stable catalysts can be simulated by signal-promoters and, in this way, we get an universality proof for time-free P systems using catalytic evolution and signaling rules.

4.2.2 Using Non-Cooperative Evolution and Signaling Rules

Using only non-cooperative evolution rules, non-cooperative signaling rules, and two regions it is possible to generate at least the family of Parikh images of languages generated by Indian parallel grammars.

Theorem 4.4 $PsIP \subseteq fPsP_2(ncoo, *)$.

Proof. Given an Indian parallel grammar $G = (N, T, S, P)$, we suppose that to each rule $r \in P$ an unique label $l(r)$ has been associated. The set of the labels associated to the rules in P is denoted by $Lab(P) = \{r_1, r_2, \cdots, r_k\}$.

We construct the following P system simulating G:

$$\Pi = (O, C, D, \mu, w_1, w_2, R_1, R_2, R_1^s, R_2^s, i_0),$$

where:

$$\begin{aligned}
O &= N \cup T \cup D \cup \{Q', T, T', Z, \#\} \cup \{T_r \mid r \in Lab(P)\},\\
C &= \emptyset,\\
D &= Lab(P) \cup \{s', s\},\\
\mu &= [_1\, [_2\,]_2\,]_1,\\
w_1 &= ZTsr_1r_2 \cdots r_k,\\
w_2 &= Ss'Q,\\
R_1 &= \{Z \to Z\},\\
R_2 &= \{\# \to \#\},\\
R_1^s &= \{Z \to \lambda|_{(s',here)},\ T \to T'|_{(s,in_2)}\}\\
&\cup\ \{T \to T_r|_{(r,in_2)},\ T_r \to T|_{(r,here)} \mid r \in Lab(P)\},\\
R_2^s &= \{X \to w|_{(r,out)} \mid X \to w \in P \text{ and } r = l(X \to w)\}\\
&\cup\ \{X \to \#|_{(s,here)} \mid X \in N\} \cup \{Q' \to \lambda|_{(s',out)},\ Q \to Q'|_{(s,here)}\},\\
i_0 &= 2.
\end{aligned}$$

The system works in the following way. In region 2 (the output region) the rules of grammar G are simulated. In region 1 the application of one of the rules in the set $\{T \to T_r|_{(r,in_2)} \mid r \in Lab(P)\}$ selects the rule of the grammar G to be activated (and, if possible, applied) in region 2, by sending the corresponding signal-promoter $r \in Lab(P)$. When an activated rule $X \to w|_{(r,out)}$ is applied in region 2, the signal-promoter r is sent back to region 1. Therefore the simulation of another rule can be made.

To stop the computation it is necessary to halt the rule $Z \to Z$ present in region 1. To do this, the rule $T \to T'|_{(s,in_2)}$ must be used in region 1. When $T \to T'|_{(s,in_2)}$ is executed, the signal-promoter s arrives in region 2, checking that all symbol-objects present are terminals (by activating the rules $X \to$

$\#|_{(s,here)}$, $X \in N$). Moreover, the rules $Q' \to \lambda|_{(s',out)}$, $Q \to Q'|_{(s,here)}$ will send to region 1 the signal-promoter s' necessary to delete Z and then to stop the computation. Therefore, the computation halts when $Z \to \lambda|_{(s',here)}$ is applied if and only if all terminals have been obtained in the output region. This means that the system Π generates exactly the Parikh image of $L(G)$. From the description above it is clear that the system is time-free because it generates the same output independently from the execution times of the rules. □

4.2.3 Using (Bi-stable) Catalysts and Signal-Promoters

In time-free P systems the synchronization can be obtained by using the short term memory of bi-stable catalysts. Actually, bi-stable catalytic evolution rules are enough to construct time-free systems that are computationally complete, even without using signaling rules. In fact, the following theorem is true.

Theorem 4.5 $fPsP_1(2cat_*, 0) = PsRE$.

Proof. We only show that $PsRE \subseteq fPsOP_1(2cat_*, 0)$ (the reverse inclusion is straightforward) by simulation of matrix grammars with appearance checking.

Given a matrix grammar $G = (N, T, S, M, F)$ in the Z-binary normal form (in the standard notation presented in Section 1.1, with k matrices of type 2 and $n - k$ matrices of type 3; all matrices are supposed labeled in a one-to-one manner with $m_1, \ldots, m_n$); we construct a time-free P system Π, using bi-stable catalytic evolution rules, generating exactly the Parikh image of $L(G)$. Consider the morphism h defined by $h(a) = a_{out}, a \in T$ and $h(A) = A, A \in N_2$.

We construct the following system:

$$\Pi = (O, C, \mu, w_1, R_1, i_0),$$

where:

$$\begin{aligned}
O &= N \cup C \cup T \cup \{E, g, \#\} \cup \{X', X'' \mid X \in N_1\},\\
C &= \{c_i, \overline{c_i} \mid 0 \leq i \leq n\},\\
\mu &= [_1\,]_1,\\
w_1 &= X_{init} A_{init} E c_0 c_1 \ldots c_n,\\
R_1 &= R_1^1 \cup R_1^2 \cup \{\# \to \#\}, where:\\
R_1^1 &= \{c_i X \to \overline{c_i} Y',\ \overline{c_i} E \to c_i \#,\ \overline{c_i} A \to c_i h(x) g \mid\\
&\quad m_i : (X \to Y, A \to x), 1 \leq i \leq k\}\\
&\cup\ \{Y' \to Y'' \mid Y \in N_1\} \cup \{c_0 g \to \overline{c_0}\} \cup \{\overline{c_0} Y'' \to c_0 Y \mid Y \in N_1\},
\end{aligned}$$

$$
\begin{aligned}
R_1^2 &= \{c_iX \to \overline{c_i}Y',\ \overline{c_i}A \to c_i\#,\ \overline{c_i}Y'' \to c_iY \mid \\
&\quad m_i : (X \to Y,\ A \to \#), k+1 \leq i \leq n\}, \\
i_0 &= 0.
\end{aligned}
$$

The system Π simulates the application of the matrices of the grammar G in region 1 and collects the result in the environment.

The rules in R_1^1 manage the simulation of a matrix of type 2. To simulate a matrix $m_i : (X \to Y, A \to x)$ the following steps need to be done: at certain step j the rule $c_iX \to \overline{c_i}Y'$ is executed and then c_i is changed into $\overline{c_i}$ and the object Y', for some $Y \in N_1$, is produced (the presence of only one object $X \in N_1$ guarantees that only one rule of this kind is applied in the step j). We suppose the rule $c_iX \to \overline{c_i}Y'$ is terminated at step $j + j'$. Then, at step $j + j' + 1$ the rule $Y' \to Y''$ and one of the two rules $\overline{c_i}E \to c_i\#$, $\overline{c_i}A \to c_ih(x)g$, are started, in parallel. If there is no object A, in region 1 then the rule $\overline{c_i}A \to c_ih(x)g$ cannot be applied and then the rule $\overline{c_i}E \to c_i\#$ is applied, leading to a non-halting computation. We suppose that at least one occurrence of object A is present in region 1; in this case $\overline{c_i}A \to c_ih(x)g$ and $Y' \to Y''$ are started, in parallel, at step $j+j'+1$. When the rule $\overline{c_i}A \to c_ih(x)g$ is terminated, the object g is produced and then the rule $c_0g \to \overline{c_0}$ can be started. On the other hand, when rule $Y' \to Y''$ is terminated the object Y'' can be used in rule $\overline{c_0}Y'' \to c_0Y$. Therefore this last rule is started only when both rules, $c_0g \to \overline{c_0}$ and $Y' \to Y''$ are terminated (and this is independent from the time of execution of the rules involved). Once the rule $\overline{c_0}Y'' \to c_0Y$ is applied, the object Y is produced and then the process can be iterated by choosing a new matrix to be simulated. Therefore the matrix $m_i : (X \to Y, A \to x)$ is correctly simulated independently from the time of execution of the rules involved in the simulation.

The simulation of a matrix of type 3, used in appearance checking mode, is managed by the rules present in R_1^2.

To simulate a matrix $m_i : (X \to Y, A \to \#)$ the following steps are done: first the rule $c_iX \to \overline{c_i}Y'$ is executed. In this way, c_i is changed to $\overline{c_i}$ and Y' is produced, for some $Y \in N_1$. If this rule is terminated at step $j + j''$ then, in the following step $j + j'' + 1$ the rules $Y' \to Y''$ and $\overline{c_i}A \to c_i\#$ are started in parallel; if no occurrence of the object A is present, then the rule is skipped (if the rule is not skipped, then the computation will never halt). If the rule $\overline{c_i}A \to c_i\#$ is not applied, then in the step following the end of rule $Y' \to Y''$, the rule $\overline{c_i}Y'' \to c_iY$ is executed, the object Y is produced and the process can be iterated by choosing a new matrix to be simulated. Thus, also the matrix $m_i : (X \to Y, A \to \#)$ is correctly simulated independently from the time of execution of the rules involved in the simulation.

Each occurrence of a terminal object is sent to the environment where the result of the computation is collected, and the computation stops when the object Z is produced, provided that no occurrence of # is present. Therefore,

the system generates exactly the Parikh image of the language generated by the matrix grammar G, independently from the times of execution of the rules (the constructed system is time-free). □

In other words, Theorem 4.5 states that, for any P system using bi-stable catalytic evolution rules, there exists an equivalent P system (still with bi-stable catalysts) that is time-free. Therefore, an immediate question is the following one: given an arbitrary P system Π using evolution rules with bi-stable catalysts, is it possible to decide whether Π is time-free? The next theorem answers negatively to this question.

Theorem 4.6 *Given an arbitrary P system Π using only bi-stable catalytic and non-cooperative evolution rules, it is undecidable whether Π is time-free.*

Proof. Given an arbitrary language $L \in RE$ over an alphabet V, it is possible to construct a time-free P system $\Pi = (O, C, \mu = [_1\,]_1, w_1, R_1, i_0)$, as in the proof given for Theorem 4.5, and generating the Parikh image of L.

We slightly modify the constructed P system Π adding an external membrane labeled 0 where the output of Π is collected.

The obtained P system $\Pi^i = (O, C, [_0\,[_1\,]_1\,]_0, \lambda, w_1, \emptyset, R_1^i, 0)$ is still time-free and generates the same output of Π but in region 0 (the evolution rules are adjusted).

Formally, for any pair of time-mappings e', e'' we have $Ps(\Pi(e')) = Ps(\Pi^i(e'')) = Q$, where Q is a certain set of vectors of natural numbers.

Consider also the non time-free P system constructed in Theorem 4.3 and denote its components as follows $\Pi^{ii} = (O^{ii}, C^{ii}, \mu^{ii} = [_1\,[_2\,]_2\,]_1, w_1^{ii} = B'B''c, w_2^{ii} = \lambda, R_1^{ii}, R_2^{ii} = \emptyset, 2)$.

We construct the P system $\Pi^{iii} = (O^{ii}, C^{ii}, [_2\,[_3\,]_3\,]_2, c, \lambda, R_1^{ii}, \emptyset, 3)$ where Π^{iii} is obtained from Π^{ii} by re-labeling membranes 1 and 2 with 2 and 3, respectively, and by using only c as the initial object present in region 2.

We insert in region 0 of Π^i the P system Π^{iii} (without loss of generality we suppose that $O^{ii} \cap O = \emptyset$).

In this way we get the P system

$$\Pi^{iv} = (O^{iv}, C^{iv}, \mu^{iv}, w_0^{iv}, w_1^{iv}, w_2^{iv}, w_3^{iv}, R_0^{iv}, R_1^{iv}, R_2^{iv}, R_3^{iv}, i_0)$$

with:

$$\begin{aligned} O^{iv} &= O^{ii} \cup O, \\ C^{iv} &= C^{ii} \cup C, \\ \mu^{iv} &= [_0\,[_1\,]_1\,[_2\,[_3\,]_3\,]_2\,]_0, \\ w_0^{iv} &= \lambda, \\ w_1^{iv} &= w_1, \end{aligned}$$

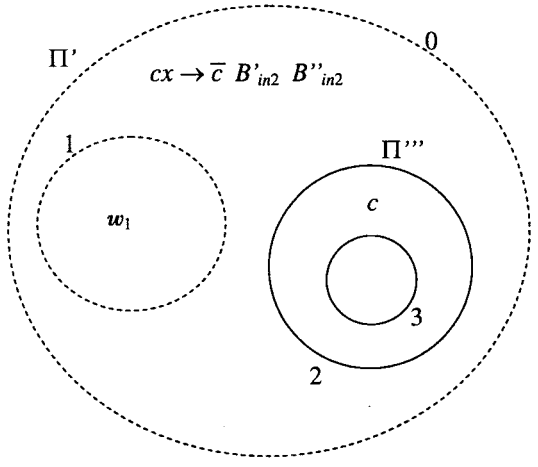

Figure 4.1: The P system Π^{iv}; dotted lines represent Π^{i}; continuous lines represent Π^{iii}. Region 0 is the output region of Π^{i}. Region 3 is the output region of Π^{iii} and of the entire Π^{iv}.

$$
\begin{aligned}
w_2^{iv} &= c, \\
w_3^{iv} &= \lambda, \\
R_0^{iv} &= \{r_a : cx \to \overline{c}B'_{in_2}B''_{in_2} \mid x \in O\}, \\
R_1^{iv} &= R_1^{i}, \\
R_2^{iv} &= R_2^{iii}, \\
R_3^{iv} &= \emptyset, \\
i_0^{iv} &= 3.
\end{aligned}
$$

A graphical representation of the system Π^{iv} can be found in Figure 4.1.

Notice that we do not know if the constructed Π^{iv} is time-free. The added rule r_a is used to introduce in region 2 of Π^{iv} the objects B' and B'' corresponding to the initial configuration of Π^{ii} (c is present from the beginning in region 2). We will denote with $\overline{0}$ a vector of natural numbers with all the components fixed to zero.

Now, suppose to have an algorithm A that takes as input an arbitrary P system with only bi-stable catalytic and non-cooperative evolution rules and decides whether it is time-free.

When this algorithm A is applied to the P system Π^{iv}, two cases are possible:

- The algorithm outputs *yes*; then Π^{iv} is time-free and this means that, for any time-mapping e, $Ps(\Pi^{iv}(e))$ is the same set of vectors of natural numbers. If there exists a time-mapping e such that $x \in Ps(\Pi^{i}(e))$, and $x \neq \overline{0}$, then, because Π^{i} is time-free, $x \in Ps(\Pi^{i}(e))$, for any mapping e. In particular, this is true when we associate to the rules

of Π^i (the ones present in region 0 and 1 of Π^{iv}) the execution times defined by the time-mappings e', e'' used in Theorem 4.3. In this case we have $x \in Ps(\Pi^i(e')), x \in Ps(\Pi^i(e''))$ and the rule r_a can be used; therefore the system Π^{iii} works like the system Π^{ii} (non time-free) and the entire system Π^{iv} cannot be time-free. Then there must be no vector $x \neq \overline{0}$ in $Ps(\Pi^i(e)) = Ps(\Pi^i)$, for any time-mapping e; therefore there is no vector $x \neq \overline{0}$ in $Ps(\Pi)$.

- The algorithm outputs *no*; then, there must be a vector $x \neq \overline{0}$ in $Ps(\Pi^i(e'))$ for a certain time-mapping e' (otherwise the rule r_a cannot be used, Π^{iii} is not used and the output region of Π^{iv}, 3, remains always empty). Because Π^i is time-free, we must have $x \in Ps(\Pi^i(e)) = Ps(\Pi^i)$, for any mapping e; therefore the vector $x \neq \overline{0}$ is in $Ps(\Pi)$.

In general, it is undecidable whether $Ps(\Pi)$ contains any vector $x \neq \overline{0}$ (because of the constructive universality proved in Theorem 4.5); therefore the algorithm A cannot exist. □

On the other hand, as we can see in the next theorem, a time-free P system with only bi-stable catalytic evolution rules and a membrane structure of the form $\mu = [_1\, [_2 \cdots [_m\,]_m \cdots]_2\,]_1$ can be simulated by a time-free P system using catalytic evolution rules, (at most) $m + 1$ membranes, and signal-promoters.

Theorem 4.7 $fPsP_2(cat_*, *) = PsRE$.

Proof. To prove the theorem we show how a time-free P system

$$\begin{aligned}\Pi_1 \;=\; (&O_1, C_1, D_1 = \emptyset, [_1\, [_2 \cdots [_m\,]_m \cdots]_2\,]_1, w_{1,1}, w_{1,2}, \cdots, w_{1,m},\\ &R_{1,1}, R_{1,2}, \cdots, R_{1,m}, R^s_{1,1} = \emptyset, R^s_{1,2} = \emptyset, \cdots, R^s_{1,m} = \emptyset, i_0),\end{aligned}$$

using bi-stable catalytic evolution rules, m membranes can be simulated by a time-free P system

$$\begin{aligned}\Pi_2 \;=\; (&O_2, C_2, D_2, [_0\, [_1\, [_2 \cdots [_m\,]_m \cdots]_2\,]_1\,]_0, w_{2,0}, w_{2,1}, w_{2,2}, \cdots,\\ &w_{2,m}, R_{2,0}, R_{2,1}, R_{2,2}, \cdots, R_{2,m}, R^s_{2,0}, R^s_{2,1}, R^s_{2,2}, \cdots, R^s_{2,m}, i_0),\end{aligned}$$

using catalytic evolution rules, $m + 1$ membranes and signal-promoters.

Without loss of generality, we suppose that each occurrence of the bi-stable catalysts present in the regions of Π_1 is named differently. We suppose that the set of bi-stable catalysts used in Π_1 is $C_1 = \{c_1, c_2, \cdots, c_h\}$.

We construct the system Π_2 in the following way.

Suppose $j \in \{1, \cdots, h\}$. For each rule $r^{1,1}_{i\langle a \to w\rangle} : \overline{c_j} a \to c_j w$ that is present in $R_{1,i}$, $1 \leq i \leq m$, we add to $R^s_{2,i}$ the signaling rule $r^{2,1}_{i\langle a \to w\rangle}$:

$c_j a \to c_j w|_{(\overline{p_j},out)}$, and to $R^s_{2,i-1}$ the signaling rules $X_j \to X''_j|_{(\overline{p_j},here)}$ and $X''_j \to X'''_j|_{(p_j,in)}$.

In a similar way, for each rule $r^{1,2}_{i\langle a\to w\rangle} : c_j a \to \overline{c_j} w$ that is present in $R_{1,i}$, $1 \leq i \leq m$, we add the signaling rule $r^{2,2}_{i\langle a\to w\rangle} : c_j a \to c_j w|_{(p_j,out)}$ to $R^s_{2,i}$, and to $R^s_{2,i-1}$ the signaling rules $X'''_j \to X'_j|_{(p_j,here)}$ and $X'_j \to X_j|_{(\overline{p_j},in)}$.

For each rule $r^{1,3}_{i\langle a\to w\rangle} : \overline{c_j} a \to \overline{c_j} w$ that is present in $R_{1,i}$, $1 \leq i \leq m$, we add the signaling rule $r^{2,3}_{i\langle a\to w\rangle} : c_j a \to c_j w|_{(\overline{p_j},here)}$ to $R^s_{2,i}$; for each rule $r^{1,4}_{i\langle a\to w\rangle} : c_j a \to c_j w$ present in $R_{1,i}$, $1 \leq i \leq m$, we add the rule $r^{2,4}_{i\langle a\to w\rangle} : c_j a \to c_j w|_{(p_j,here)}$ to $R^s_{2,i}$.

For each non-cooperative rule $r^{1,0}_{i\langle X\to w\rangle} : X \to w$ that is present in $R_{1,i}$, $1 \leq i \leq m$, we add the non-cooperative rule $r^{2,0}_{i\langle X\to w\rangle} : X \to w$ to the set $R_{2,i}$.

We take the alphabet $O_2 = O_1 \cup \{X_i, X''_i, X'_i, X'''_i \mid 1 \leq i \leq h\}$ and the set of signal-promoters as $D_2 = \{\overline{p_i}, p_i \mid 1 \leq i \leq h\}$. The set of catalysts C_2 is exactly the set C_1 except the fact that objects in C_2 are used as catalysts and not as bi-stable catalysts.

Finally we construct $w_{2,l}$, for $1 \leq l \leq m$, in the following way (by definition, $w_{2,0} = \lambda$). Add to $w_{2,l}$, $1 \leq l \leq m$, all the objects $x \in O_1 - C_1$ present in $w_{1,l}$. If $w_{1,l}$ contains $c_{j'}$, for some $j' \in \{1, 2, \cdots, h\}$, then add $p_{j'}$ and $c_{j'}$ to $w_{2,l}$; if $w_{1,l}$ contains $\overline{c_{j'}}$ for some $j' \in \{1, 2, \cdots, h\}$, then add $\overline{p_{j'}}$ and $c_{j'}$ to $w_{2,l}$; if $w_{1,l+1}$ contains $\overline{c_{j'}}$, for some $j' \in \{1, 2, \cdots, h\}$, then add $X_{j'}$ to $w_{2,l}$; if $w_{1,l+1}$ contains $c_{j'}$, for some $j' \in \{1, 2, \cdots, h\}$, then add $X'''_{j'}$ to $w_{2,l}$.

The main idea of the proof is that the "change of state" of a bi-stable catalyst present in region i of Π_1 is simulated by an exchange of signal-promoters between region i and the surrounding region $i-1$ in Π_2.

For instance (in all other cases the situation is similar), the execution of the rule $r^{1,1}_{i\langle a\to w\rangle} : \overline{c_j} a \to c_j w$ present in region i of Π_1, for some $j \in \{1, 2, \cdots, h\}$, is simulated in Π_2 in the following way. First, rule $c_j a \to c_j w|_{(\overline{p_j},out)}$ is executed in region i; at the end of its execution the signal-promoter $\overline{p_j}$ is sent out to the surrounding region $i-1$. There, the two rules $X_j \to X''_j|_{(\overline{p_j},here)}$ and $X''_j \to X'''_j|_{(p_j,in)}$ are executed sequentially and they send inside region i the signal-promoter p_j. In region i of Π_2 the presence of signal-promoter p_j activates now all (and only) the rules catalyzed by c_j; in this way the simulation of the execution of rule $r^{1,1}_{i\langle a\to w\rangle} : \overline{c_j} a \to c_j w$ in Π_1 has been completely simulated (the obtained object X'''_j in region $i-1$ stores the information that c_j is in state c_j).

The execution of rule $r^{1,2}_{i\langle a\to w\rangle} : c_j a \to \overline{c_j} w$ present in region i of Π_1 is simulated in Π_2 in the following way. First the rule $r^{2,2}_{i\langle a\to w\rangle} : c_j a \to c_j w|_{(p_j,out)}$ in $R^s_{2,i}$ of Π_2 is executed; at the end of its execution the signal-

promoter p_j is sent out to the surrounding region $i-1$. There, both rules $X_j''' \to X_j'|_{(p_j,here)}$ and $X_j' \to X_j|_{(\overline{p_j},in)}$ are executed and the signal-promoter $\overline{p_j}$ is sent to region i. In region i of Π_2 the presence of the signal-promoter $\overline{p_j}$ activates now all (and only) the rules catalyzed by $\overline{c_j}$; in this way the simulation of the execution of rule $r^{1,2}_{i\langle a\to w\rangle} : c_j a \to \overline{c_j} w$ has been completely simulated (the object X_j obtained in region $i-1$ stores the information that c_j is in state $\overline{c_j}$).

From the way Π_2 has been constructed and because Π_1 is time-free, then also Π_2 is time-free; moreover, they generate the same set of vectors of numbers. Therefore, because of Theorem 4.5, the statement is true. □

4.2.4 Synchronization by Transport Mechanisms: The Case of Symport/Antiport Rules

Synchronization in time-free P systems can be also obtained by using transport mechanisms. We investigate systems where the synchronization during the computation is obtained by only moving objects across the membranes of the system.

In particular, in this section we recall time-free P systems using symport/antiport rules.

Actually, we present the definition of a P system with symport/antiport rules in an *accepting variant*. The definition is slightly different from the one given in Section 1.3.2. In fact, the system contains a finite and infinite environment and an input region where the vector to accept is initially added.

Definition 4.4 *An accepting P system with symport/antiport rules of degree $m \geq 1$ is a construct*

$$\Pi = (O, \mu, w_1, \cdots, w_m, E_{inf}, E_{fin}, R_1, \cdots, R_m, i_{inp}),$$

where:

- *O is the alphabet of Π; its elements are called* objects*;*
- *μ is a membrane structure consisting of m membranes injectively labeled with $1, 2, \cdots, m$;*
- *w_i, $1 \leq i \leq m$, specifies the multisets of objects present in the corresponding region i at the beginning of the computation;*
- *$E_{inf} \subseteq O$ is the set of objects which are supposed to appear in arbitrary many copies in the environment;*
- *E_{fin} specifies the multiset of objects present, at the beginning of the computation, in a finite number of copies, in the environment;*

- $R_1, \cdots, R_m$ *are finite sets of* symport/antiport rules *over* O *associated with the membranes* $1, 2 \cdots, m$ *of* μ*; a* symport rule *is of the form* (x, in) *or* (x, out)*, while an* antiport rule *is of the form* $(x, in; y, out)$*, where* x, y *are strings representing multisets of objects of* O*;*

- $i_{inp} \in \{1, 2, \cdots, m\}$ *is the label of the* input region.

In an accepting P system with symport/antiport rules, the *initial configuration* is described by the m-tuples of multisets of objects present in the m regions of the system and by the multiset describing the objects present in the environment in a finite number of copies.

The objects of E_{inf} are supposed to be present in the environment in an arbitrarily number of copies and their number remains arbitrarily irrespective of how many copies are introduced in the system during the computation.

Given a computable mapping

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

and the system Π as defined above, it is possible to construct a *timed accepting P system with symport/antiport rules* $\Pi(e)$ as $(O, \mu, w_1, \cdots, w_m, E_{inf}, E_{fin}, R_1, \cdots, R_m, i_{inp}, e)$ working in the following way.

As above, we suppose to have an external clock (that does not have any influence on the system) that starts at time 0 and ticks from each time j to the next one $j + 1$.

In the accepting mode, a timed system Π starts computing in the configuration obtained by adding a multiset x to w_{inp}, in the initial configuration.

The computation is then defined like for P systems with symport/antiport rules (Section 1.3.2) except that *when a rule* r *(either symport or antiport) is started at time* j*, then its execution terminates at time* $j + e(r)$ *(therefore, from this time, the objects moved can be used).* If two rules start at the same time, then possible conflicts for using the occurrences of objects are solved assigning the occurrences in a non-deterministic way.

The computation stops when no rule can be applied for any membrane and no rule is in execution. In this case the computation is *successful* and the vector of multiplicities of objects in x is accepted.

Definition 4.5 *An accepting P system with symport/antiport rules*

$$\Pi = (O, \mu, w_1, \cdots, w_m, E_{inf}, E_{fin}, R_1, \cdots, R_m, i_{inp}),$$

is time-free *if and only if every timed system in the set*

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

for $R = R_1 \cup \cdots \cup R_m$*, accepts the same set of vectors of numbers.*

We use the notation $fPsPP_m(i,k)$ to denote the family of sets of vectors of natural numbers accepted by *time-free* accepting P systems with symport/antiport rules, with at most m membranes, symport rules of weight at most i and antiport rules of weight at most k.

Symport and antiport rules are powerful mechanisms for synchronizing. In fact, even by using only two membranes, symport rules of weight one and antiport rules of weight at most two, it is possible to construct universal time-free P systems as shown in the following theorem.

Theorem 4.8 $fPsPP_1(1,2) = PsRE$.

Proof. Consider a register machine $M = (n, R, l_0, l_h)$ according to the definition given in Section 1.1.5. We will simulate its accepting computation with a P system constructed in the following way. The value stored in register j of M will be represented by the multiplicity of an associated object a_j. Let $Lab(R) = \{l_1, l_2, \cdots, l_q\}$.

Formally, we construct the P system $\Pi = (O, \mu, w_1, E_{inf}, E_{fin}, R_1, 1)$ where:

$O = Lab(\mathcal{P}) \cup \{S_l, T_l \mid l \in Lab(R)\} \cup E_{inf} \cup \{k, k_1\}$;

$E_{inf} = \{a_r \mid 1 \leq r \leq n\}$;

$E_{fin} = l_1 l_2 \cdots l_q S_1 S_2 \cdots S_q T_1 T_2 \cdots T_q k k_1$;

$\mu = [_1\,]_1$;

$w_1 = l_0$; in addition, for a vector $(r_1, \cdots, r_{n-2})$ to be accepted by M, we introduce in the input region 1 the multiset $a_1^{r_1} \cdots a_{n-2}^{r_{n-2}}$;

R_1 is defined as follows:

- for each addition instruction $(l_1 : \text{ADD}(r), l_2) \in R$, $l_1, l_2 \in Lab(R)$, $1 \leq r \leq n$, we add to R_1 the rule $(a_r l_2, in; l_1, out)$;
- for each subtraction instruction $(l_1 : \text{SUB}(r), l_2, l_3) \in R$, $l_1, l_2, l_3 \in Lab(R)$, $1 \leq r \leq n$, we add to R_1 the rules:

  $(S_{l_1} k, in; l_1, out)$,

  $(T_{l_1}, in; S_{l_1} a_r, out)$,

  $(k_1, in; k, out)$,

  $(l_3, in; S_{l_1} k_1, out)$,

  $(l_2, in; T_{l_1} k_1, out)$;
- for instruction $(l_h : \text{HALT}) \in R$, $l_h \in Lab(R)$, we add to R_1' the rule (l_h, out).

The system Π simulates the work of the register machine program as follows. Suppose that the content of region 1 is described by the multiset $a_1^{i_1} \cdots a_n^{i_n} l_1$.

The simulation of an instruction $(l_1 : \text{ADD}(r), l_2) \in R$ is straightforward: the object l_1 (which corresponds to register machine instruction label l_1) is sent out of the membrane, while, in the same time, objects a_r and l_2 are brought inside. In this way, the new instruction label is generated and the number of objects a_r is incremented.

If an instruction $(l_1 : \text{SUB}(r), l_2, l_3)$ has to be executed by M, then object l_1 is sent out, as in the previous case, and at the same time, objects S_{l_1} and k are brought inside the region. Object S_{l_1} represents the "command" to remove one object a_r from the current multiset in case this is possible; object k represents a "checker" – it will check whether or not the subtraction command was successfully accomplished.

If in the region there exists at least one copy of object a_r, then rule $(S_{l_1} a_r, out; T_{l_1}, in)$ starts in the same time with rule $(k, out; k_1, in)$. Next, since object S_{l_1} is not anymore in region 1, only the rule $(T_{l_1} k_1, out; l_2, in)$ can start. Therefore, the label l_2 of the next register machine instruction is generated and the simulation can continue.

In case in region 1 there is no object a_r, only rule $(k, out; k_1, in)$ can be started (in fact, rule $(S_{l_1} a_r, out; T_{l_1}, in)$ cannot be started). Next, since in region 1 there is no object T_{l_1}, only the rule $(S_{l_1} k_1, out; l_3, in)$ can be started. Therefore, the label l_3 of the next register machine instruction is generated and simulation can continue.

When object l_h appears in region 1, indicating that the register machine program has to terminate, then the rule (l_h, out) is executed and the simulation terminates as well.

One can notice that the simulation of register machine instructions is deterministic. In addition, the result of the simulations does not depend on the execution times of the involved rules. Therefore, the constructed system is time-free. Consequently, the system Π accepts the same set of vectors as accepted by M, irrespective of the duration of rules. □

4.3 Final Remarks

Proton pumping P systems have been introduced in [9]. The definition and the results presented in Section 4.1 are from this paper. Actually, the universality result for proton pumping P systems has been recently improved in [8]. There, it has been shown that universality can be obtained by using only one proton and two membranes. In [8] the idea of protons has been also generalized to symport rules: a symport can have weight at most two, but one of the two objects transported must be a proton. Also in this case universality has been obtained by using one proton and two membranes.

In the same paper it has been shown how, by interpreting protons as bi-stable catalysts, universality is obtained with one bi-stable catalyst and one membrane, thus improving the previous result present in [95]. In [8] there have also been considered time-free proton pumping P systems showing that, in this case, four protons and two membranes are enough to get universality.

Signal-promoters have been introduced in [19] where the reader can also find more detailed biological motivations. In that paper there is also the suggestion to avoid the global clock by using signals-promoters to synchronize the execution of independent rules. However, in [19] the time of execution of each rule is still fixed to be one clock-step. The idea to associate a time of execution to the rules of a P system is presented in [58], where timed and time-free P systems have been introduced. Actually, in the same paper two classes of P systems have been introduced: time-free P systems (the ones recalled in this chapter) and *clock-free P systems*, where each application of a rule can have a different time of execution. In [58] it has been proved that the class of clock-free P systems with promoters and catalytic evolution rules (one catalyst) is universal.

The results given in Section 4.2.1 are from [48]. The result concerning non-cooperative evolution and signaling rules presented in Section 4.2.2 have been recalled from [58]. The universality for time-free P systems with bi-stable catalytic evolution rules (with an unbounded number of bi-stable catalysts) has been originally proved in [48] and the proof is presented here in Theorem 4.5. Recently, this result has been improved in [8]: there, it has been shown that four bi-stable catalysts are enough to get universality. Deciding whether a P system with bi-stable catalysts is time-free is not possible (Theorem 4.6); the proof of this theorem has been recalled from [48]. A first universality proof for P systems using catalytic evolution and signaling rules together with priority can be found in [58]. An improved result that does not make use of priority is reported in [59] and the proof is presented here in Theorem 4.7. Section 4.2.4 concerning time-free P systems with symport/antiport rules is from [59].

In [11], the authors have considered time-free evolution-communication P systems showing that they are universal by using two membranes, non-cooperative evolution rules, and symport/antiport rules of weight at most one. Because the result is obtained for time-free systems, it can be considered a generalization of the result presented here in Theorem 2.2. Finally, it is worth to mention that in [58] the authors introduced and briefly investigate *partially time-free P systems*, that are P systems producing always the same result when the time of execution of the rules fits certain specified conditions.

In [117] it is presented the class of *non-synchronized P systems* (non-synchronized P systems are systems with symbol-objects where in each region a rule $A \to A$ is present, for each symbol A in the alphabet). Time-free P systems and non-synchronized P systems have strong similitudes – in both

the models there is a certain degree of "asynchronism" that must be managed during the computation. However, the results obtained are strongly different for the two models: while time-free P systems with bi-stable catalytic evolution rules are universal (as shown in this chapter), non-synchronized P systems can generate only the family of length sets of context-free languages when using bi-stable catalysts (the proof can be found in [117]). This big difference of computational power is due to the fact that in time-free P systems it is still possible to make use of the maximal parallelism as a tool for synchronizing the computation. Finally, we recall that in [74] one investigates asynchronous (tissue) P systems where the rules are executed in a sequential way.

Chapter 5

Evolution and Observation

As shown in the previous chapter it is possible to get new bio-inspired computing devices by looking "inside" living cells, and considering the processes that take place there.

However, we can get new paradigms of computation also by looking to the procedures that biologists follow during their experiments. For instance, the role (and the power) of the observer is fundamental in most biological experiments.

In this chapter we show that observation can have a crucial role also in the construction of a computing device.

In Chapter 3 a software simulator for EC P systems has been presented. There, we have discussed the results of the simulations, playing, in a certain sense, the role of an observer of the biological system simulated. In that case, the ability to obtain an useful result was related with the ability to observe the system (looking, for instance, at the results produced by the software).

We can generalize this idea and say that the observer can be any machine that is able to watch a (configuration of a) system and associate a certain meaning to it.

Starting from this idea, a new way to look at a computation can be introduced.

A computing device can be constructed using two less powerful systems: the first one, which is a biological system, simply "lives", passing from one state (configuration) to the next one, producing some behavior; the second system is placed outside and watches the biological system (for this reason this system is called observer). The observer, following a set of rules, translates the behavior of the biological system into a more readable output. In this way, the pair composed by the biological system and the observer can be considered a computing device, as described in Figure 5.1. The computation is divided between the biological system and the observer, that is able to watch and to interpret the behavior of the biological system. This new

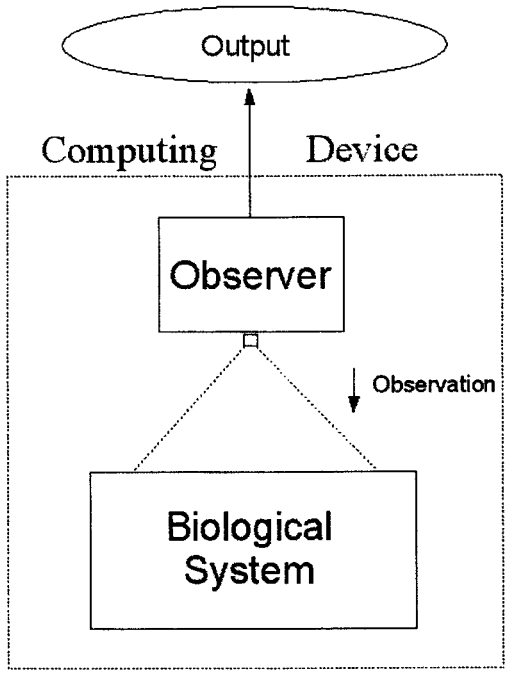

Figure 5.1: Conceptual view of the evolution/observation framework.

framework is refereed to as *system/observer* or, also, *evolution/observation.*

An immediate application of this framework is to membrane computing. *P/O systems*, presented in this chapter, are computing devices, composed by a P system with symbol-objects, which plays the role of the biological system, and by a multiset finite automaton, which represents the observer.

The membrane system realizes its part of computation passing from one configuration to the next one: the collection of all the possible computations of the membrane system defines its behavior. A single computation can be seen, in a natural way, like the sequence of configurations "touched"; therefore, a behavior is a set of sequences of configurations and we can consider behaviors consisting of finite sequences (if we collect only halting computations) or behaviors consisting of infinite sequences (if we collect only non-halting computations). The multiset automaton processes every sequence of configurations present in the behavior of the P system and translates it into a conventional string over a finite alphabet. The collection of all these strings is the output language of the P/O system.

Moreover, during this process of translation, the multiset finite automaton can decide to "discard" a string if it represents a "wrong" computation (i.e., a non-desired configuration is detected in the sequence). We show that, using this strategy, even membrane systems with only non-cooperative evo-

lution rules observed by multiset finite automata are universal computing devices.

The evolution/observation framework is general and can be also applied to the more classical theory of computation. An idea is that, instead of considering the observation of biological systems, we can consider a more general system/observer architecture where the bio-system is substituted by any system that, according to some specific rules, is able to move from one configuration to the next, producing in this way a behavior; the observer is any recognizing machine that translates such a behavior into a readable output.

We show how this general architecture can be applied to formal languages theory, presenting the class of *grammar/observer (G/O) systems*, where a grammar plays the role of the system and an automaton – be it a finite automaton or even a Turing machine – plays the role of the observer. A useful feature is to let the observer output a special label, if an undesired configuration/sentential form is observed. Any string containing it is then not considered as a valid result. This quite simple approach turns out to be very powerful.

We present three modes of generating languages for G/O systems. In the first mode G/O systems constituted by a context-free grammar and by a finite state automaton generate a family of languages between the context-free and context-sensitive languages. In the other two cases the same combination is even equivalent to a Turing machine. In one case this result can be further strengthened to using only a commutative context-free grammar instead of a conventional one.

The evolution/observation framework can be also used to construct accepting devices. Actually, using this idea, it is possible to imagine any biological system as an accepting device. This is achieved by taking a model of a biological system, introducing an input to such a system and observing its evolution.

If the evolution of the system is of an expected type (for example, it follows a regular predetermined pattern), the input is accepted by the bio-system, otherwise it can be considered rejected.

More formally, an external observer is used for extracting an abstract, formal behavior from the evolution of the biological system. A decider is an automata-like machine that checks whether the observed behavior of the biological system is of the expected type. The combination observed-decider can be seen like an accepting device.

This strategy can be seen as a non-standard way to approach natural computing.

We apply this strategy to splicing systems, a formal model of the recombination of double stranded DNA sequences (for simplicity, we call them

DNA strands) under the action of a ligase and restriction enzymes (endonucleases). In particular, we present an accepting device, *splicing recognizer* (in short, SR), constructed by joining a decider, an observer, and a splicing system.

A schematic view of the model is depicted in Figure 5.2.

The SR works in the following way. An input marked DNA strand (represented by a string w) is inserted in a test tube. Due to the presence of restriction enzymes, the input strand changes, as it starts to recombine with other DNA strands present in the test tube. A sequence of intermediate marked DNA strands is generated. This constitutes the evolution of the input marked DNA strand. Schematically this is presented with the sequence of w, w', w'', w''' in Figure 5.2.

The external observer associates to each intermediate marked strand a certain label taken from a finite set of possible labels. It writes these labels onto an output tape in their chronological order. In Figure 5.2 this corresponds to the string $a_1a_2a_3a_4$. This string represents a code of the obtained evolution. When the marked strand becomes of a certain predetermined "type" the observation stops.

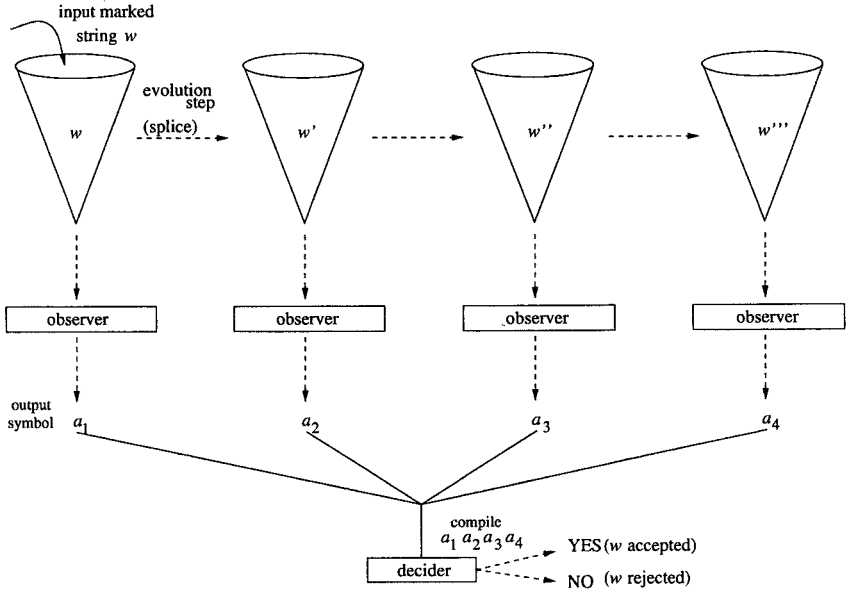

Figure 5.2: The splicing/observer architecture.

At this point the decider checks if the entire evolution of the input marked DNA strand described by the string $a_1a_2a_3a_4$ has followed a certain pattern, i.e., if it is in a certain language. If this is true, the input string w is accepted by the SR; otherwise it is considered to be rejected.

Using this strategy it is possible to obtain very powerful accepting sys-

tems even when very simple components are used.

For instance, we show that having only a finite state automaton as observer of the evolution of a finite splicing system (with a finite set of splicing rules) is already enough to simulate a Turing machine. This is a remarkable jump in acceptance power since it is well known that a finite splicing system by itself can generate only a subclass of the class of regular languages.

The application of the evolution/observation strategy to splicing systems is motivated by recent developments in biology where several techniques for observing the dynamics of a single DNA molecule and in general of a single biomolecule have been developed.

5.1 P/O Systems

In this section we present P/O systems based on the idea that is possible to compute by observing the evolution of a membrane system.

To define P/O systems we need to use an MFA (multiset finite automaton) like a transducer: to each final state is associated a label and a successful/accepting computation produces the label of the final state it halts in, other computations produce λ as output. The labels constitute the *output alphabet* Σ *of the automaton.* In the case of P/O systems, such automata are applied over multisets that represent a configuration of a membrane system. As we have already seen, a configuration of a membrane system is the m-tuple of multisets of objects present at any time in the m regions of the system. For simplicity, here, the configuration is represented by just one multiset, in which every object is indexed with the label of the region where it appears.

The output of an MFA A when applied over (a multiset representing) a configuration c is denoted by $A(c)$; for a sequence $c_1 \dots c_n$ of configurations we write $A(c_1 \dots c_n)$ for the string $A(c_1) \cdot \dots \cdot A(c_n)$; in a natural way this can be extended to an infinite sequence of configurations $c_1 \dots c_n \dots$, considering $A(c_1 \dots c_n \dots)$ for the infinite string $A(c_1) \cdot \dots \cdot A(c_n) \cdot \dots \cdot$.

First we recall the formal definition of P/O systems; later we give an example that shows how the new constructed device works and we present the proof of a (quite surprising) universality.

Definition 5.1 *A P/O system is a pair $\Omega = [\Pi, A]$ constituted of a P system Π and an observing MFA A with output alphabet Σ, which then is also the output alphabet of the entire system. The language generated by the system is then*

$$L(\Omega) = \{A(s) \mid s \in B(\Pi)\},$$

where the behavior $B(\Pi)$ *is the set of all possible sequences of configurations during any computation (halting or non-halting) of the system Π. Thus the language contains all those words which the observer can produce during the possible computations of the underlying P system.*

5.1.1 Considering Non-Halting Computations

We first investigate P/O systems where only the non-halting computations of the observed P system are considered.

In particular, we consider *conservative* P systems. Formally a *conservative P system with symbol-objects* is a P system

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

where for each evolution rule $u \rightarrow v$ in R_i and for all $i = 1, 2, \cdots m$, we have that $|u| = |v|$ and through the external membrane nothing is sent out.

Of course, in this case the number of objects present in the system can only remain constant. If not specified, a P system is *non-conservative.*

In this case we do not need to consider explicitly an MFA mapping configurations into an alphabet because due to the finite number of configurations of the observed P system, we can simply use the set of all these as output alphabet Σ of the MFA (for this reason such MFA is called *trivial* MFA with output alphabet Σ).

We define $\mathbb{B}_{inf} = \{L(\Omega) \cap \Sigma^\omega \mid \Omega = [\Pi, A]$ where Π is a conservative P system and A is a trivial MFA with output alphabet $\Sigma\}$.

Informally, $\mathbb{B}_{inf}$ is the class of all infinite behaviors of conservative P systems.

Now we first notice the expected (from the finiteness of the number of configurations) fact that conservative P systems only exhibit regular behaviors.

Theorem 5.1 *$\mathbb{B}_{inf}$ is a proper subset of the family of ω-regular languages.*

Proof. To prove the inclusion we construct for every conservative system an automaton which accepts exactly the language describing the system's behavior. As already mentioned, a conservative P system can reach only finitely many configurations. We now draw a directed graph with one node for every such configuration. Edges go from every node to the nodes of configurations which can be reached from the configuration of the original node in one computation step. The edges are labeled with the names of the configurations of the nodes they end in. We declare all these nodes as final. Then we add one more initial node with just one accordingly labeled edge to the system's starting configuration node. Thus we quite obviously obtain a Büchi automaton, which recognizes exactly the words describing behaviors of the original system.

The language $\phi(a^*b^\omega)$ is ω-regular but cannot be the behavior of any conservative system, because any system which allows a transition from configuration a directly to a does so infinitely often; thus also the word a^ω

would have to be in the behavior. This shows that the inclusion is proper. □

It should be noted that the constructed Büchi automaton is deterministic. With Theorem 1.17 we immediately obtain the following two statements about languages in $\mathbb{B}_{inf}$.

Corollary 5.1 *Every language in $\mathbb{B}_{inf}$ contains an ultimately periodic word.*

Corollary 5.2 *Two behaviors of conservative systems are equal if and only if they contain the same ultimately periodic words.*

At the same time, clearly, most behaviors also contain non-periodic sequences.

As a rather trivial example a system of one membrane $\Pi = (\{a, b\}, [\,]_1, a, \{a \to b, a \to a, b \to a, b \to b\})$, results in such an automaton accepting the language $\phi(a \circ (a \vee b)^\omega)$ with a and b being the labels for the configurations where the respective object is present.

Returning to Theorem 5.1, a subsequent task is to characterize more closely those ω-regular languages which can be behaviors of conservative systems. Upon closer inspection one notices that one major source of problems is the star operation (as seen in the second part of the proof of Theorem 5.1). Informally speaking, this is the case, because an iteration once started can go on forever.

However, even the elimination of the bounded iteration from the ω-regular expressions does not restrict the ω-regular languages sufficiently for describing $\mathbb{B}_{inf}$. A further condition is that the context to the right of all occurrences of a specific letter must be the same – this is imposed by the fact that a system in a certain configuration can always do exactly the same things in the following steps. The following definition can help to cope with this problem.

Definition 5.2 *An ω-regular expression is called* without finite loops, *if it does not use the operator * and does not contain any letter more than once.*

With this we are able to make one more step in characterizing those ω-regular expressions which describe behaviors of conservative systems.

Theorem 5.2 *Every ω-regular expression without finite loops describes a finite union of behaviors of conservative systems.*

Proof. For every ω-regular expression there is a Büchi-automaton accepting exactly those infinite words which are in the expression's interpretation. This automaton we can transform into an equivalent one, where all transitions leading to a state read the same letter. If this is not the case in the original automaton, then we can make the following transformation.

A state with more than one, say m, letters labeling transitions leading to it is replaced by m new states. Each of these has as incoming transitions all the ones that used to lead with the same letter to the original state; in case these included a loop, from all other new states according transitions are added, too. The outgoing transitions for all the new states are the same as for the original one with the aforementioned way of treating loops. Obviously, the new automaton is equivalent to the old one and step by step we can arrive at an automaton of the desired form. Thus we can can put on each state as a label the unique letter already labeling all incoming transitions.

If there is an initial state s without any incoming transition, it does not receive a label – in that case we make all the states reachable by one transition from s initial and delete s itself along with all transition exiting from it. Thus we arrive at an automaton A, which for our purposes is equivalent to the original one; the labels of the deleted transitions are still preserved in their former target states. Suppose now that this automaton A has exactly n states. We draw the transition diagram of A and construct a membrane system with only one membrane and n different objects.

A technical problem appearing now is that the automaton might have several, say k, initial states, while the conservative system only has one initial configuration. Therefore we take the union over the behaviors of k such systems, which are identical up to their initial configurations. In the respective configurations only one copy of the objects corresponding to A's respective initial states is present. Further the systems have an evolution rule $a \to b$ exactly if in A there is a transition between the corresponding states. There are no further rules.

Finally, it should also be noted that a Büchi automaton constructed from an expression without finite loops cannot contain any loop without a final state. Thus any infinite behavior generated by the system will be a word to be accepted, the membrane system cannot generate any incorrect infinite behavior. □

Corollary 5.3 *Every ω-regular expression not using * describes the homomorphic image of a finite union of behaviors of conservative systems.*

Proof. For every expression not using * we can construct an expression of the same structure, only replacing each copy of each letter by a differently indexed one. Following Theorem 5.2 every such expression describes the behavior of a conservative system. A morphism "deleting the indices" will take us back to the original language. □

Notice that all considerations/results of this section are also true for quasi-conservative systems, which for example delete objects or create only

a limited number of them (for example, by a rule $a \rightarrow bb$ where no new a are produced). As quasi-conservative we classify all systems with only a finite number of possible configurations.

5.1.2 Considering Halting Computations

As already mentioned before now we restrict our attention to languages of finite words. In the definition of a P/O system we consider only the halting computations of the underlying P system.

Then the class of languages considered is $\mathbb{B}_{fin} = \{L_{fin}(\Omega) = (L(\Omega) \cap \Sigma^*) \mid \Omega \text{ is a P/O system with output alphabet } \Sigma\}$.

Here we add a special feature: the ability of a P/O system $\Omega = [\Pi, A]$ to "select" the words produced.

When a sequence of configurations is to be rejected, then, the observer A produces the *special symbol* $\perp$, that is still associated to some final states of A, but is *not in the output alphabet* Σ of the P/O system Ω. Then the intersection with Σ^*, used in the definition of $\mathbb{B}_{fin}$, filters out all generated words containing $\perp$.

From the combination system/observer a great power emerges: a P/O system is universal even when it is composed by a simple P system with symbol-objects and non-cooperative rules, and by an observer that is a DMFAD, with the ability to produce the special symbol $\perp$. We recall that (Theorem 1.20 and Theorem 1.19) the power of the two systems do not exceed context-free (in their respective domains).

Theorem 5.3 *For each $L \in RE$ there exists a P/O system $\Omega = [\Pi, A]$, where Π is a P system with symbol-objects and non-cooperative evolution rules and A is a DMFAD, such that $L_{fin}(\Omega) = L$.*

Before presenting the universality proof we prefer to give an example that shows how a P/O system works and how a big power can emerge from the combination system/observer.

The following example shows how a P/O system composed by a P system with non-cooperative evolution rules and by an MFA can generate the non-semilinear language $\{a^{2^n} \mid n \geq 1\}$.

The P system Π is defined in the following way:

$$\Pi = (O, \mu, w_0, w_1, R_0, R_1),$$

where

$$\begin{aligned} O &= \{a, b\}, \\ \mu &= [_0\, [_1\,]_1\,]_0, \\ w_0 &= \lambda, \\ w_1 &= a, \end{aligned}$$

$$R_0 = \{b \to \lambda\},$$
$$R_1 = \{a \to aa, a \to b, b \to b, b \to b_{out}\}.$$

It should be noticed that we do not define any output membrane (the P/O system does not need it).

The MFA A is defined over the input alphabet $V = \{a, b_0, b_1\}$ that is used to represent the configuration of the P system Π during a computation. We recall that each configuration is represented by one multiset, in which every object is indexed with the label of the region where it occurs (in the alphabet V we use the letter a because the symbol-object a can occur exclusively in the region 1).

The output alphabet of the MFA A is $\Sigma = \{\alpha\}$. Moreover, the MFA uses the special symbol $\perp$ (not present in Σ) to mark the unsuccessful termination of string generation, in the way described before.

The transitions of the MFA are described in Figure 5.3 (we recall that $\overline{a}$ stands for checking that there is no a present in the multiset in input and the symbol generated as output by the MFA is written on the edges that exit from the final states).

In the P/O system, strings are generated as follows: first only a is present and the observer's path to the right ensures that only the rule $a \to aa$ is applied in the P system, in region 1; thus after n steps there are 2^n objects a in region one. In order to terminate they must all be changed into b at some step, using the rule $a \to b$ in region one, and then the second branch of the observer controls the further process. The copies of b are put one by one into region zero, using the rule $b \to b_{out}$, and there they are immediately deleted. Every configuration where there is a b in membrane zero results in an output α, all others result in λ or $\perp$.

Thus $L_{fin}(\Omega) = \{\alpha^{2^n} \mid n \geq 1\}$.

This example shows clearly how the non-determinism in form of the competing rules $a \to aa$ and $a \to b$, as well as $b \to b$ and $b \to b_{out}$ can be "controlled" by the observer. In the first case it checks that the two rules are not applied simultaneously, in the second case the observer ensures that the second rule is applied to only one b, while all the others remain unchanged ($b \to b$ is applied).

Now we are ready to give the proof of Theorem 5.3.

We construct, for every two-counter automaton C accepting a language from V^*, a P/O system $\Omega = [\Pi, A]$ with output alphabet $\Sigma = V$, and using the special symbol $\perp \notin \Sigma$, generating the same language. The two-counter automaton has a finite number of transitions, say n; these we can number and label by $t_1, \ldots, t_n$ and sometimes we will say "transition t_j" for "the transition with label t_j". The automaton's set of states shall be $Q = \{s_0, \ldots, s_m\}$.

The basic idea of the simulation presented here is that the membrane

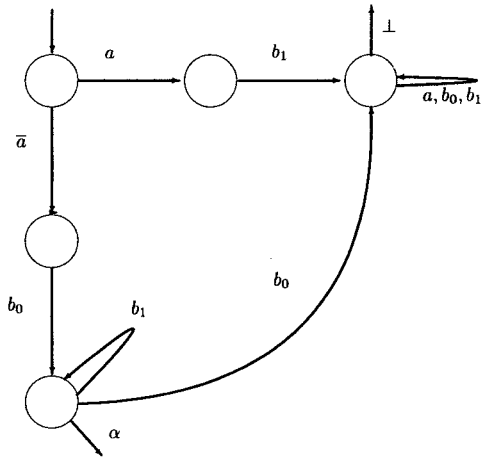

Figure 5.3: The observer of the system generating $\{a^{2^n} \mid n \geq 1\}$.

system simulates the transitions of the two-counter machine using non-cooperative evolution rules and choosing in a non-deterministic way the next transition to execute, and, at the same time, the observer checks (using the ideas illustrated in the example) that the sequence of rules applied by the membrane system corresponds to a correct sequence of transitions of the two-counter machine C. We suppose the two-counter automaton C in the notation given in Section 1.1.5.

Formally, we present the membrane system with symbol-objects, and we give some hints on how the observer, a deterministic multiset finite automaton, works.

The membrane system is

$$\Pi = (O, \mu, w_1, w_2, w_3, w_4, R_1, R_2, R_3, R_4),$$

with:

$$\begin{aligned}
O &= Q \cup \{s'_k, s''_k, s'''_k \mid k \in \{0, \ldots, m\}\} \cup \{s^1_k, s^2_k \mid k \in \{0, \ldots, m\}\} \\
&\cup \{S, temp, temp', temp'', temp''', x, b, b', c, c', s^{\nabla}\} \\
&\cup \{a, a', a'' \mid a \in V\} \cup \{t_j \mid t_j \text{ is a transition}\}, \\
\mu &= [_1\, [_2\, [_3\,]_3\, [_4\,]_4\,]_2\,]_1, \\
w_1 &= S, \qquad w_2 = \lambda, \\
w_3 &= \lambda, \qquad w_4 = \lambda, \\
R_1 &= \{S \to aS, a \to a, a \to a_{in2} \mid a \in V\} \cup \{S \to temp_{in2}(s_0)_{in2}\} \\
&\cup \{b' \to b'', c' \to c''\}, \\
R_2 &= \{temp \to temp', temp' \to temp'', temp'' \to temp'''\} \\
&\cup \{c \to c'_{in3}, c \to c'_{out}, b \to b'_{in4}, b \to b'_{out}, a \to a',
\end{aligned}$$

$$
\begin{aligned}
& a'' \to (a)temp', x \to temp\} \\
\cup\ & \{s_k \to s'_k, s'_k \to s''_k, s''_k \to s'''_k, s'''_k \to (s^{\nabla})_{out}, s^1_k \to s'_k, s^2_k \to s_k\} \\
\cup\ & \{t_j \to (t'_j)_{out} \mid t_j \text{ is a transition}\} \\
\cup\ & \{a' \to s^2_k b' c' t_j x \mid \text{ transition } t_j \text{ is} \\
& \delta(s, a, \alpha, \beta) = [s_k, \mathrm{R}, +1, +1]\} \\
\cup\ & \{a' \to s^2_k b'_{in4} t_j x \mid \text{ transition } t_j \text{ is } \delta(s, a, \alpha, \beta) = [s_k, \mathrm{R}, +1, \eta]\} \\
\cup\ & \{a' \to s^2_k c'_{in3} t_j x \mid \text{ transition } t_j \text{ is } \delta(s, a, \alpha, \beta) = [s_k, \mathrm{R}, \eta, +1]\} \\
\cup\ & \{a' \to s^2_k t_j x \mid \text{ transition } t_j \text{ is } \delta(s, a, \alpha, \beta) = [s_k, \mathrm{R}, \eta, \eta]\} \\
\cup\ & \{a' \to s^1_k b'_{in4} c'_{in3} t_j a'' \mid \text{ transition } t_j \text{ is} \\
& \delta(s, a, \alpha, \beta) = [s_k, \text{stay}, +1, +1]\} \\
\cup\ & \{a' \to s^1_k b'_{in4} t_j a'' \mid \text{ transition } t_j \text{ is} \\
& \delta(s, a, \alpha, \beta) = [s_k, \text{stay}, +1, \eta]\} \\
\cup\ & \{a' \to s^1_k c'_{in3} t_j a'' \mid \text{ transition } t_j \text{ is} \\
& \delta(s, a, \alpha, \beta) = [s_k, \text{stay}, \eta, +1]\} \\
\cup\ & \{a' \to s^1_k t_j a'' \mid \text{ transition } t_j \text{ is } \delta(s, a, \alpha, \beta) = [s_k, \text{stay}, \eta, \eta]\} \\
& \text{for all } a \in V, k \in \{0, \ldots, m\} \\
& (\text{recall } m = |Q| + 1), \alpha, \beta \in \{z, nz\}, \eta \in \{0, -1\}, \\
R_3 \ =\ & \{c' \to c, c \to c, c \to c_{out}\}, \\
R_4 \ =\ & \{b' \to b, b \to b, b \to b_{out}\}.
\end{aligned}
$$

In a first phase, in a non-deterministic way, the letters of a word w in V^* are generated in region 1 like by a regular grammar using the evolution rules present there: $\{S \to aS, a \to a \mid a \in V\} \cup \{S \to temp_{in2}(s_0)_{in2}\}$. When the letters have been generated, the symbol *temp* is produced, sent in region 2, and the simulation of the counter automaton can start.

In the first step, *temp* is changed to *temp'* and together with the application of this rule, one symbol-object a (for some $a \in V$) is sent from region 1 to region 2 – this simulates the reading of the input by the counter machine. The observer uses *temp'* to check that the configuration reached after this type of step is correct: exactly one symbol a is introduced from region 1 to 2; only if region 1 is empty no symbol is introduced. For each (correct) configuration c of the membrane system, where region 2 contains an object $a \in V$, we have $A(c) = a$. All the other configurations are mapped to λ or $\perp$ by the observer. Also in this step, the symbol s_k becomes s'_k in region 2; s_k stores the value of the state of the counter machine, reached after the last transition.

In the next step, *temp'* becomes *temp''* and the injected symbol a becomes a'. Also the two counters (represented by regions 3 and 4) are decreased each by one unit, if possible: this means that, from region 3 and

4 is sent out one copy of the object c and b respectively. Moreover in this same step the symbol s'_k becomes s''_k. The configuration obtained after such step is checked by the observer: in particular, the observer controls that the decrease of the counters has been executed correctly (each one decreased by one; if not, then the observer checks if it was empty).

Following this, $temp''$ becomes $temp'''$ and in this step the rule (we call these rules "transition-rules") associated to some transition of the counter automaton, in which the symbol a is read, is applied (we say the "transition is applied"). In this same step the b and c extracted from the regions (counters) 3 and 4, become b' and c' and they move (one, or both), independently, to region 1 if the transition applied decrease (one, or both) the counters, or otherwise they move into their respective region. By application of a transition-rule the symbol t_j is produced (it stores the label of the transition applied). Then, looking at the configuration obtained, the observer "reads" the t_j produced, the symbol s'''_k (that is storing the "old" state, from before the transition), the (old) values of the counters (looking at symbols b and c in regions 4 and 3 and, b' and c', present, if any, in regions 1, 3 or 4). Using these informations the observer checks if the transition was made correctly (the rule corresponding to the right transition in the two-counter machine was applied).

Moreover, the transition-rule applied has produced also the object a'' and the object s^1_k, or the objects x and s^2_k; a'' and s^1_k (that indicates the new state reached after the transition) are produced if in the transition simulated the head is not moved (the symbol introduced before must be read again), otherwise x and s^2_k are produced (in this case a new symbol $a \in V$ from region 1 must enter in region 2). If a'' has been produced, then it generates, in the next step, the objects a and $temp'$, and, at the same time, s^1_k becomes s'_k and stores the new state reached after the transition. In the same step the counters have been updated and the old state has been converted to the symbol s^{∇}, which is never used again. In this case a new transition rule can be applied in the next step, and no new symbol enters from region 1; the simulation continues as shown before.

In the other case, the x generates in the next step $temp$, and at the same time s^2_k becomes s_k; now this stores the new state reached after the transition. In the same step the counters have been updated and the old state has been removed. A new symbol $a \in V$ must enter from region 1 in the next step and the simulation continues as shown before.

When the computation of the automaton halts exactly after the application of the last transition, there is no more letter a or a'' present anywhere, but a state has just been produced in region 2. This situation can be detected by the observer, and then also the conditions for accepting can be checked – if they are not fulfilled then the special symbol $\perp$ is put out like in all the other cases above, where a check is unsuccessful. Otherwise the membrane system will halt within a few steps and the word w will have been

generated by Ω. □

5.2 Evolution and Observation: A Non-Standard Way to Look at Formal Languages

In this section we consider *grammar/observer (G/O) systems* that, as described in the Introduction to this chapter, are generative devices based on the presented evolution/observation framework. In this case, a formal grammar plays the role of the bio-system and an automaton – be it a finite automaton or even a Turing machine – plays the role of the observer.

Initially, a formal definition of the observer is given, and then the definition of a G/O system is recalled. Three ways of working for a G/O system (*initial, free, always writing*) are presented.

5.2.1 Automata with Singular Output

First we need to formalize the notion of an observer. We essentially need a device mapping arbitrarily long strings into just one singular symbol. Therefore we use a special variant of finite automata: the set of states is labeled with the symbols of an output alphabet Σ or with λ. Any computation of the automaton produces as output the label of the state it halts in (we are not interested in accepting / not accepting computations and therefore also not interested in the presence of final states); because the observation of a certain string should always lead to a fixed result, we consider here only deterministic and complete automata.

This way an automaton with singular output is a tuple $A = (Q, V, \Sigma, q_0, \delta, \sigma)$ with state set Q, input alphabet V, initial state $q_0 \in Q$, and a complete transition function δ as known from conventional finite automata; further there is the output alphabet Σ and a labeling function $\sigma : Q \mapsto \Sigma \cup \{\lambda\}$.

The output of the automaton is the label of the state it stops in. For a string $w \in V^*$ and an automaton A we then write $A(w)$ for this output; for a sequence $w_1, \ldots, w_n$ of $n \geq 1$ strings over V^* we write $A(w_1, \ldots, w_n)$ for the string $A(w_1) \cdots A(w_n)$.

Moreover, we will often also want the observer to be able to reject some words. To realize this we simply choose a special symbol $\perp \notin \Sigma$ and an extended output alphabet $\Sigma_\perp = \Sigma \cup \{\perp\}$; σ then is a mapping from the set of states Q to $\Sigma_\perp \cup \{\lambda\}$. $\perp$ is produced, when a bad sentential form is observed and thus the entire sequence is to be rejected. Then, using the intersection with the set Σ^*, it is possible to filter out the strings produced containing the special symbol $\perp$.

The class of all (deterministic) automata with singular output will be denoted by FA_O. In the same way observers can be obtained from other

classes of automata such as pushdown automata, linear bounded automata or Turing machines.

5.2.2 G/O Systems

Now we define the central notion of this section, the *G/O system.*

A *G/O system* is a pair $\Omega = (G, A)$ constituted by a generative grammar $G = (N, T, S, P)$ and an observing automaton with singular output $A = (Q, V, \Sigma, q_0, \delta, \sigma)$ with output alphabet Σ, which then is also the output alphabet of the entire system. The automaton's input alphabet must be the union of N and T from the grammar to make the desired interaction possible, i.e., $V = N \cup T$.

We distinguish three different modes of generation that define three different models of *G/O systems*:

1. writing a non-empty output in every step (*always writing G/O systems*),
2. writing a non-empty output in every step after an initialization phase of writing only λ (*initial G/O systems*), and
3. changing between empty and non-empty output in an arbitrary manner (*free G/O systems*).

In the case of an *always writing G/O system* Ω the language generated is

$$\begin{aligned} L_a(\Omega) &= \{A(w_1, w_2, \ldots, w_n) \mid S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n, \\ &\quad w_n \in T^* \text{ and } A(w_i) \neq \lambda, \text{ for all } 1 \leq i \leq n\}. \end{aligned}$$

Note that the very first sentential form, which is always the starting symbol, is excluded from the observation. Otherwise, all words in $L(\Omega)$ would start with the same letter, if the observer was deterministic. The sentential forms $w_i, 1 \leq i \leq n$, are obtained by applying the productions of G, in the standard sequential way, as recalled in Section 1.1.2.

The best way to ensure the last condition, i.e., that λ is never written as output, is of course to define the observer in such a way that it can never produces empty output.

For an *initial G/O system* the output is defined as

$$\begin{aligned} L_i(\Omega) &= \{A(w_0, w_1, \ldots, w_n) \mid S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n, \ w_n \in T^*, \\ &\quad \text{and for all } i \in \{1, \ldots, n\}, A(w_0, w_1, \ldots, w_{i-1}) \neq \lambda \\ &\quad \text{implies } A(w_i) \neq \lambda\}. \end{aligned}$$

Finally, a *free G/O system* generates a language in the following non-restricted manner:

$$L_f(\Omega) = \{A(w_0, w_1, \ldots, w_n) \mid S = w_0 \Rightarrow w_1 \Rightarrow \ldots \Rightarrow w_n, \ w_n \in T^*\}.$$

Thus in all three cases the language contains all those words which the observer can produce during the possible terminating derivations of the underlying grammar. Derivations which do not terminate do not produce a valid output; this means that we only take into account finite words. Of course, by considering the other case of non-terminating derivations the G/O systems could also be used to generate languages of infinite words.

We have already mentioned and will actually mainly investigate the variant, where the observer can also produce a special symbol $\perp \notin \Sigma$; whenever it appears in a word, this word should not be considered for the output language, thus we take

$$L_{\perp,a}(\Omega) = L_a(\Omega) \cap \Sigma^*.$$

Analogously, the languages $L_{\perp,i}(\Omega)$ and $L_{\perp,f}(\Omega)$ are defined.

For a class $\mathcal{G}$ of grammars and a class $\mathcal{O}$ of observers, $\mathcal{L}_a(\mathcal{G},\mathcal{O})$, $\mathcal{L}_i(\mathcal{G},\mathcal{O})$, $\mathcal{L}_f(\mathcal{G},\mathcal{O})$, $\mathcal{L}_{\perp,a}(\mathcal{G},\mathcal{O})$, $\mathcal{L}_{\perp,i}(\mathcal{G},\mathcal{O})$, and $\mathcal{L}_{\perp,f}(\mathcal{G},\mathcal{O})$ denote the classes of all languages generated by G/O systems with grammars from $\mathcal{G}$ and observers from $\mathcal{O}$ in the respective modes. Quite obviously we obtain for fixed classes of grammars $\mathcal{G}$ and observers $\mathcal{O}$ the inclusions

$$\mathcal{L}_a(\mathcal{G},\mathcal{O}) \subseteq \mathcal{L}_i(\mathcal{G},\mathcal{O}) \subseteq \mathcal{L}_f(\mathcal{G},\mathcal{O})$$

and the same for the variants where the special symbol $\perp$ can be written.

In a completely analogous way, *L/O systems* can be defined, where the grammar is replaced by a Lindenmayer system. Here we refrain from doing so more formally, because in the sequel this notion will only be used on a rather informal level.

For simplicity, in what follows, we present only the mappings that the observers define, without giving a real implementation (in terms of finite automata) for them.

5.2.3 Always Writing G/O Systems

The first mode of generation we present is the one of writing an output in every step, i.e., we consider the model of always writing G/O systems. This is maybe the most natural one, since in most cases the observation of an experiment should be complete, at least if about the outcome nothing is known beforehand. We will consider here the variant where also $\perp$ can be written as output.

We can see that every context-free language L is in $\mathcal{L}_{\perp,a}(CF, FA_O)$. For this consider a context-free grammar in the Greibach normal form. There, all right sides of rules are elements of TN^*; this means that in every step exactly one terminal is produced. Since the grammar is still context-free, there is a leftmost derivation for every word in L. In this derivation all sentential forms except the initial S are strings over T^+N^*. An observer

can check that a derivation produces only sentential forms of this structure. Then it can output the rightmost terminal for each one of these sentential forms, and the result equals the string derived by the original grammar.

However, $\mathcal{L}_{\perp,a}(CF, FA_O)$ is bigger than only CF. As an example for a non-context-free language from this class we present $\{a^n b^n c^n \mid n > 0\}$. The grammar for this language is

$$G = (\{S, A, B, C\}, \{t\}, S, \{S \to A, A \to AB, A \to C, B \to C, C \to t\}).$$

The derivations whose observations will result in the output of words $a^n b^n c^n$ are the ones of the form

$$S \Rightarrow A \overset{n-1}{\Rightarrow} AB^{n-1} \Rightarrow CB^{n-1} \overset{n-1}{\Rightarrow} C^n \Rightarrow tC^{n-1} \overset{n-1}{\Rightarrow} t^n.$$

To produce the output and rule out all other derivations, the observing automaton A will realize the following mapping from the set of sentential forms of G into $\{a, b, c, \perp\}$:

$$A(w) = \begin{cases} a & \text{if } w \in AB^*, \\ b & \text{if } w \in C^+B^*, \\ c & \text{if } w \in t^+C^*, \\ \perp & \text{else.} \end{cases}$$

While $\{a^n b^n c^n \mid n > 0\}$ is still semilinear, also the language $\{a^{2^n} \mid n > 0\}$, which is not semilinear, lies in the class $\mathcal{L}_{\perp,a}(CF, FA_O)$. To show this, we first recall that a binary tree of depth k has exactly $2^n - 1$ nodes, if the root is considered to already have depth one. Therefore a context-free grammar, which can have full binary trees as derivation trees, and an observer, which can check that only such derivations are made, can generate $\{a^{2^n} \mid n > 0\}$ when interacting in a G/O system. For this example our grammar is $G = (\{S, A, B, C, T_1, T_2, T_3\}, \{t\}, S, P)$, where the set P of productions is

$$\{S \to A, A \to BB, B \to CC, C \to AA, A \to BT_1, B \to CT_2,$$
$$C \to AT_3, A \to t, B \to t, C \to t, T_1 \to t, T_2 \to t, T_3 \to t\}.$$

Now, for example, derivations resulting in the outputs a^2 and a^8 are

$$S \Rightarrow A \Rightarrow t$$

and

$$S \Rightarrow A \Rightarrow BB \Rightarrow CCB \Rightarrow CCCT_2 \Rightarrow tCCT_2 \Rightarrow ttCT_2 \Rightarrow tttT_2 \Rightarrow tttt,$$

respectively. The conditions that the observer has to check (for putting out a in every step) are rather straightforward to see after the first example.

A sentential form containing any A must be completely changed to one from B^*, the same from B to C, and finally from C to A. This is done by

use of the rules $A \to BB, B \to CC$ and $C \to AA$ respectively. To ensure that the entire sentential form is completely changed, the observer maps to a only the sentential forms of the form $A^+B^* \cup B^+C^* \cup C^+A^*$ – others result in the output $\perp$. We notice that there are never more than two different non-terminals present at the same time.

To stop the derivation the rightmost non-terminal of the sentential form must produce the corresponding T_i, with $i \in \{1, 2, 3\}$, by using one of the rules $A \to BT_1$, $B \to CT_2$, or $C \to AT_3$, and then the only possible further steps are to derive all non-terminals to t. In these cases, the sentential forms mapped to a must be of the form $t^*A^*T_3 \cup t^*B^*T_1 \cup t^*C^*T_2 \cup t^+$.

Therefore the mapping of the observer is

$$A(w) = \begin{cases} a & \text{if} \quad w \in A^+B^* \cup B^+C^* \cup C^+A^* \cup \\ & \qquad t^*A^*T_3 \cup t^*B^*T_1 \cup t^*C^*T_2 \cup t^+, \\ \perp & \text{else.} \end{cases}$$

We note here one difference between the two examples: while in the first case for a word of length n the workspace used by the grammar is $\frac{n}{3}$, in the second case the space is logarithmic in the length of the output.

Trying to characterize the class $\mathcal{L}_{\perp,a}(CF, FA_O)$ more closely, we can easily see that it is contained in the class of context-sensitive languages. In fact, the G/O system must write a symbol of output at each step and then the total space used by such a system for the generation of a word w is bounded by a constant depending only on the context-free grammar. Therefore, every language generated by an always writing G/O systems is context-sensitive by the workspace theorem.

5.2.4 Initial G/O Systems

The second variant of G/O systems is called *initial G/O system* and in this model the sentential forms of an initial phase are mapped exclusively to λ. After the first non-empty output only non-empty outputs can be produced – looking back to the biochemical motivation of the concept of evolution and observer, this would correspond to a phase of initializing an experiment and then a phase of actual observation.

Such an initialization phase – which is not restricted, but can be much longer than the actually observed phase – greatly enhances the power of our G/O systems. Indeed, with the same classes of grammars and observers as in Section 5.2.3 we obtain computational universality in this case.

Theorem 5.4 $\mathcal{L}_{\perp,i}(CF, FA_O) = RE$.

Proof. The inclusion from left to right is obvious, because every G/O system of the considered type can be simulated by a Turing machine. To show the opposite direction, we start from a grammar $G_1 = (N, T, S, P_1)$

in Kuroda normal form and construct an equivalent G/O system with a context-free grammar $G_2 = (N_2, T, S, P_2)$ and an FA_O observer A such that $L_{\perp,i}(G_2, A) = L(G_1)$.

For every derivation of G_1 resulting in a word w, grammar G_2 will derive a sentential form of non-terminals $\overline{a}$ corresponding to the letters a of w. Up to that point no output is written. Then, starting from the front, the non-terminals are rewritten to the corresponding terminals, and always the last one already converted is written to the output by the observer.

Our starting point will be G_1. The rules of the forms $A \to BC, A \to a$, and $A \to \lambda$ from G_1 we can adopt for G_2 without any change. Only the rules $AB \to CD$ have to be simulated by the context-free grammar in several steps. For this we assume a one-to-one labeling from the set P_1' of all these rules in G_1 into $r_1, \ldots, r_l$; without loss of generality we can assume that the four non-terminals of each such rule are distinct. Further we will use a set $\overline{T} = \{\overline{a} \mid a \in T\}$ of non-terminals and for every rule $r : AB \to CD$ a set $R_i = \{A_r, B_r, C_r, D_r\}$. Then we have

$$N_2 = N \cup \overline{T} \cup \bigcup_{r_i \in P_1'} R_i$$

as the set of non-terminals of G_2. The set of rules is

$$\begin{aligned} P_2 &= \{A \to BC, A \to \lambda \mid \text{productions of } P_1\} \\ &\cup \{A \to \overline{a} \mid A \to a \in P_1\} \\ &\cup \{A \to A_r, A_r \to C_r, C_r \to C, \\ &\quad B \to B_r, B_r \to D_r, D_r \to D \mid r : AB \to CD \in P_1'\} \\ &\cup \{\overline{a} \to a \mid a \in T\}. \end{aligned}$$

With this grammar any derivation of G_1 can be simulated, where all applications of rules of the forms $A \to BC, A \to a$, and $A \to \lambda$ can be done in one step, just as in the original grammar. Only when rules of the form $AB \to CD$ have to be simulated, the execution of the necessary steps in the correct order must be guaranteed by the observer such that only derivations possible already in G_1 can be realized. How all this is done can be seen again by looking at the mapping the observer A realizes:

$$A(w) = \begin{cases} \lambda & \text{if } W \in N^*, \\ \lambda & \text{if } w \in N^* A_r N^* \text{ and } r : AB \to CD \in P_1', \\ \lambda & \text{if } w \in N^* A_r B_r N^* \text{ and } r : AB \to CD \in P_1', \\ \lambda & \text{if } w \in N^* C_r B_r N^* \text{ and } r : AB \to CD \in P_1', \\ \lambda & \text{if } w \in N^* C_r D_r N^* \text{ and } r : AB \to CD \in P_1', \\ \lambda & \text{if } w \in N^* D_r N^* \text{ and } r : AB \to CD \in P_1', \\ \lambda & \text{if } w \in \overline{T}^* N^*, \\ a_i & \text{if } w \in T^* a_i \overline{T}^*, \\ \perp & \text{else.} \end{cases}$$

As all the cases (except N^* and $\overline{T}^*N^*$, which have equal output λ, though) are disjoint and described in a regular way, clearly a finite automaton with singular output can realize this mapping. The structure of the case $T^*a_i\overline{T}^*$ guarantees that the resulting word is read from left to right and only after all non-terminals remaining in the case $\overline{T}^*N^*$ have been converted to non-terminals from $\overline{T}$. □

This last theorem can also give a characterization of the recursively enumerable languages in terms of always writing G/O systems. In particular, using the same construction as in the proof of Theorem 5.4, only writing a special letter c instead of λ, we arrive at the following result.

Corollary 5.4 *Every recursively enumerable language over an alphabet T is the left quotient of some $L_{\perp,a}(G, A)$ with c^*, where $c \notin T$, G is a context-free grammar and A is a finite automaton with singular output with output alphabet $\Sigma = T \cup \{c\}$.*

5.2.5 Free G/O Systems

An immediate corollary of Theorem 5.4 is the fact that $\mathcal{L}_{\lambda,\perp}(CF, FA_O) = RE$. In this section, however, we show how a G/O system composed of even less powerful components, namely a locally commutative context-free grammar and a finite state automaton is universal, if the output λ can be used without any restriction and if some derivations can be ignored by using the symbol $\perp$.

We give here only the sketch of the proof, because the idea is very similar to the proof given for the universality of a evolution/observation system composed by a membrane system, with non-cooperative rules, observed by a multiset finite automaton (see Theorem 5.3). Essentially, the membrane system's evolution must be sequentialized, therefore we concentrate here on the grammar.

The proof is based on the fact that every recursively enumerable language is accepted by a two-counter automaton.

These automata serve very well for our purposes, because in contrast to a stack or a working tape a counter can be simulated without paying attention to the order of its elements; they are all the same and can therefore be distributed freely throughout the entire sentential form.

Theorem 5.5 $\mathcal{L}_{\perp,f}(LCCF, FA_O) = RE$.

Proof. From an arbitrary two-counter automaton C accepting a language L, and thus for any recursively enumerable language L, we construct a G/O system $\Omega = (G, A)$ composed by a locally commutative context-free grammar G and a finite automaton with singular output A, such that $L_{\perp,f}(\Omega) = L$. The grammar will first generate the letters of the output

word, then simulate a computation of C; all this is done with the sentential form consisting only of non-terminals, i.e. everything remains rewriteable. In a final phase G rewrites everything to terminals to end the derivation and thereby the computation of the entire G/O system. The observer's task is to guarantee that the grammar's derivation steps occur in the desired order.

Since C has only a finite number of transitions, we can label them starting from T_0; we will in the sequel speak of "transition T_t" or just "T_t" instead of "the transition with label T_t". We suppose that the two-counter automaton C is given according to the notation presented in Section 1.1.5 (i.e., V is the input alphabet, Q is the set of states).

Now the locally commutative context-free grammar G for our system is the quadruple $(N, \{c, f\}, S, P)$ where

$$\begin{aligned} N &= \{S, \Delta_0, \Delta_1, \Delta_2, \Delta_3, M, M', M'', M_{pop}, N, N', N'', N_{pop}, \diamond\} \\ &\cup \{A_i, A_i', \underline{A}_i \mid a_i \in V\} \\ &\cup \{T_t \mid t_j \text{ is a transition}\} \\ &\cup \{S_k \mid s_k \in Q\}. \end{aligned}$$

The set of productions of G is

$$\begin{aligned} P &= \{S \to [A_i, S], S \to [S_0, \Delta_0]\} \\ &\cup \{A_i \to A_i' \mid a_i \in V\} \\ &\cup \{\Delta_0 \to \Delta_1, \Delta_1 \to \Delta_2, \Delta_2 \to \Delta_3, \Delta_3 \to \Delta_0, \} \\ &\cup \{M \to M_{pop}, M \to M', M_{pop} \to \diamond, M' \to M, M'' \to M\} \\ &\cup \{N \to N_{pop}, N \to N', N_{pop} \to \diamond, N' \to N, N'' \to N\} \\ &\cup \{S_k \to S_k', S_k' \to \diamond \mid s_k \in Q\} \\ &\cup \{S_k \to f \mid s_k \text{ is a final state of } C\} \\ &\cup \{X \to c \mid X \in N\} \\ &\cup \{T_t \to \diamond \mid T_t \text{ is a transition}\} \\ &\cup \{A_i' \to [S_k, M'', N'', T_t], \\ &\quad \underline{A}_i \to [S_k, M'', N'', T_t] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, R, +1, +1]\} \\ &\cup \{A_i' \to [S_k, M'', T_t], \\ &\quad \underline{A}_i \to [S_k, M'', T_t] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, R, +1, \eta]\} \\ &\cup \{A_i' \to [S_k, N'', T_t], \\ &\quad \underline{A}_i \to [S_k, N'', T_t] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, R, \eta, +1]\} \\ &\cup \{A_i' \to [S_k, T_t], \\ &\quad \underline{A}_i \to [S_k, T_t] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, R, \eta, \eta]\} \\ &\cup \{A_i' \to [S_k, M'', N'', T_t, \underline{A}_i], \\ &\quad \underline{A}_i \to [S_k, M'', N'', T_t, \underline{A}_i] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = \end{aligned}$$

$$
\begin{aligned}
& [s_k, \text{stay}, +1, +1]\} \\
\cup\ & \{A'_i \to [S_k, M'', T_t, \underline{A}_i], \\
& \underline{A}_i \to [S_k, M'', T_t, \underline{A}_i] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, \text{stay}, +1, \eta]\} \\
\cup\ & \{A'_i \to [S_k, N'', T_t, \underline{A}_i], \\
& \underline{A}_i \to [S_k, N'', T_t, \underline{A}_i] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, \text{stay}, \eta, +1]\} \\
\cup\ & \{A'_i \to [S_k, T_t, \underline{A}_i], \\
& \underline{A}_i \to [S_k, T_t, \underline{A}_i] \mid T_t \text{ is } \delta(s, a_i, \alpha, \beta) = [s_k, \text{stay}, \eta, \eta]\} \\
& \text{for all } a_i \in V, s, s_k \in Q, \alpha, \beta \in \{z, nz\}, \eta \in \{0, -1\}.
\end{aligned}
$$

We describe a derivation in G that results in a valid output by A and along the way we hint to the conditions that this automaton has to check. We shall see that it will essentially suffice to check whether certain non-terminals are present zero, one, or more times (and these conditions can be represented using a regular expression and then checked by the observer).

To simulate a computation of the two-counter automaton accepting a word w, the grammar first produces in a regular derivation the word wS from S; here w stands for the word of non-terminals A_i corresponding to the word of letters a_i in the obvious way.

Once W is generated, the rule $S \to S_0\Delta_0$ introduces the initial state of C and Δ_0 to synchronize the further steps. The actual simulation of C can begin.

For a transition reading a new symbol, one A_j from the sentential form is selected to be read and marked by converting it to A'_j. Depending on whether the transition will pop from a counter or not, one M and one N are changed to M_{pop}, N_{pop} or M', N'. Ending this preparation phase, also the state symbol is primed.

After the application of the rule $\Delta_0 \to \Delta_1$, i.e., when all the mentioned symbols are present, and an A'_j has been "read", the observer produces the corresponding a_j as an output. This is the unique situation where A produces a non-empty output other than $\perp$.

Now the transition to be applied is selected, by applying the unique rule with the corresponding T_t in its right side. This is a very crucial step, because the compatibility of the transition and the symbols present must be checked by the observer: had the change of the stack been guessed right? Was the original state – still being present as S'_l – one in which T_t can be applied? We shall look at the conditions to be checked in some more detail and first recall that we can view a transition as an element of the set

$$Q \times V \times \{z, nz\} \times \{z, nz\} \times Q \times \{\text{stay}, R\} \times \{+1, 0, -1\}, \times \{+1, 0, -1\}.$$

The S'_l and S_k present in the sentential form are the two elements from Q, the states before and after execution of the transition, respectively. Since they are unique for each transition T_j, their correctness can be checked. The

letter from the input alphabet does not have to be checked, because any T_t is produced directly from the correct A_j. The emptiness of the counters is equivalent to the non-occurrence of M and M' and M_{pop} or the respective Ns. The correct move of the input head is already implicit in the rule producing T_t.

Moreover the observer must also check whether the update of the values of the counters is done correctly. Here the presence of only M_{pop} stands for decrement, the presence of only M' will result in an unchanged counter, and the presence of M' and M'' increments the counter, M'' alone increments an empty counter, and non-occurrence of all three means that there was an empty counter (no M) and it is not incremented. All other combinations of the three symbols are illegal, for the other counter the Ns are checked analogously. Since every T_t effects a unique stack behavior, this check does not require any additional states of A. If not everything is alright, $\perp$ is written as output.

The next step in C's simulation is the production of Δ_2, which signals the beginning of the final phase, where all the symbols not necessary any more are cleaned up. In the remaining steps the stack symbols are either restored to M and N or converted to $\diamond$. Also T_t and S'_k become $\diamond$; this symbol is ignored by the observer in the sequel. Finally, when Δ_3 is present, A checks that this clean-up has completely been done. Now again we have arrived at a sentential form ready for the application of another transition, and Δ_0 can be produced.

In a very similar manner, transitions not reading anything new from the input tape are simulated. Only in the first step no symbol A'_j is produced, but rather $\underline{A_j}$ is present from the start in the sentential form. It was already produced in the simulation of the last transition, which did not move the input head and therefore left $\underline{A_j}$ as a copy of the symbol it had read.

In contrast to the corresponding previous case, no letter a_j must be produced as output, because the symbol read from the input tape has already been read in a previous step. In this case the $\underline{A_j}$ present selects the transition T_t to simulate (analogously to the A'_j in the previous case); then this transitions simulation continues with sentential forms of the same content as already explained for the latter case.

The two-counter automaton accepts a word, if it has read the entire input, the counters are empty and it is in a final state. Therefore for all final states a derivation to the terminal f exists. Whenever f and only nonterminals are present, the observer can check whether these conditions are satisfied (no M, N, A_i), and otherwise output $\perp$. Once f is present, no more transitions of C can be simulated and the observer maps every sentential form to λ. With a termination of the derivation taking all remaining nonterminals to c, the grammar stops and w has been produced as output.

Because the special symbol $\perp$ is introduced every time that the conditions checked by the observed are not respected and because of the synchro-

nization provided by the Δs, then every output word is also accepted by C and, on the other hand, every computation of C can be simulated in Ω. □

Taking into account the close relation between context-free grammars and 0L systems, it is natural to expect a similar result for L systems. Indeed, we immediately obtain a corresponding result for a system consisting of an E0L system and again a finite automaton.

Corollary 5.5 *Every recursively enumerable language can be generated by a free L/O system constituted by an E0L system and a finite automaton with singular output.*

Proof. From the grammar of Theorem 5.5, we can construct an E0L system with the same set of rules plus a rule $X \to X$ for each non-terminal of the grammar except the synchronizing Δs. Thus the enforced changing of Δ in every step ensures that all the other changes are made, especially that a configuration producing an output does not remain unchanged and produces this output twice. Using the same observer, this L/O system generates the same language as the G/O system from Theorem 5.5. □

Remark. Another interesting fact is that intuitively the observer's ability to produce $\perp$, i.e., to eliminate certain computations, seems a powerful and essential feature in all three models investigated in this and the previous sections. However, we obtain all recursively enumerable languages over Σ simply by intersection of a language over $\Sigma_\perp$ with the regular language Σ^*. Now notice that recursive languages are closed under intersection with regular sets (can be obtained just by looking to the definition of recursive languages). Therefore, there must exist some grammar/observer systems Ω generating a non-recursive $\mathcal{L}_f(\Omega)$. Because in the proof of Theorem 5.5 we make very heavy use of the option to trash strings, it is by no means obvious, how this result can be obtained in a direct way.

5.3 Splicing Recognizers

A *splicing recognizer* (in short, SR) is an accepting device constructed by applying the idea of evolution and observation to splicing systems in the way presented in the Introduction to this chapter. In this case as underlying (observed) biological system we consider a splicing system (more precisely an H scheme) with the particular feature that, at any time, exactly one string of the produced language is marked.

Consider an alphabet V (splicing alphabet) and two special symbols # and \$ not in V. A splicing rule (over V) is a string of the form $u_1\#u_2\$u_3\#u_4$, where $u_1, u_2, u_3, u_4 \in V^*$.

For a splicing rule $r = u_1\#u_2\$u_3\#u_4$ and strings $x, y, z_1, z_2 \in V^*$ we write $(x, y) \Longrightarrow_r (z_1, z_2)$ iff $x = x_1u_1u_2x_2$, $y = y_1u_3u_4y_2$, $z_1 = x_1u_1u_4y_2$, $z_2 = y_1u_3u_2x_2$. We refer to z_1 (z_2) as the first (second) string obtained by applying the splicing rule r.

An H scheme is a pair $\nu = (V, R)$ where V is an alphabet, and $R \subseteq V^*\#V^*\$V^*\#V^*$ is a set of splicing rules. For a given H scheme $\nu = (V, R)$ and a language $L \subseteq V^*$ we define $\nu(L) = \{z_1, z_2 \in V^* \mid (x, y) \Longrightarrow_r (z_1, z_2)$, for some $x, y \in L$, $r \in R\}$.

When restriction enzymes (and a ligase) are present in a test tube, they do not stop acting after one cut and paste operation, but they act iteratively.

Given a *initial language* $L \subseteq V^*$, and an H scheme $\nu = (V, R)$ we define the iterated splicing as: $\nu^0(L) = L$, $\nu^{i+1}(L) = \nu^i(L) \cup \nu(\nu^i(L)), i \geq 0$.

In this work, as previously discussed, we are interested in observing the evolution of a specific marked string introduced, at the beginning, in the initial language L and called *input marked string.*

Given an initial language L, an *input marked string* $w \in L$, a *target marked language* L_t and an H scheme ν, the scheme defines a sequence of marked strings that represents the evolution of the input marked string w, according to the splicing rules defined in ν (for simplicity we suppose $w \notin L_t$). The sequence of marked strings, $\langle w_0 = w, w_1, \cdots, w_k \rangle$, for $k \geq 1$ and $w_k \in L_t$, is constructed in the following iterative way (w_i is the marked string associated to the set $\nu^i(L), 0 \leq i \leq k$).

Each new marked string is obtained by splicing the old marked string, until a marked string w_k from the target marked language L_t is reached or the marked string cannot be spliced.

The first string of the sequence is the input marked string, $w_0 = w$.

If $w_i \in L_t$, $i \geq 1$, then the sequence stops (the marked string is among the ones of the target marked language).

If there is no $x \in \nu^i(L)$, $i \geq 0$, such that $(w_i, x) \Longrightarrow_r (z_1, z_2)$ or $(x, w_i) \Longrightarrow_r (z_1, z_2)$ for some $r \in R$, then the sequence stops (the marked string cannot be spliced).

If $x, y \in \nu^i(L)$, $i \geq 0$, with $w_i = x$ (or $w_i = y$) and there exists a rule $r \in R$ such that $(x, y) \Longrightarrow_r (z_1, z_2)$, then $w_{i+1} = z_1$. In this case, if the marked string can be subject to more than one splicing rule, producing different strings, the choice of the next marked string is done in a non-deterministic way. Notice that we always consider the first string produced as the new marked one.

Because the update of a marked string is made in a non-deterministic way, given an input marked string w, an initial language L, a target marked language L_t, and an H scheme ν, it is possible to get different sequences of intermediate marked strings. The collection of all these sequences is denoted by $\nu(w, L, L_t)$.

For a splicing rule $r = u_1\#u_2\$u_3\#u_4$ we denote by $rad(r)$ the length

of the longest string u_1, u_2, u_3, u_4; we say that this is the *radius* of r. The radius of an H scheme is the maximal radius of its rules.

In what follows, we denote by FIN_{H_k} the class of H schemes with radius at most k and using finite set of splicing rules.

Like in Section 5.2.1, as observer we use a finite state automaton with singular output defined as $O = (Q, V, \Sigma, q_0, \delta, \sigma)$ with state set Q, input alphabet V, initial state $q_0 \in Q$, and a complete deterministic transition function δ; the output alphabet is Σ and the labeling function is $\sigma : Q \mapsto \Sigma$. The automaton with singular output works exactly like the one defined in Section 5.2.1.

As *deciders* we require devices accepting a certain language over the output alphabet Σ of the corresponding observer as just introduced. For this we do not need any new type of device but can rely on conventional finite automata with input alphabet Σ. The output of the decider D, for a word $w \in \Sigma^*$ in input, is denoted by $D(w)$. It consists of a simple `yes` or `no`.

Following the idea presented in the Introduction to this Chapter, we put together the components just defined and a *splicing recognizer* (in short *SR*) is a quintuple $\Omega = (\nu, O, D, L, L_t)$; $\nu = (V, R)$ is an H scheme, O is an observer $(Q, V, \Sigma, q_0, \delta, \sigma)$, D is a decider with input alphabet Σ, L and L_t are finite languages, respectively, the initial and the target marked language for ν.

The *language accepted by the SR* Ω is the set of all words $w \in V^*$ provided there exists a sequence $s \in \nu(w, L, L_t)$ such that $D(O(s)) =$ `yes`; formally,

$$L(\Omega) = \{w \in V^* \mid \text{there is } s \in \nu(w, L, L_t) \text{ such that } D(O(s)) = \texttt{yes}\}.$$

5.3.1 A Short Example

It is well-known in the splicing literature that the family of languages generated by splicing systems using only a finite set of splicing rules and a finite initial language is strictly included in the family of regular languages. In the following example we show that an SR composed by such an H scheme with a finite set of rules, finite initial language, finite target marked language and finite state automata as observer and decider, can recognize non-regular languages. This example is just a hint towards the fact that the combination splicing system-observer-decider can be powerful even when the single components are simple.

In particular, we construct an SR recognizing the non-regular language $\{o_l a^n b^n o_r \mid n \geq 0\}$. The SR $\Omega = (\nu, O, D, L, L_t)$ is defined as follows: the H scheme is $\nu = (V, R)$, with $V = \{o_l, o_r, a, b, a', b', X_1, Y_1, X_2, Y_2\}$ and $R = \{r_1 : \#bo_r\$X_2\#b'o_r,\ r_2 : o_l a'\#Y_2\$o_l a\#,\ r_3 : \#b'o_r\$X_1\#o_r,\ r_4 : o_l\#Y_1\$o_l a'\#\}$. The initial language is $L = \{X_2 b' o_r, o_l a' Y_2, X_1 o_r, Y_1 o_l\}$. The target marked language is $L_t = \{o_l o_r\}$.

The observer O has input alphabet V and output alphabet $\Sigma = \{l_0, l_1, l_2, l_3, \perp\}$. The mapping it implements is:

$$O(w) = \begin{cases} l_0 & \text{if } w \in o_l(a^*b^*)o_r, \\ l_1 & \text{if } w \in o_l(a^*b^*b')o_r, \\ l_2 & \text{if } w \in o_l(a'a^*b^*b')o_r, \\ l_3 & \text{if } w \in o_l(a'a^*b^*)o_r, \\ \perp & \text{else.} \end{cases}$$

The decider D is a finite state automaton, with input alphabet Σ, that gives a positive answer exactly if a word belongs to the regular language $l_0(l_1l_2l_3l_0)^*$.

The observer checks that the splicing rules are applied in the order r_1, r_2, r_3, r_4, and this corresponds to remove, in an alternating way, a b from the right and an a from the left of the input marked string. In this way, at least one of the evolutions of the input marked string will be of the kind accepted by the decider if, and only if, the input marked string is in the language $\{o_l a^n b^n o_r \mid n \geq 0\}$. Notice that, at each step, the marked string present is spliced only with one of the strings present in the initial language.

To clarify the working of the SR Ω we show the acceptance of the input marked string $w_0 = o_l aabbo_r$. For simplicity, we only show the evolution of the input marked string and the output of the observer, step by step.

- Step 0: input marked string $w_0 = o_l aabbo_r$; $O(w_0) = l_0$;
- Step 1: apply rule r_1; new marked string $w_1 = o_l aabb'o_r$; $O(w_1) = l_1$;
- Step 2: apply rule r_2; new marked string $w_2 = o_l a'abb'o_r$; $O(w_2) = l_2$;
- Step 3: apply rule r_3; new marked string $w_3 = o_l a'abo_r$; $O(w_3) = l_3$;
- Step 4: apply rule r_4; new marked string $w_4 = o_l abo_r$; $O(w_4) = l_0$;
- Step 5: apply rule r_1; new marked string $w_5 = o_l ab'o_r$;$O(w_5) = l_1$;
- Step 6: apply rule r_2; new marked string $w_6 = o_l a'b'o_r$; $O(w_6) = l_2$;
- Step 7: apply rule r_3; new marked string $w_7 = o_l a'o_r$; $O(w_7) = l_3$;
- Step 8: apply rule r_4; new marked string (in the target marked language) $w_8 = o_l o_r$; $O(w_8) = l_0$.

Obviously, the entire observed evolution $l_0l_1l_2l_3l_0l_1l_2l_3l_0$ is of the kind accepted by the decider D, so the string w_0 is accepted by the SR Ω.

5.3.2 Preliminary Results

An SR can accept even non-context-free languages as stated in the following theorem. The trick used here consists in the rotation of the input marked string, during its evolution. The regular observer can control that this kind of rotation is done in a correct way.

Theorem 5.6 *There is an SR Ω such that $L(\Omega)$ is a non-context-free, context-sensitive language. Moreover, the splicing scheme of Ω can be taken to be of radius ≤ 3.*

Proof. We construct an SR Ω accepting the non-context-free language

$$\{o_l w o_r \mid w \in \{a,b,c\}^+, |w|_a = |w|_b = |w|_c\}.$$

The SR $\Omega = (\nu, O, D, L, L_t)$ is defined as follows: the H scheme is $\nu = (V, R)$, with

$$V = \{a, b, c, o_l, o_r, X_1, X_2, X_3, X_4, X_5, X_6, X_a, X'_a, X_b, X'_b, X_c, X'_c\}.$$

The set of splicing rules of R is divided in two groups, according to their use.

The first group consists of the rules used to rotate the marked string.
r_1 : $\{d\#o_r\$X_1\#X_a o_r \mid d \in \{a,b,c\}\}$,
r_2 : $\{\#dX_e o_r\$X_2\#X_d X_e o_r, \mid e,d \in \{a,b,c\}, e \neq d,\}$,
r_3 : $\{o_l X'_e\#X_3\$o_l\#d, \mid e,d \in \{a,b,c\}\}$,
r_4 : $\{\#X_d X_e o_r\$X_4\#X_e o_r, \mid e,d \in \{a,b,c\}, e \neq d\}$,
r_5 : $\{o_l e\#X_5\$o_l X'_e\# \mid e \in \{a,b,c\}\}$.

The second group of splicing rules is used to remove a symbol a, b, or c from the marked string.
r_6 : $\#aX_a o_r\$X_6\#X_b o_r$,
r_7 : $\#bX_b o_r\$X_6\#X_c o_r$,
r_8 : $\#cX_c o_r\$X_6\#X_a o_r$.

The initial language of the SR is the finite language

$$\begin{aligned} L &= \{X_1 X_e o_r, o_l X'_e X_3, X_4 X_e o_r, o_l e X_5 \mid e \in \{a,b,c\}\} \\ &\cup \{X_2 X_d X_e o_r \mid d,e \in \{a,b,c\}, e \neq d\} \\ &\cup \{X_6 X_b o_r, X_6 X_c or, X_6 X_a o_r\}. \end{aligned}$$

The target marked language is $L_t = \{o_l X_a o_r\}$.
The observer O has input alphabet V and output alphabet

$$\Sigma = \{l_0, \perp\} \cup \{l_{e,1}, l_{e,2}, l_{e,3}, l_{e,4} \mid e \in \{a,b,c\}\}.$$

The mapping implemented by the observer is

$$O(w) = \begin{cases} l_0 & \text{if } w \in o_l\{a,b,c\}^+ o_r, \\ l_{e,1} & \text{if } w \in o_l\{a,b,c\}^+ X_e o_r,\ e \in \{a,b,c\}, \\ l_{e,2} & \text{if } w \in o_l\{a,b,c\}^* X_d X_e o_r,\ e,d \in \{a,b,c\}, \\ l_{e,3} & \text{if } w \in o_l X'_d\{a,b,c\}^* X_d X_e o_r,\ e,d \in \{a,b,c\}, \\ l_{e,4} & \text{if } w \in o_l X'_d\{a,b,c\}^* X_e o_r,\ e,d \in \{a,b,c\}, \\ \lambda & \text{if } w \in \{o_l X_a o_r\}, \\ \perp & \text{else.} \end{cases}$$

The decider D is a finite state automaton, with input alphabet Σ, that gives a positive answer exactly if and only if, a word belongs to the regular language

$$l_0(l_{a,1}(l_{a,2}l_{a,3}l_{a,4}l_{a,1})^* l_{b,1}(l_{b,2}l_{b,3}l_{b,4}l_{b,1})^* l_{c,1}(l_{c,2}l_{c,3}l_{c,4}l_{c,1})^*)^+.$$

At the beginning of the computation the input marked string is of the kind $o_l\{a,b,c\}^+ o_r$ and it is mapped by the observer to l_0. If the input marked string is not of this type, then the observer outputs something different from l_0, and the entire evolution is not accepted by the decider D. In the first step, the splicing rule $d\#o_r\$X_1\#X_a o_r$ from r_1 is used, and in this way a new marked string of the type $o_l\{a,b,c\}^+ X_a o_r$ is obtained and mapped by the observer to $l_{a,1}$. The introduced symbol X_a indicates that we want to search (and then to remove) a symbol a from the obtained marked string. This searching is done by rotating the marked string, until a symbol a becomes the symbol immediate to the left of X_a. The rotation of the string is done by using the splicing rules given in the first group.

A rotation of the string consists in moving the symbol immediately to the left of X_a, to the right of o_l; one rotation is done by applying, in a consecutive way, a rule from r_2, from r_3, from r_4 and finally from r_5 (the precise rules to apply depend on the symbol to move during the rotation). The sequence of marked strings obtained during a rotation is mapped by the observer to the string $l_{a,2}l_{a,3}l_{a,4}l_{a,1}$. The $*$ present in the regular expression describing the decider language, indicates the possibility to have 0, or more consecutive rotations before a symbol a comes to be the symbol immediately to the left of X_a.

The observer checks that each rotation is made in a correct way; that is, the symbol removed from the left of X_a by using a rule from r_4, is exactly the same symbol introduced to the right of o_l, by using the corresponding rule in r_3. This condition is checked in the fourth line of the observer mapping; if this regular condition is not respected, then the observer outputs $\perp$ and the entire evolution of the input marked string will not be accepted by the decider D.

Once a symbol a becomes the symbol immediately to the left of X_a, and the rotations can stop, then it is deleted by using the splicing rule r_6.

When rule r_6 is applied, the new marked string obtained is of the kind $o_l\{a,b,c\}^+X_bo_r$ that is mapped by the observer to $l_{b,1}$; the inserted symbol X_b, indicates that now we search the symbol b.

In an analogous way, by using consecutive rotations, a symbol b is placed immediately to the left of X_b and then is removed by using rule r_7. In this case, the sequence of marked strings obtained during each rotation is mapped by the observer to $l_{b,2}l_{b,3}l_{b,4}l_{b,1}$. Once rule r_7 is applied, the new marked string obtained is of the kind $o_l\{a,b,c\}^+X_co_r$ and is mapped by the observer to $l_{c,1}$.

Again analogously, the symbol c is searched for and then deleted by using rule r_8; in this case, the sequence of marked strings obtained during each rotation is mapped by the observer to the string $l_{c,2}l_{c,3}l_{c,4}l_{c,1}$. At this point the entire process can be iterated. By searching and removing a new symbol a, and then again a b, and again a c, until the marked string $o_lX_ao_r$, from the target language is reached (the string obtained when all symbols a, b and c, have been deleted from the input marked string). Notice that at each step the current marked string is spliced with a string from the initial language.

This explanation shows that all strings from the language $\{o_lwo_r \mid w \in \{a,b,c\}^+, |w|_a = |w|_b = |w|_c\}$ can indeed be accepted by Ω. The fact that only such strings can be accepted is guaranteed by the particular form of sequences accepted by the decider in combination with the very specific form of the observed strings leading to such a sequence. □

5.3.3 Universality

Following the idea used in the proof of Theorem 5.6, it is possible to prove that SRs are universal. In informal words, this means that *it is possible to simulate an accepting Turing machine by observing, with a simple observer, the evolution of a simple splicing system.*

Theorem 5.7 *For each RE language L over the alphabet A there exists an SR Ω using a splicing scheme $\nu \in FIN_{H_4}$, such that Ω accepts the language $\{o_l'wo_r' \mid w \in L\}$, with $o_l', o_r' \notin A$.*

Proof. Any SR of the specified type can be simulated by a Turing machine. Thus we only show that, for any Turing machine, there can be constructed an equivalent SR system Ω composed of a splicing system using a finite set of rules, a finite initial language and target marked language and by an observer and a decider that are finite state machines. In this proof we use off-line Turing machines with only a single combined input/working tape. The set δ of transitions is composed of elements of the form $Q \times A \rightarrow Q \times A \times \{R, L\}$, where Q is the set of states, A the tape alphabet, and R or L denotes a move to the right or left, respectively. Usually, an input word

is accepted, if and only if, the Turing machine stops in a state that belongs to $F \subseteq Q$ of final states. Without loss of generality, we suppose that the machine M accepts the input, if and only if it reaches a configuration where the tape is entirely empty, and M is in a state that belongs to F. The initial state of M is $q_0 \in Q$. The special letter $\Box \in A$ denotes an empty tape cell.

We construct an SR Ω simulating M. Before giving the formal details, we outline the basic idea of the proof. The input string to the Turing machine is inserted as input marked string to the SR Ω, delimited by two external markers o'_l, o'_r. This does not restrict the generality of the theorem, because these two symbols could be added to any input string in two initializing steps by the SR. However, we want to spare ourselves the technical details of this.

Initially, an arbitrary number of empty tape cells $\Box$ is added to the left and to the right of the input marked string. When this phase is terminated, some new markers o_l and o_r are added to the left and right of the produced marked string; starting from this step, the transitions of the Turing machine M are simulated on the current marked string; the marked string contains, at any time, the content of the tape of M, the current state and the position of the head of M over the tape. To read the entire tape of M the marked string is rotated using a procedure very similar to the one described in the proof of Theorem 5.6; like there, the observer can check that the rotations are done in a correct way. The computation of Ω stops when the target marked string is reached, that is when a marked string representing an empty tape is reached.

Formally, the SR $\Omega = (\nu, O, D, L, L_t)$ is constructed in the following way. The H scheme $\nu = (V, R)$ has alphabet (we denote $A' = A \cup (A \times Q)$)

$$\begin{aligned} V &= \{o_r, o_l, o'_r, o'_l, X_1, X_2, \cdots, X_{12}\} \\ &\cup A' \cup \{X_e, X'_e \mid e \in A'\}. \end{aligned}$$

The splicing rules present in R are divided in groups, according to their use.

Initialization

$r_1 : \{o'_l(a, q_0)\#X_1\$o'_l a\#,\ a \in (A - \{\Box\})\};$
$r_2 : \{\#o'_r\$X_2\#\Box o'_r\};$
$r_3 : \{o'_l\Box\#X_3\$o'_l\#\};$
$r_4 : \{\#o'_r\$X_4\#o_r\};$
$r_5 : \{o_l\#X_5\$o'_l\#\};$

Rotations

$r_6 : \{a\#eo_r\$X_6\#X_e o_r,\ e \in A', a \in A\};$
$r_7 : \{o_l X'_e\#X_7\$o_l\#f,\ e, f \in A'\};$
$r_8 : \{a\#X_e o_r\$X_8\#o_r,\ e \in A', a \in A\};$
$r_9 : \{o_l e\#X_9\$o_l X'_e\#f,\ e, f \in A'\};$

Transitions

r_{10} : $\{\#(a,q_1)bo_r\$X_{10}\#c(b,q_2)o_r,\ q_1,q_2 \in Q, a,b,c \in A,$
$(q_1,a) \to (q_2,c,R) \in \delta\ \}$;
r_{11} : $\{\#b(a,q_1)do_r\$X_{11}\#(b,q_2)cdo_r,\ q_1,q_2 \in Q, a,b,c,d \in A,$
$(q_1,a) \to (q_2,c,L) \in \delta\}$;

Halting phase

r_{12} : $\{o_l\#\$X_{12}\#o_r\}$.

The initial language L is the finite language containing the strings used by the mentioned splicing rules; in particular,

$$\begin{aligned} L &= \{o_l'(a,q_0)X_1 \mid a \in (A - \{\square\})\} \\ &\cup \{X_2o_r', o_l'\square X_3, X_4o_r, o_lX_5, X_8o_r, X_{12}o_r\} \\ &\cup \{X_6X_eo_r, o_lX_e'X_7, o_leX_9 \mid e \in A'\} \\ &\cup \{X_{10}c(b,q_2)o_r \mid q_2 \in Q, c,b \in A\} \\ &\cup \{X_{11}(b,q_2)cdo_r \mid b,c,d \in A, q_2 \in Q\}. \end{aligned}$$

The target marked language is $L_t = \{o_lo_r\}$. The observer has input alphabet V and output alphabet $\Sigma = \{l_0, l_1, \cdots, l_8, l_f, \perp\}$.

The mapping implemented by the observer is

$$O(w) = \begin{cases} l_0 & \text{if } w \in o_l'(A - \{\square\})^+o_r', \\ l_1 & \text{if } w \in o_l'(a,q_0)(A - \{\square\})^*o_r',\ a \in (A - \{\square\}), \\ l_2 & \text{if } w \in o_l'(A' - \{\square\})^+(\square)^+o_r', \\ l_3 & \text{if } w \in o_l'(\square)^+(A' - \{\square\})^+(\square)^+o_r', \\ l_4 & \text{if } w \in \{o_l'w'o_r \mid w' \in (\square)^*(A' - \{\square\})^+(\square)^*, |w'| \geq 3\}, \\ l_5 & \text{if } w \in (o_l(A')^+o_r - \{w \mid w \in E\}), \\ l_6 & \text{if } w \in o_l(A')^*X_eo_r,\ e \in A', \\ l_7 & \text{if } w \in o_lX_e'(A')^*X_eo_r,\ e \in A', \\ l_8 & \text{if } w \in o_lX_e'(A')^*o_r,\ e \in A', \\ l_f & \text{if } w \in E, \\ \perp & \text{else.} \end{cases}$$

where $E = o_l(\square)^*(\square,q)(\square)^+o_r \cup o_l(\square)^+(\square,q)(\square)^*o_r \cup o_l(\square)^+(\square,q)(\square)^+o_r$, $q \in Q$.

The decider is a finite state automaton, with input alphabet Σ that accepts the regular language $E_1 \cup E_2$, where

$$\begin{aligned} E_1 &= l_0l_1(l_2)^+(l_3)^*l_4(l_5 \cup l_5l_5)(l_6l_7l_8(l_5 \cup l_5l_5))^*l_f, \\ E_2 &= l_0l_1l_4(l_5 \cup l_5l_5)(l_6l_7l_8(l_5 \cup l_5l_5))^*l_f. \end{aligned}$$

The main point of the proof is to show that, given an input marked string w, at least one of its (observed) evolutions is of the type accepted by the decider if and only if the string w is accepted by the Turing machine M.

We now describe the (observed) evolution of a correct input marked string; from this, we believe it will be clear that non correct strings will not have an evolution of the kind accepted by the decider, and, therefore will not be accepted by the SR Ω. The reader can compare the observed evolution of the input marked string with the language accepted by the decider.

Actually we introduce in the system Ω not the string w but a string of the type $o'_l w o'_r$ where o'_l, o'_r are left and right delimiters. In general the input marked string will be of the type $o'_l(A - \{\Box\})^+ o'_r$ and is mapped by the observer to l_0. The pairs in $Q \times A$ are used to indicate in the string the state and the position of the head of M. Initially the head is positioned on the leftmost symbol of the input marked string, starting in state q_0 (by using a rule in r_1); the obtained marked string is of the kind $o'_l(a, q_0)(A - \{\Box\})^* o'_r$, $a \in (A - \{\Box\})$ mapped to l_1 by the observer.

Then empty cells $\Box$ are added to the right and to the left of the marked string using rules in r_2 and in r_3, respectively. The marked string obtained at the end of this phase will be of the kind $o'_l(\Box)^+(A' - \{\Box\})^+(\Box)^+ o'_r$ mapped to l_3 by the observer. This phase is optional, and therefore the language of the decider is described by the union of E_1 where the adding of spaces is used and E_2, where no spaces are added, i.e., l_2 and l_3 are missing.

Then, by using rules in r_4 and in r_5 the delimiters o'_l and o'_r are changed into o_l and o_r, respectively. When a rule in r_4 is applied, the marked string obtained is of the kind $o'_l w' o_r, w' \in (\Box)^*(A' - \{\Box\})^+(\Box)^*$ mapped to l_4 if the size of the string w' (possibly, including empty cells) is at least of 3 symbols; this condition is useful during the following phases of rotations and does not imply a loss of generality.

When a rule in r_5 is applied, also o'_l is removed and the marked string obtained is mapped to l_5 by the observer. At this point there are two possible ways to continue the computation. In the first case the symbol indicating the head of M, (a, q_1) is exactly one symbol away from o_r; then a splicing rule in r_{10} or in r_{11} is applied. The one symbol left between the symbol representing the head and the delimiter o_r is useful in case of the simulation of a right transition. The rule sets r_{10} and r_{11}correspond to transitions moving right and left, respectively.

Once a transition is simulated, the obtained marked string is again of the type mapped to l_5 by the observer (this is why it is possible to have in the language of the decider the substring $l_5 l_5$). At any rate it is not possible to have immediately another transition after a transition, because the symbol corresponding to the head of M is moved. At least one rotation must be first executed.

In case the symbol representing the head of M is not exactly one symbol away from o_r, then the marked string is rotated until this condition is not true any more. The rotation of one symbol in the string (i.e., moving the symbol present to the left of o_r, to the immediate right of o_l) is done by applying, in this order, splicing rules from r_6, r_7, r_8 and from r_9. The marked

strings obtained during this phase are mapped by the observer to l_6, l_7, l_8 and finally l_5. At the end of a rotation a transition can be simulated; more consecutive rotations can be done until the necessary condition to simulate a transition is reached. This explains why $(l_6 l_7 l_8 (l_5 \cup l_5 l_5))^*$ forms part of the decider language.

When, after a transition, the marked string obtained represents the empty tape of M, then the computation of the SR can stop. The marked strings representing an empty tape are the ones in the language E and they are mapped by the observer to l_f. After the observer has output l_f, the splicing rule in r_{12} can be applied and the unique string in the target marked language $o_l o_r$ can be reached. If the rule in r_{12} is applied before the observer outputs l_f, then the entire evolution is not accepted by the decider. Notice that during the entire computation the marked string can be spliced only with a string from the initial language.

From the above explanation, it follows that, an input marked string written in the form $o'_l w o'_r$ is accepted by Ω, if and only if, w is accepted by the Turing machine M. □

5.4 Final Remarks

The idea of computing by observing the evolution is initially introduced in [53] in the framework of membrane computing. The results and definition presented in Section 5.1 are extracted from this paper. The generalization of the evolution/observation framework to the formal language area has been proposed and investigated in [54]. The definitions and results of Section 5.2 are based on this paper.

The first idea of using the evolution/observation framework to construct accepting devices is presented in [55]. There, a recognizing device based on the evolution/observation idea is defined and investigated. In particular, the device presented is based on the following strategy: a word is accepted if the (observed) evolution of a certain system starting from this input follows a regular pattern. The following result is shown: checking if the (observed) evolution of a context-free system follows a regular pattern is enough to accept every recursively enumerable languages. On the other hand, if we observe the evolution of systems using very simple rules (of the kind $a \to b$), then it is possible to accept exactly the class of context-sensitive languages.

In [56] has been shown how powerful accepting devices can be realized by adding an observer to classical string-rewriting systems.

The first application of the evolution/observation idea to other fields of natural computing is presented in [10] where the authors investigate a computing device where the computation is based on observing the evolution of a sticker system, a formal model of DNA self-assembly. In [10] it is

shown that even regular simple sticker systems (whose generative power is subregular) become universal when considered in this new framework.

A very particular kind of observation is the one observing ("counting") only the number of steps that a (membrane) system does from the beginning to the end of a halting computation and taking this number as the result of the computation. This idea has been investigated in [49] and in [89].

Section 5.3 is based on the paper [51]. The standard notions of splicing systems as well as the result that the family of languages generated by finite splicing systems is strictly included in the family of regular languages can be found in the specific chapter of the monograph [122]. Section 5.3 is motivated from the fact that several methods for observing the dynamics of single DNA molecules are present in literature. A survey of these methods can be found in [98]. Usually, these techniques can be used to observe only three different colors in fluorescent microscope, but it is possible to obtain more colors by *multiplexing*, as suggested by [97].

We also have mentioned quantum dots as a new way to mark (and then, to observe) single DNA molecules; a very recent review on the use of quantum dots in vivo imaging can be found in [107].

Chapter 6

Concluding Remarks and Open Problems

Natural computing is a recent and fast growing research area that is developing in several different branches; a first one investigates the construction of computing devices by using biochemical substances like, for instance, DNA strands; a second branch tries to investigate biological processes by using computational methods, like, for instance, the analysis of the gene assembly process in ciliates; a third branch is the one that investigates new computing devices inspired by the mechanisms and by the structure of biological processes and organisms. In this line of research we can find several bio-inspired models and among them membrane systems, computational devices inspired by the functioning and the structure of living cells. Since their introduction in 1998, membrane systems have been the subject of a fast evolving interest and they are still intensively investigated.

In this Thesis we have shown how membrane systems can be used to model biological processes taking place in living cells and how, at the same time, the interaction with biology brings new ideas, new models, and new bio-inspired paradigms of computation.

The Thesis can be seen as divided in four main parts; Initially, the evolution-communication model is presented. This model is biologically inspired and it has been shown to be useful also from a biological point of view: indeed, a part of the Thesis is devoted to the modeling of biological processes by using this model. On the other hand, the interaction and the feedback with biology brings new bio-inspired computational paradigms; this last point is treated in the last part of the Thesis where time-free P systems and a new paradigm of computation based on evolution and observation have been presented.

We devote this final chapter to an overview of these four parts, outlining the main open problems and providing several lines for further researches.

6.1 Evolution-Communication Models

Evolution-communication P systems (EC P systems) are based on the composition of two main ingredients: evolution rules represented by multiset rewriting rules and symport/antiport rules represented by uni-diretional/bi-diretional synchronized exchanges of objects across a membrane. EC P systems have been recalled in Chapter 2.

In Section 2.1.1 we have shown that EC P systems are universal when using catalytic rules (with only one catalyst), symport/antiport rules of weight one and using the mixed approach for computation, based on a mixed application of evolution rules and symport/antiport rules.

The computation for an EC P system can be also defined according to two other approaches proposed in [46]:

1. the *evolutive approach*:
   the simple evolution rules of the system have priority over the symport/antiport rules (in biological terms: an organism evolves by itself as much as possible (*evolution*), until it needs something from others organisms (*communication*));

2. the *communicative approach*:
   the symport/antiport rules of the system have priority over the simple evolution rules (in biological terms: an organism tries to *communicate* with the others organisms and only when it cannot communicate anymore, then it starts (or continue) its *evolution*).

The universality proof for EC P systems, using two membranes, non-cooperative evolution rules, and symport/antiport rules of weight one, given in [6] works also for systems using the communicative approach. It is an open problem which are the ingredients necessary to obtain universality for EC P systems working with the evolutive approach.

On the other hand, there could be other ways of interleaving evolution and communication, for instance, the $= k$ mode and t mode, as used in *grammar systems*, [65].

It is also possible to consider EC P systems using only one membrane (all the universality have been obtained by using at least two membranes) possibly with an infinite environment (in the definition considered here the environment is empty at the beginning of the computation). Moreover, in every universality proof symport rules are also used; can they be avoided?

It would be also interesting to discover which restrictions can be imposed on the rewriting rules or/and on the symport/antiport rules to get an infinite hierarchies on the number of membranes.

There are other two models that can be considered inspired or particular cases of the evolution-communication model; the one with symport/antiport

of rules and the one using proton pumping rules, inspired by the proton pumping phenomena present in living cells; these two models have been presented in Sections 2.2 and 4.1, respectively.

The main feature of P systems with symport/antiport of rules (CR P systems) consists in the fact that objects cannot be moved while evolution rules can be moved by using symport/antiport rules. Several problems have been left open for this model. For instance, the universality has been proved by considering antiport rules of weight two and no symport rules (Theorem 2.6). What can we obtain by considering only antiport rules of weight one? Is this class of systems still universal? (From Example 2.1 we know that such systems can generate the Parikh set of non-semilinear languages).

Moreover, what can CR P systems generate by using only symport of rules?

On the other hand, in the case of non-cooperative evolution rules, is it possible to get the family of Parikh images of ET0L languages using antiport rules of bounded weight and no priority? We recall that, in the proof presented in Section 2.2, the bound on the weight of the antiport rules used is obtained at the price of using priorities among the communication rules.

In [78] using non-cooperative evolution rules, the systems considered generate exactly the family of Parikh images of ET0L languages; however, in that case, the computation is divided into two substeps that are applied in an interleaved way: first, all possible communication rules are applied, and then, the evolution rules are applied, in a non-deterministic maximally parallel manner.

Is it possible to prove the same equality for CR P systems? Actually this seems hard because of the mixed application of communication and evolution rules. Moreover, in [78] the universality is obtained by using only one pure catalyst. Is it possible to obtain the same result in the case of CR P systems?

It would be also interesting to consider two other basic types of CR P systems: the first type should use multisets of rules; what happens if we do not consider the occurrence of rules in the set sense but in a multiset sense? (i.e., more than one copy of a rule can be present in a region); the second type should use strings instead of symbol-objects.

A more general suggestion is to use CR P systems considering the restrictions already investigated for classical P systems with symport/antiport rules and for evolution-communication P systems (for instance, the evolutive and communicative approach).

A proton pumping P system is an evolution–communication P system with a set of special objects called protons that are never created and never destroyed and where only antiports that can exchange some symbol-objects (also protons among them) for a single proton are admitted (such particular

antiports are called proton pumping rules). Universality of proton pumping P systems has been shown by using symport and proton pumping rules of weight one, non-cooperative evolution rules, three membranes and unbounded number of protons, (Theorem 4.1).

This result has been improved in [8]. There, it has been shown that universality can be obtained by using only one proton and two membranes.

Notice that, in most of the constructions given for EC and for proton-pumping P systems, multiset rewriting is used in all the regions. Is this feature really necessary to get universality?

On the other hand, it would be interesting to characterize the generative power of restricted proton pumping P systems where, for instance, the only uniport rules allowed are uniport rules of protons. A technical open problem concerns the number of protons necessary to get universality for time-free proton pumping P systems. In [8] four protons are used: are they necessary to get universality or their number can be decreased?

6.2 Modeling Biological Processes

In Chapter 3 we have presented a probabilistic simulator of P systems recently implemented and we have shown how various (simple) cellular processes can be modeled in the P systems framework and then simulated using the software. Initially we have recalled the main features of the simulator, showing its working and presenting the (strange and interesting) result of the simulation of a very simple probabilistic P system. In this respect, we have suggested the existence of a possible link between the emergence of life studied in biology and some kind of probabilistic P systems. We also believe that a possible fruitful research topic is to study which are the fields (human life, economy, cell biology, etc.) that can be modeled by (probabilistic) P systems and then simulated on a computer; we have discussed only biological applications but we believe that the P systems framework can be applied in many other different fields.

In Section 3.3 we have shown how to model some biological processes in the P systems framework and, in particular, we have studied two important biological processes: the (final step of) respiration in *Escherichia coli* bacteria and the interaction between respiration and photosynthesis in cyanobacteria, considering the consumption of oxygen and the accumulation of protons in the environment.

We have shown how to translate these biological processes in the mathematical framework of P systems and then how to obtain results relevant from a biological point of view.

It would be interesting to model other biological processes with more biological details considering the concept of affinity introduced in the simulator by the so-called probability to win and the concept of availability of a rule,

modeled in the software by the probability of a rule to be available; in particular, it would be very useful to add to the simulator the ability to change, in run-time, the values, of some of the biological parameters considered (like the affinity of an enzyme, that, for example in *Escherichia coli*, changes according to the concentration of molecular oxygen in the substrate).

Concerning the modeling of the process of respiration in *Escherichia coli* (considered in Section 3.3.1), in future should be considered processes where the enzymes *cytochrome bo* and *cytochrome bd* oxydase are involved in the following interesting situations:

1. *Decreasing of oxygen concentration:* The oxygen concentration decreases from saturation toward zero. In this case, both the reactions (rules) can be applied but the activity rates of the two enzymes change, according to their affinities.

2. *Increasing of oxygen concentration:* It is the dual case of (1): the oxygen concentration increases from zero toward saturation.

3. *Constant oxygen concentration:* In this case the oxygen concentration is maintained constant (at a chosen value, between 10% and 100% oxygen saturation). Moreover, the activity rate of the two enzymes is fixed, according to the chosen concentration of oxygen, and the system is not closed but feed in with oxygen.

Finally, a possible and very interesting experiment could be the measurement, by using the simulator, of the activity rate and of the quantity used for each enzyme and, based on previous experimental knowledge, the determination of either oxygen consumption and/or oxygen concentration in the growing medium. Especially this last application could be of real practical interest both in academic studies and in bioindustrial activities, because, while oxygen measurements are usually carried out (both in academia and industry) enzymes activity (or enzymes quantity) cannot be measured in intact cells. These measurements are very laborious and time consuming (for this reason the on-line control of bioindustrial processes is practically impossible at this level).

The on-line knowledge of enzymes activity and enzymes quantity in bioindustrial processes can enable the on-line optimization (with respect to costs and production of useful chemicals) of such processes.

In the considered interaction between respiration and photosynthesis in cyanobacteria (Section 3.3.3) one could add more details to the system, considering also other kinds of interactions existing between the two membranes, and, at the same time, the presence of regulatory mechanisms that are able to change the activity rate of the photosynthesis and respiration processes.

Moreover, it would be interesting to verify the stability of the system when it becomes *open* to the interaction with the environment (the introduction of oxygen or carbon dioxide from the environment).

As a more general line of research we want to mention that *timed P systems* – P systems with time of execution associated to the rules – could turn out to be very useful for modeling biological processes composed by biochemical reactions having different times of completion (that is an usual situation in biology).

Gh. Păun suggested in [119] several "intrinsic features" that makes membrane systems naturally attractive for applications, especially in biology. These features can be represented by the following keywords: *distribution* (with the system-part interaction, emergent behavior, non-linearly resulting from the composition of local behaviors), *discrete mathematics* (continuous mathematics, for instance differential equations, have been very much used in physics and related fields, but they cannot be used to model more than local processes in biology because of the complexity of the processes and, in many cases, their imprecise character), *algorithmiticity* (membrane systems are computing devices similar to Turing machines and then easy to simulate on a computer), *scalability/extensibility* (is the main difficulty of using differential equations in biology), *transparency* (multiset rewriting rules are written exactly like chemical reactions, then easy to understand), *parallelism* (typical of biology), *non-determinism* (it makes more flexible the way of "programming" the evolution of a system), *communication* (biology is in fact rich of many different ways and many still unknown way of communication, very efficient, in contrast to the situation present in classical computers, where the cost of communication increases prohibitively when the number of processors increases).

So, in some sense, membrane systems were born with features typical of biological processes and then might be much easier to try to use them for modeling biological processes than forcing models introduced in other scientific areas.

As mentioned in the final remarks of Chapter 3, there are already papers where scientific data of well-known biological experiments have been confirmed by computer simulations of membrane systems models. The next step should be to use the model (and the simulators) to help biologists, for instance, by solving biological open problems already stated in other terms and other frameworks. On the other hand, it should be also possible to discover biological properties in a static way, without running a simulator, possibly by using the large number of mathematical results obtained for the various models of membrane systems.

Concluding, we can say that the investigations presented in Chapter 3 are somewhat preliminaries, but the progresses are obvious and there are several reasons to make us optimist about the future of this research line.

6.3 Time-Free P Systems

In Section 4.2 time-free P systems have been presented. Time-free P systems are, essentially, systems that produce always the same result independently of the time of execution of the rules. A basic question is to decide whether an arbitrary P system is time-free. Unfortunately, for bi-stable catalytic P systems this problem is undecidable (Theorem 4.6). On the other hand, Theorem 4.2 says that every P system using only non-cooperative evolution rules is time-free; this is not true for catalytic P systems as proved in Theorem 4.3, but it is an open problem whether there is an algorithm to decide the time-freeness of a catalytic P system. It is also not known whether time-freeness is decidable for P systems with signaling-rules.

A more general problem is to find a class C of computational devices based on the P systems framework with the following properties:

- for each device m in C is possible to find (in an "easy" way) a time-free machine m' in C equivalent to m (i.e., m and m' produce the same output);
- the class C has "sufficient" computational power (that means sufficient cooperation);
- the class C has "sufficient" parallelism (to avoid sequential "slow" time-free machines; on the other hand, how to measure in a formal way the degree of parallelism?);
- it is possible to decide (and in an "easy" way) if an arbitrary machine in C is time-free.

The discovering of a class with these properties can open a new way to work with (asynchronous) parallel computing devices. The class of P systems using only catalytic rules (or bi-stable catalytic rules, with few bi-stable catalysts) is a possible candidate.

While it is known that time-free bi-stable catalytic P systems are universal (Theorem 4.5), it is an interesting open problem whether also catalytic time-free P systems are universal (this is an even more interesting problem since standard catalytic P systems – with each rule executed in one time-unit – are universal with two catalysts, [75]). Maybe time-free catalytic P systems constitute a non-universal class that would prove that "time is important".

In Section 1.3.2 universality has been shown for P systems using symport/antiport rules of weight one and three membranes. Theorem 4.8 shows that universality can be also obtained for time-free P systems using antiports of weight at most two and one membrane. Can universality be obtained also in case of time-free systems by using symports and antiports of weight at most one (at the cost, for instance, of increasing the number of membranes)?

Another interesting way to investigate time-free P systems is to impose certain conditions on the time of execution of the rules (this class has been referred to as *partially time-free P systems* in [58]). How to formalize such conditions? Which are the conditions that should be used?

Moreover, something that should be addressed concerns new methods to synchronize the computation in time-free P systems. In Section 4.2 we have presented signal-promoters, (bi-stable) catalysts, symport/antiport rules. Are there other synchronizing methods (maybe inspired from cell functioning)? It is also possible to add other parameters to construct a more realistic model of P systems; for instance, it would be interesting to associate to each rule a time of delay that indicates the time to wait before a rule is started and then to study a class of delay-free systems. This might model the fact that, sometimes, chemical rules are not started immediately, even in presence of the necessary chemicals.

6.4 Evolution and Observation

Evolution and observation is a general way to look at computation based on the idea that a computing device can be constructed using two less powerful systems: the first one, which is a biological system, simply "lives", passing from one state (configuration) to the next one, producing some behavior; the second system (observer) is placed outside and watches the biological system.

The observer, following a set of rules, translates the behavior of the biological system into a more readable output. In this way, the pair composed by the biological system and the observer can be considered a computing device.

In Section 5.1 it has been shown how the evolution/observation framework can be applied to the membrane computing framework; in particular P/O systems have been presented.

P/O systems are composed by a membrane system with symbol-objects (that constitutes the biological system) and by a multiset finite automaton (representing the observer). As usually defined, with this membrane system a computation (sequence of configurations) can be associated that can be halting or non-halting. The observer then transforms this sequence into a string, mapping each configuration into a symbol of a finite alphabet, possibly ignoring the computation, if it contains undesired configurations. In this way, a P/O system produces a language of strings. A language of finite words is obtained if the computations of the membrane system considered are the halting ones, otherwise the language obtained is composed of infinite words.

For the case of P/O systems with DMFAD as observers, Theorem 5.3 shows that already rather little computational power in both components

leads to universality, here in the strong sense of strings, not numbers. This result demonstrates that the approach of "joining", in the way described, two simple models of computation, can give us great (and unexpected) generative power.

Plenty of well-motivated new topics in dynamics of membrane systems suggest themselves for investigation: from a biological/chemical point of view it would be very useful to study P/O systems with a restricted observer. In Theorem 5.3 the MFA used is without any restriction; in particular, such MFAs are able to check the presence of zero, one or more symbol-objects of the same kind in a configuration; this observer is not so realistic looking at what is realizable in a laboratory where normally only the presence or absence of a substance can be detected in an easy way. This suggests using observers which can only check for the presence or absence of an object, but not calculate with its numbers. Demanding that on every path each letter is read at most once would provide an appropriate restriction of an MFA. Then probably also the membrane system's structure would become more important.

Moreover, many other restrictions can be used, on the side of the membrane system as well as on the side of the observer. Characterizing the classes of languages generated by such systems is the most natural task; the subregular and universal variants of this treatise provide some kind of lower and upper bound for the search of interesting classes.

In this respect, the proof of Theorem 5.2 leaves a somewhat bitter taste, because it seems to some degree just like putting the entire automaton itself into the membrane. Additionally it seems unlikely that different types of chemicals in arbitrary numbers with the necessary properties could be found to realize such a system. Therefore, it is natural to ask whether Theorem 5.2 is still true when the number of objects in the system is bounded. In this context it should be noted that – parallel to regular languages of arbitrary state complexity – for every positive integer k there are behaviors which can be accepted by Büchi automata with k states, but not by any such device with less than k states.

On the other hand, a more general question is whether for an arbitrary membrane system the property of being quasi-conservative (i.e., with a finite number of configurations) is decidable.

In Section 5.2 it has been shown how the evolution/observation framework can be applied to the general theory of computation, in particular to formal language theory, introducing and studying the class of grammar/observer (G/O) systems, where a grammar plays the role of the system and an automaton – be it a finite automaton or even a Turing machine – plays the role of the observer.

Three types (initial, free, always-writing) of G/O systems have been presented. Universality can be obtained, for initial G/O systems, by using a context-free grammar observed by a finite state automaton (Theorem 5.4)

and for free G/O systems by using even a commutative context-free grammar observed by a finite state automaton (Theorem 5.5).

The results appear to be a bit surprising taking into account the very weak power of the involved components. Therefore it is natural the comparison of G/O systems with more common computing devices, for instance, Turing machines. We can consider Turing machines (still universal) having a combined input/working tape, a finite state control and an output tape, on which they can only write. This general setup resembles very much that of the G/O systems. In the sentential form of the context-free grammar one symbol can be rewritten in any step – if during almost the entire computation the sentential form consists of non-terminals as in Theorem 5.4, then, at any time basically every symbol can be rewritten. The finite automaton generates an output depending on the content of the sentential form, just like the finite state control does depending on the symbol read from the input/working tape. So what are the differences? For one thing the interaction between control and workspace is greatly decreased: the automaton only writes the output, but has no direct influence on the changes in the sentential form. Further it always starts in the same initial state and therefore cannot remember anything from the earlier computation steps. All this is compensated by the ability to read the entire workspace in every step. Thus the automaton can on the one hand, have some memory of earlier steps there, and, on the other hand, based on this memory, it can check, whether the grammar has made exactly the changes which in a Turing machine the control would effect directly.

Using only regular grammars would mean that basically nothing that is written on the workspace can be rewritten. Therefore, the conjecture is that G/O systems with both the grammar and the observer regular should themselves only generate regular languages. If this proves to be true, then the same might hold for linear grammars, whose sentential forms also contain at most one non-terminal at a time.

There are many other interesting questions that remain to be answered about the G/O system architecture. The most interesting open problem left in this thesis is the exact characterization of the languages generated by always writing G/O systems: are they the context-sensitive languages? The conjecture is that this is not the case. To prove that, one might search for a language whose generation by an LBA inevitably requires the entire (linearly bounded) space and at the same time heavily reuses all this space. Actually, even a lower bound of $n + 1$ steps for the generation of a word of length n by an always writing G/O system would suffice. For example, the language over the alphabet $\{a\}$ consisting of all words of prime length appears as a good candidate for this.

If not all the context-sensitive languages can be generated, which are the properties of this language family? Perhaps it equals some family known from regulated rewriting.

Further, we have only marginally treated all the variants where all observed behaviors must be considered and $\perp$ is not used. While we have mentioned that also these generate some non-recursive languages in the cases discussed in Sections 5.2.4 and 5.2.5, it seems doubtful that they are universal.

The evolution/observation framework can be also extended to other areas of natural computing. For instance, in Section 5.3, we have presented how, using this framework, an accepting device (splicing recognizer) can be constructed based on DNA splicing.

In particular it has been shown that observing the evolution of only one marked DNA strand by means of a simple observer and decider can be a powerful tool which theoretically is sufficient to simulate a Turing machine. The components involved are rather simple (finite splicing and finite state automata), hence the computational power seems to stem mainly from the ability to observe, in real-time, the changes (the dynamics) of a particular (marked) DNA strand, under the action of restriction enzymes.

The proposed approach suggests several problems, if this were to be implemented in practice.

For instance, the process of observation as defined is non-deterministic; that is the marked DNA strand inputed is accepted if at least one of its observed evolution follows an expected pattern, while there might be several possible evolutions of this DNA strand since there might be several different ways to splice the strand. From a practical point of view this would require several copies of the same input DNA strand, each copy marked with a different "color". The observer should follow, separately, the evolution of each one of these strands. This theoretically requires an unbounded number of copies of DNA strands, each one marked with a different color. In practice, however, using many marked copies may increase the chance to obtain the needed evolutions.

A possible way to implement this might be the use of the multiplexing technique introduced in [97] used to mark several molecules, each one with a different "color". Another way may be marking the strands with quantum dots, [40]. However, none of these techniques have been used for observing splicing and the problems that may arise during the implementation may be numerous.

Further theoretical investigations may provide better solutions if it can be shown that by increasing the complexity of the observer and the decider, then a ("more") deterministic way of generating the splicing evolutions can be employed. We recall that in the model presented the observers and deciders are with very low computational power, i.e., finite state automata.

Another problem that needs to be taken care of if implementing an SR is the real-time observation: in the model presented it is supposed that the observer is able to catch, in the molecular soup, every single change of the marked DNA strand. In practice, it is very questionable whether every step

of the evolution can be observed. It should be assumed that only some particular types of changes, within a certain time-interval can be observed (see [98]). Therefore, another variant of SR needs to be, at least theoretically, investigated in which an observer with "realistic" limitations on the ability of observation is considered. For instance, the observer might be able to watch only a window or a scattered subword of the entire evolution.

On the other hand, universal computational power has been obtained for SRs by using an H scheme of radius 4 (Theorem 5.7). The conjecture is that it is possible to decrease the radius, hence the question arises of what is the minimum radius that provides universal computation.

It remains also to investigate SRs using simpler and more restricted variants of H schemes, like the ones with simple splicing, [106]. Notice that, from a pure theoretical point of view, the observer and decider could be joined in an unique finite state automaton, which may provide a better framework for theoretical investigation. We have preferred to leave the two "devices" of observer and decider separated since this situation can be envisioned to be closer to reality.

Moreover, one can interpret a given H scheme with an observer as a device computing a function, by considering as input the input marked string, and as output its (observed) evolution. What kind of functions can be computed in this way?

These are only a few of the possible directions of investigation that the presented approach suggests.

Bibliography

[1] L.M. Adleman, Molecular Computation of Solutions to Combinatorial Problems. *Science*, 266 (1994), pp. 1021–1024.

[2] L.M. Adleman, P.W.K. Rothemund, S. Roweiss, E. Winfree, On Applying Molecular Computation to Data Encryption Standard. In [22], pp. 28–48.

[3] B. Alberts, *Essential Cell Biology. An Introduction to the Molecular Biology of the Cell.* Garland Publ. Inc., New York, London, 1998.

[4] S. Alexeeva, K.J. Hellingwerf, M.J. Teixeira de Mattos, Quantitative Assessment of Oxygen Availability: Perceived Aerobiosis and Its Effect on Flux Distribution in the Respiratory Chain of Escherichia coli. *Journal of Bacteriology*, 184 (2002), pp. 1402–1406.

[5] S. Alexeeva, B. de Kort, G. Sawers, K.J. Hellingwerf, M.J. Teixeira de Mattos, Effects of Limited Aeration and of the ArcAB System on Intermediary Pyruvate Catabolism in Escherichia coli. *Journal of Bacteriology*, 182 (2000), pp. 4934–4940.

[6] A. Alhazov, Minimizing Evolution-Communication P Systems and EC P Automata. In [57], pp. 23–31, and *New Generation Computing*, 22, 4 (2004), pp. 299–310.

[7] A. Alhazov, On Determinism of Evolution-Communication P Systems. In [120] and *Journal of Universal Computer Science*, 10, 5 (2004), pp. 502–508.

[8] A. Alhazov, Number of Protons/Bi-stable Catalysts and Membranes in P Systems. Time-Freeness. In [77], pp. 102–122.

[9] A. Alhazov, M. Cavaliere, Proton Pumping P Systems. In [102], pp. 70–88.

[10] A. Alhazov, M. Cavaliere, Computing by Observing Bio-Systems: The Case of Sticker Systems. In [72], pp. 1–13.

[11] A. Alhazov, M. Cavaliere, Evolution-Communication P Systems: Time-Freeness. In [84], pp. 1–18.

[12] A. Alhazov, R. Freund, Yu. Rogozhin, Computational Power of Symport/Antiport: History, Advances and Open Problems. In [77], pp. 44–78.

[13] A. Alhazov, M. Margenstern, V. Rogozhin, Yu. Rogozhin, S. Verlan, Communicative P Systems with Minimal Cooperation. In [105], pp. 161–177.

[14] A. Alhazov, C. Martín-Vide, Gh. Păun, eds., *Pre-Proceedings of the Workshop on Membrane Computing WMC-2003.* Tarragona, Spain, July 2003, Technical Report GRLMC 28/03, University of Tarragona, 2003.

[15] I.I. Ardelean, The Relevance of Cell Membranes for P Systems. General Aspects. *Fundamenta Informaticae*, 49, 1-3 (2002), pp. 35–43.

[16] I.I. Ardelean, Molecular Biology of Bacteria and Its Relevance for P Systems. In [123], pp. 1–18.

[17] I.I. Ardelean, D. Besozzi, M.H. Garzon, G. Mauri, S. Roy, P System Models for Mechanosensitive Channels. In [63], pp. 43–80.

[18] I.I. Ardelean, M. Cavaliere, Modeling Biological Processes by Using a Probabilistic P System Software. *Natural Computing*, 2-3 (2003), pp. 173–197.

[19] I.I. Ardelean, M. Cavaliere, D. Sburlan, Computing Using Signals: From Cells to P Systems. In [120], pp. 60–73, and *Soft Computing*, 9-9 (2005), pp. 631–639.

[20] R. Barish, P.W.K. Rothemund, E. Winfree, Algorithmic Self-Assembly of a Binary Counter Using DNA Tiles. In [43], pp. 196.

[21] S. Basu, D. Karig, R. Weiss, Engineering Signal Processing in Cells: Towards Molecular Concentration Band Detection. *Natural Computing*, 2-4 (2003), pp. 463–478.

[22] E. Baum, L. Landweber, eds., *DNA Based Computers, Proceedings Second Annual Meeting.* Vol. 44 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, AMS, 1996.

[23] E. Baum, R.J. Lipton, eds., *DNA Based Computers.* Vol. 27 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS, 1995.

[24] Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, Eh. Shapiro, DNA Molecule Provides a Computing Machine with Both Data and Fuel. *Proc. Nat. Acad. Sci. (PNAS)*, 100, 5 (2003), pp. 2191–2196.

[25] Y. Benenson, G. Binyamin, U. Ben-Dor, R. Adar, Eh. Shapiro, An Autonomous Molecular Computer for Logical Control of Gene Expression. *Nature*, 429 (2004), pp. 423–429.

[26] Y. Benenson, T. Paz-Elizur, R. Adar, Eh. Keinan, Z. Livneh, Eh. Shapiro, Programmable and Autonomous Computing Machine Made of Biomolecules. *Nature*, 414 (2001), pp. 430–434.

[27] C.H. Bennett, Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17 (1973), pp. 525–532.

[28] C.H. Bennett, Thermodynamics of Computation – a Review. *International Journal of Theoretical Physics*, 21 (1982), pp. 905–940.

[29] F. Bernardini, M. Gheorghe, On the Power of Mininal Symport/Antiport. In [14], pp. 72–83.

[30] F. Bernardini, M. Gheorghe, N. Krasnogor, R.C. Muniyandi, M.J. Pérez-Jiménez, F.J. Romero-Campero, On P Systems as a Modelling Tool for Biological Systems. In [77], pp. 193–213.

[31] F. Bernardini, V. Manca, P Systems with Boundary Rules. In [123], pp. 97–101.

[32] J. Berstel, J.E. Pin, *Infinite Words*, in preparation; draft at `http://www.liafa.jussieu.fr/~jep/Resumes/InfiniteWords.html`.

[33] D. Besozzi, *Computational and Modelling Power of P Systems*. Ph.D. Thesis, University of Milan, Milan, Italy, 2004.

[34] D. Besozzi, G. Mauri, C. Zandron, Hierarchies of Parallel Rewriting P Systems – A Survey. *New Generation Computing*, 22, 4 (2004), pp. 331–347.

[35] L. Bianco, F. Fontana, G. Franco, V. Manca, P Systems for Biological Dynamics. In [63], pp. 81–126.

[36] L. Bianco, F. Fontana, V. Manca, P Systems and the Modeling of Biochemical Oscillations. In [77], pp. 214–225.

[37] S. Bozapalidis, ed., *Proceedings of the 3rd International Conference on Developments in Language Theory*. Aristotle University of Thessaloniki, 1997.

[38] R.S. Braich, N. Chelyapov, C. Johnson, P.W.K. Rothemund, L.M. Adleman, Solution to a 20-Variable 3-SAT Problem on a DNA Computer. *Science*, 296-5567 (2002), pp. 499–502.

[39] R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, Membrane Systems with Marked Membranes. In preparation.

[40] M. Bruchez, M. Moronne, P. Gin, S. Weiss, A.P. Alavisatos, Semiconductor Nanocrystals as Fluorescent Biological Labels. *Science*, 281 (1998), pp. 2013-2016.

[41] C.S. Calude, Gh. Păun, *Computing with Cells and Atoms: An Introduction to Quantum, DNA and Membrane Computing.* Taylor and Francis, London, 2000.

[42] C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds., *Multiset Processing, Mathematical, Computer Science and Molecular Computing Points of View, WMP2000.* Curtea de Argeş, Romania, August 2000, LNCS, 2235, Springer-Verlag, Berlin, 2001.

[43] A. Carbone, M. Daley, L. Kari, I. McQuillan, N. Pierce, eds., *11th International Meeting on DNA Computing, DNA 11, Preliminary Proceedings.* University of Western-Ontario, Canada, June 2005.

[44] L. Cardelli, Brane Calculi. Interactions of Biological Membranes. In [67], pp. 257–278.

[45] L. Cardelli, Gh. Păun, An Universality Result for a (Mem)Brane Calculus Based on Mate/Drip Operations. In [82], pp. 75–94.

[46] M. Cavaliere, Evolution-Communication P Systems. In [123], pp. 134–145.

[47] M. Cavaliere, I.I. Ardelean, Modelling Respiration in Bacteria and Respiration/Photosynthesis Interaction in Cyanobacteria by Using a P System Simulator. In [63], pp. 129–158.

[48] M. Cavaliere, V. Deufemia, Further Results on Time-Free P Systems. *International Journal of Foundations of Computer Science*, to appear.

[49] M. Cavaliere, R. Freund, A. Leitsch, Gh. Păun, Event-Related Outputs of Computations in P Systems. In [84], pp. 108–122.

[50] M. Cavaliere, D. Genova, P Systems with Symport/Antiport of Rules. *Journal of Universal Computer Science*, 10-5 (2004), pp. 540–558.

[51] M. Cavaliere, N. Jonoska, P. Leupold, Recognizing DNA Splicing. In [43], pp. 6–16.

[52] M. Cavaliere, N. Jonoska, S. Yogev, R. Piran, Eh. Keinan, N.C. Seeman, Biomolecular Implementation of Computing Devices with Unbounded Memory. In [72], pp. 35–49.

[53] M. Cavaliere, P. Leupold, Evolution and Observation – A New Way to Look at Membrane Systems. In [102], pp. 70–87.

[54] M. Cavaliere, P. Leupold, Evolution and Observation–A Non-Standard Way to Generate Formal Languages. *Theoretical Computer Science*, 321, 2-3 (2004), pp. 233–248.

[55] M. Cavaliere, P. Leupold, Evolution and Observation – A Non-Standard Way to Accept Formal Languages. In [101], pp. 152–162.

[56] M. Cavaliere, P. Leupold, Observation of String-Rewriting Systems. *Fundamenta Informaticae*, to appear.

[57] M. Cavaliere, C. Martín-Vide, Gh. Păun, eds., *Proceedings First Brainstorming Week on Membrane Computing.* Tarragona, 2003, GRLMC Report 26/03, Rovira i Virgili University, 2003.

[58] M. Cavaliere, D. Sburlan, Time-Independent P Systems. In [105], pp. 239–258.

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

[60] H.L. Chen, A. Goel, Error Free Self-Assembly Using Error Prone Tiles. In [72], pp. 62–75.

[61] H.L. Chen, A. Goel, R. Schulman, E. Winfree, Error Correction for DNA Self-Assembly: Preventing Facet Nucleation. In [43], pp. 197.

[62] J. Chen, J. Reif, eds., *DNA Computing: 9th International Workshop on DNA Based Computers, DNA9.* Madison, USA, June 2003, LNCS 2943, Springer-Verlag, Berlin, 2004.

[63] G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez, eds., *Applications of Membrane Computing.* Springer-Verlag, Berlin, 2006.

[64] A. Condon, G. Rozenberg, eds., *DNA Computing: 6th International Workshop on DNA-Based Computers, DNA6.* Leiden, The Netherlands, June 2000, LNCS 2054, Springer-Verlag, Berlin, 2000.

[65] E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun, *Grammar Systems: A Grammatical Approach to Distribution and Cooperation.* Gordon and Breach Science Publ., Yverdon, 1994.

[66] E. Csuhaj-Varjú, C. Martín-Vide, V. Mitrana, *Multiset Automata.* In [42], pp. 69-83.

[67] V. Danos, V. Schachter eds., *Computational Methods in Systems Biology, International Conference CMSB 2004.* Paris, France, 2004, LNCS 3082, Springer-Verlag, Berlin, 2005.

[68] J. Dassow, Gh. Păun, *Regulated Rewriting in Formal Language Theory.* Springer-Verlag, Berlin, 1989.

[69] A. Ehrenfeucht, T. Harju, I. Petre, D.M. Prescott, G. Rozenberg, *Computation in Living Cells.* Springer-Verlag, Berlin, 2004.

[70] P. Fay, C. Van Baalen, eds., *The Cyanobacteria.* Elsevier Science Publishers, Amsterdam, 1987.

[71] H. Fernau, F. Stephan, How Powerful is Unconditional Transfer? When UT Meets AC. In [37], pp. 249–260.

[72] C. Ferretti, G. Mauri, C. Zandron eds., *10th International Workshop on DNA Computing, DNA10.* Milan, Italy, June 2004, LNCS 3384, Springer-Verlag, Berlin, 2005.

[73] R.P. Feynman, There's plenty of room at the bottom. In [81], pp. 282–286.

[74] R. Freund, Asynchronous P Systems and P Systems Working in the Sequential Mode. In [105], pp. 36–62.

[75] R. Freund, L. Kari, M. Oswald, P. Sosik, Computationally Universal P Systems Without Priorities: Two Catalysts are Sufficient. *Theoretical Computer Science*, 330, 2 (2005), pp. 251–266.

[76] R. Freund, L. Kari, Gh. Păun, DNA Computing Based on Splicing: The Existence of Universal Computers. *The Theory of Computing Systems*, 32, 1 (1999), pp. 69–112.

[77] R. Freund, G. Lojka, M. Oswald, Gh. Păun eds., *Pre-Proceedings of the 6th International Workshop on Membrane Computing, WMC6.* University of Vienna, July 2005.

[78] R. Freund, M. Oswald, P Systems with Antiport Rules for Evolution Rules. In [120], pp. 183–192.

[79] P. Frisco, *Theory of Molecular Computing. Splicing and Membrane Systems.* Ph.D. Thesis, University of Leiden, 2004-03.

[80] K. Fujinbayashi, S. Murata, A Method of Error Suppression for Self-Assembly DNA Tiles. In [72], pp. 113-127.

[81] H.D. Gilbert ed., *Miniaturization.* Reinhold, New York, 1961.

[82] M.A. Gutiérrez-Naranjo, Gh. Păun, M.J. Pérez-Jiménez, eds., *Proceedings of the ESF Exploratory Workshop on Cellular Computing (Complexity Aspects).* Fénix Ed., Seville, Spain, 2005.

[83] M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, Available Membrane Computing Software. In [63], pp. 410–435.

[84] M.A. Gutiérrez-Naranjo, A. Riscos-Núñez, F.J. Romero-Campero, D. Sburlan, eds., *Proceedings of the Third Brainstorming Week on Membrane Computing.* Seville, 2005, GCN Technical Report 01/2005, University of Seville, 2005.

[85] M. Hagiya, A. Ohuchi, eds., *DNA computing: 8th International Workshop on DNA Based Computers, DNA8.* Sapporo, Japan, June 2002, LNCS 2538, Springer-Verlag, Berlin, 2003.

[86] D.O. Hall, K.K. Rao, *Photosynthesis.* Cambridge University Press, 1994.

[87] T. Head, Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors. *Bulletin of Mathematical Biology,* 49 (1987), pp. 737–759.

[88] J.E. Hopcroft, J.D. Ullman, *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading MA, 1979.

[89] O.H. Ibarra, A. Păun, Counting Time in Computing with Cells. In [43], pp. 25–36.

[90] N.D. Jones, A Survey of Formal Language Theory. Technical Report 3, University of Western-Ontario, Computer Science Department, 1966.

[91] N. Jonoska, S.A. Karl, M. Saito, Three Dimensional DNA Structures in Computing. In [94], pp. 143–153.

[92] N. Jonoska, Gh. Păun, G. Rozenberg, eds., *Aspects of Molecular Computing, Essays Dedicated to Tom Head on the Occasion of His 70th Birthday.* LNCS 2950, Springer-Verlag, Berlin, 2004.

[93] N. Jonoska, N.C. Seeman, eds., *DNA Computing: 7th International Workshop on DNA-Based Computers, DNA7.* Tampa, FL, USA, June 2001, LNCS 2340, Springer-Verlag, Berlin, 2001.

[94] L. Kari, H. Rubin, D.H. Woods, eds., *Proceedings of the 4th DIMACS Meeting on DNA Based Computers.* 52, 1-3. Elsevier, 1999.

[95] S.N. Krishna, A. Păun, Three Universality Results on P Systems. In [57], pp. 198–206.

[96] S.N. Krishna, A. Păun, Some Universality Results on Evolution-Communication P Systems. In [57], pp. 207–215, and *New Generation Computing*, 22, 4 (2004), pp. 377–394.

[97] J.M. Levsky, S.M. Shenoy, R.C. Pezo, R.H. Singer, Single-Cell Gene Expression Profiling. *Science* 297 (2002), pp. 836–840.

[98] J. Lippincott-Schwartz, E. Snapp, A. Kenworthy, Studying Protein Dynamics in Living Cells. *Nature Rev. Mol. Cell. Biol.*, 2 (2001), pp. 444–456.

[99] R.J. Lipton, Using DNA to Solve NP-Complete Problems. *Science*, 268 (1995), pp. 542–545.

[100] V. Manca, C. Zandron, A DNA Algorithm for 3-SAT(11,20). In [93], pp. 172–181.

[101] M. Margenstern, ed., *Proceedings of MCU 2004, Machines, Computations and Universality*. St. Petersburg, Russia, September 2004, LNCS 3354, Springer-Verlag, Berlin, 2005.

[102] C. Martín-Vide, G. Mauri, Gh. Păun, G. Rozenberg, A. Salomaa, eds., *Workshop on Membrane Computing, International Workshop, WMC2003*. Tarragona, Spain, July 2003, LNCS 2933, Springer-Verlag, Berlin, 2004.

[103] C. Martín-Vide, Gh. Păun, *Workshop on Membrane Computing WMC2001*. Curtea de Argeş, Romania, August 2001, *Fundamenta Informaticae*, 49, 1-3 (2002).

[104] C. Martín-Vide, Gh. Păun, G. Rozenberg, Membrane Systems with Coupled Transport. Universality and Normal Form. *Fundamenta Informaticae*, 49, 1-3(2002), pp. 1–15.

[105] G. Mauri, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa, eds., *Membrane Computing, 5th International Workshop, WMC2004*. Milano, Italy, June 2004, LNCS 3365, Springer-Verlag, Berlin, 2005.

[106] A. Mateescu, Gh. Păun, G. Rozenberg, A. Salomaa, Simple Splicing Systems. *Discrete Applied Mathematics*, 84 (1998), pp. 145–163.

[107] X. Michalet, F.F. Pinaud, L.A. Bentolila, J.M. Tsay, S. Doose, J.J. Li, G. Sundaresan, A.M. Wu, S.S. Gambhir, S. Weiss, Quantum Dots for Live Cells, in Vivo Imaging and Diagnostic. *Science*, 307-5709 (2005), pp. 538–544.

[108] S.L. Miller, L.E. Orgel, *The Origin of Life on Earth.* Prentice Hall, Englewood Cliffs, 1973.

[109] M.L. Minsky, *Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, 1967.

[110] J. Mira, J.R. Álvarez, *Mechanisms, Symbols and Models Underlying Cognition: First International Work-Conference on the Interplay Between Natural and Artificial Computation, IWINAC 2005.* Las Palmas, Canary Islands, Spain, June 2005, LNCS 3561, Springer-Verlag, Berlin, 2005.

[111] H. Nakagawa, K. Sakamoto, Y. Sakakibara, Development of an In Vivo Computer Based on Escherichia Coli. In [43], pp. 68–77.

[112] T.Y. Nishida, Simulation of Photosynthesis by a K-Subset Transforming System with Membranes. *Fundamenta Informaticae*, 49, 1-3 (2002), pp. 249–259.

[113] A. Păun, Gh. Păun, The Power of Communication: P Systems with Symport/Antiport. *New Generation Computing*, 20 (2002), pp. 295–305.

[114] Gh. Păun, Five (Plus Two) Universal DNA Computing Models Based on the Splicing Operation. In [22], pp. 67–86.

[115] Gh. Păun, Computing with Membranes. *Journal of Computer and System Sciences*, 61, 1(2000), pp. 108–143. First circulated as TUCS Research Report 28, 1998.

[116] Gh. Păun, P Systems with Active Membranes: Attacking NP Complete Problems. *Journal Automata Languages and Combinatorics*, 6-1 (2001), pp. 75–90, and CDMTCS Research Report 102, Auckland University, 1999.

[117] Gh. Păun, *Membrane Computing. An Introduction.* Springer-Verlag, Berlin, Heidelberg, 2002.

[118] Gh. Păun, Membrane Computing: Some Non-Standard Ideas. In [92], pp. 322–377.

[119] Gh. Păun, Introduction to Membrane Computing. In *Proceedings First Brainstorming Workshop on Uncertainty in Membrane Computing*, Palma de Mallorca, Spain, November 2004, pp. 1–42.

[120] Gh. Păun, A. Riscos-Núñez, A. Romero-Jiménez, F. Sancho-Caparrini, eds., *Proceedings Second Brainstorming Week on Membrane Computing.* Seville, 2004, GCN Technical Report 01/2004, University of Seville, 2004.

[121] Gh. Păun, G. Rozenberg, A Guide to Membrane Computing. *Theoretical Computer Science*, 287-1 (2002), pp. 73–100.

[122] Gh. Păun, G. Rozenberg, A. Salomaa, *DNA Computing - New Computing Paradigms*. Springer-Verlag, Berlin, 1998.

[123] Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron, eds., *Membrane Computing 2002*. Curtea de Argeş, Romania, August 2002, LNCS 2597, Springer-Verlag, Berlin, 2003.

[124] J. Pelmont, *Catalyseurs du monde vivant*. Press Universitaires de Grenoble, Grenoble, 1995.

[125] M.J. Pérez-Jiménez, F.J. Romero-Campero, Modeling EGFR Signaling Cascade Using Continuous Membrane Systems. In [131].

[126] M.J. Pérez-Jiménez, F.J. Romero-Campero, A Study of the Robustness of the EGFR Signaling Cascade Using Continuous Membrane Systems. In [110], pp. 268–278.

[127] G.A. Peschek, Respiratory Electron Transport. In [70], pp. 119–161.

[128] G.A. Peschek, C. Obinger, S. Fromwald, B. Bergman, Correlation Between Immuno-Gold Based and Activities of the Cytochrome-c Oxidase ($aa_3 - type$) in Membranes of Salt Stressed Cyanobacteria. *FEMS – Microbiology Letters*, 124 (1994), pp. 431–438.

[129] G.A. Peschek, R. Zoder, Temperature Stress and Basic Bioenergetic Strategies for Stress Defence. In [134], pp. 203–258.

[130] D. Pescini, D. Besozzi, C. Zandron, G. Mauri, Analysis and Simulation of Dynamics in Probabilistic P Systems. In [43], pp. 310–321.

[131] G. Plotkin, ed., *Pre-Proceedings of the Third International Workshop on Computational Methods in Systems Biology, CMSB 2005*. Edinburgh, UK, 2005.

[132] I. Prigogine, R. Lefever, Symmetry Breaking Instabilities in Dissipative Systems II. *Journal of Chemical Physics*, 48(1968), pp. 1695–1700.

[133] A. Puustinen, M. Finel, T. Haltia, R.B. Gennis, M. Wikstrom, Properties of the Two Terminal Oxidades of Escherichia Coli. *Biochemistry*, 30 (1991), pp. 3936–3942.

[134] L.C. Rai, J.P. Gaur, eds., *Algal Adaptation to Environmental Stress*. Springer-Verlag, Berlin, 2001.

[135] P.W.K. Rothemund, A DNA and Restriction Enzyme Implementation of Turing Machines. In [23], pp. 75–120.

[136] S. Roweis, E. Winfree, R. Burgoyne, N. Chelyapov, M. Goodman, P.W.K. Rothemund, L.M. Adleman, A Sticker Based Architecture for DNA Computation. In [22], pp. 1–27.

[137] G. Rozenberg, A. Salomaa, *The Mathematical Theory of L Systems.* Academic Press, New York, 1980.

[138] G. Rozenberg, A. Salomaa, eds., *Handbook of Formal Languages.* Springer-Verlag, Berlin, 1997.

[139] H. Rubin, D. Wood, eds., *DNA Based Computers, Proceedings of the Third Annual Meeting.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 48, AMS, 1999.

[140] K. Sakamoto, H. Gounzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, M. Hagiya, Molecular Computation by DNA Hairpin Formation. *Science*, 228 (2000), pp. 1223–1226.

[141] A. Salomaa, *Formal Languages.* Academic Press, New York, 1973.

[142] M. Soreni, S. Yogev, E. Kossoy, Y. Shoham, Eh. Keinan, Parallel Biomolecular Computation on Surfaces with Advanced Finite Automata. *Journal American Chemical Society*, 127, 11 (2005), pp. 3935–3943.

[143] R. Weiss, S. Basu, S. Hooshangi, A. Kalmbach, D. Karig, R. Mehreja, I. Netravali, Genetic Circuit Building Blocks for Cellular Computation, Communications, and Signal Processing. *Natural Computing*, 2-1 (2003), pp. 47–84.

[144] E. Winfree, On the Computational Power of DNA Annealing and Ligation. In [23], pp. 199–210.

[145] E. Winfree, T. Eng, G. Rozenberg, String Tile Models for DNA Computing by Self-Assembly. In [64], pp. 63–68.

[146] E. Winfree, D.K. Gifford, eds., *DNA Based Computers, Proceedings of the Fifth Annual Meeting.* DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 54, AMS, 2000.

[147] E. Winfree, F. Lin, L.A. Wenzler, N.C. Seeman, Design and Self-Assembly of Two-Dimensional DNA Crystals. *Nature*, 394-6693 (1998), pp. 539–545.

[148] E. Winfree, X. Yang, N.C. Seeman, Universal Computation Via Self-Assembly of DNA: Some Theory and Experiments. In [22], pp. 191–214.

[149] An On-line Biology Book, http://www.emc.maricopa.edu/faculty/farabee/BIOBK/BioBookTOC.html.

[150] ISI web-page, http://esi-topics.com/erf/october2003.html.

[151] P systems web-page, http://psystems.disco.unimib.it/.

Matteo Cavaliere

"Evolution, Communication, Observation: From Biology to Membrane Computing and Back"

Dottorato di Ricerca

cum laude

1 Febbraio del 2006

IL PRESIDENTE,