Quantum Computing

Neural Networks

Evolutionary Computing

DNA Computing

Gheorghe Păun

Membrane Computing

An Introduction

NATURAL COMPUTING SERIES

Springer

Natural Computing Series

Series Editors: G. Rozenberg (Managing)
Th. Bäck A.E. Eiben J.N. Kok H.P. Spaink
Leiden Center for Natural Computing

Advisory Board: S. Amari G. Brassard M. Conrad
K.A. De Jong C.C.A.M. Gielen T. Head L. Kari
L. Landweber T. Martinetz Z. Michalewicz M.C. Mozer
E. Oja G. Păun J. Reif H. Rubin A. Salomaa M. Schoenauer
H.-P. Schwefel D. Whitley E. Winfree J.M. Zurada

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Gheorghe Păun

Membrane Computing

An Introduction

With 37 Figures and 5 Tables

Springer

Gheorghe Păun

Institute of Mathematics
of the Romanian Academy
P.O.Box 1-764
70700 Bucuresti, Romania
and
Rovira i Virgili University
Pl. Imperial Tarraco, 1
43005 Tarragona, Spain
gpaun@imar.ro, gp@astor.urv.es
www.imar.ro/~gpaun

Series Editors

G. Rozenberg (Managing Editor)
Th. Bäck, A.E. Eiben, J.N. Kok, H.P. Spaink
Leiden Center for Natural Computing
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
rozenber@cs.leidenuniv.nl

Library of Congress Cataloging-in-Publication Data

Păun, Gheorghe, 1950-
Membrane computing: an introduction/Gheorghe Păun.
p. cm. – (Natural computing series)
Includes bibliographical references and index.
ISBN 3540436014 (hc: alk. paper)
1. Molecular computers. I. Title. II. Series.

QA 76.887 .P39 2002
621.39'1–dc21 2002075730

ACM Computing Classification (1998): F.1.1, F.4.2, F.4.3
ISSN 1619-7127
ISBN 3-540-43601-4 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York,
a member of BertelsmannSpringer Science+Business Media GmbH
http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002

The use of general descriptive names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Cover Design: KünkelLopka, Heidelberg
Typesetting: Camera ready by the author
Printed on acid-free paper SPIN 10877938 45/3142PS – 5 4 3 2 1 0

To my wife Anişoara
(Ana de la Argeş)

Preface

Membrane computing is a branch of natural computing which abstracts from the structure and the functioning of living cells. In the basic model, the membrane systems – also called P systems – are distributed parallel computing devices, processing multisets of objects, synchronously, in the compartments delimited by a membrane structure. The objects, which correspond to chemicals evolving in the compartments of a cell, can also pass through membranes. The membranes form a hierarchical structure – they can be dissolved, divided, created, and their permeability can be modified. A sequence of transitions between configurations of a system forms a computation. The result of a halting computation is the number of objects present at the end of the computation in a specified membrane, called the output membrane. The objects can also have a structure of their own that can be described by strings over a given alphabet of basic molecules – then the result of a computation is a set of strings. An important version of membrane systems deals with membranes arranged not in a hierarchical structure (which mathematically corresponds to a tree), but in a tissue-like structure (which mathematically corresponds to a graph).

This computing model (more accurately, this class of computing models) was proposed at the end of 1998, and, in less than four years since then (until the time of completing the manuscript of this book), more than 180 papers have been written, as well as three PhD theses (with several more in preparation now) on the subject. Hence the task of presenting membrane computing in a monograph is by no means a simple one. In particular, new notions are currently introduced, new results are proved and old results are improved; the focus of research may still change depending on various criteria (ranging from mathematical appeal to practical attractiveness), and therefore, the task of choosing central (lasting) notions and results for this monograph was really difficult.

This is why this book is only an introduction to membrane computing, with a subjective selection of basic notions and (classes of) results. The book is only a snapshot of developments in a vivid research area, and it is meant to be an invitation to witness and to participate in this research. Hopefully, further "generations" of this book will be made both possible and necessary by the reader's contributions to membrane computing – such contributions

are very much encouraged by the style of the book: it is as self-contained as possible from biological and computer science points of view, with many open problems and research topics explicitly formulated. The problems are numbered, from Q1 to Q39; a table at the end of the book gives the pages where these problems are stated.

Another list at the end of the book contains the universality results proved or only stated in the text; the goal of this list is to provide a clear overview of universality results, and in particular of the role of various parameters (such as, for example, the number of membranes) in obtaining universality.

The reader interested in further details, or in the current developments in membrane computing, is invited to visit the web page at the address `http://bioinformatics.bio.disco.unimib.it/psystems`.

*

This book has grown continuously during the last four years, in collaboration with the many friends from many countries with whom I have had the privilege to work. I will only name them, hopefully without too many omissions: I. Ardelean, F. Arroyo, A. Atanasiu, A. Baranda, D. Besozzi, P. Bottoni, J. Castellanos, G. Ciobanu, E. Csuhaj-Varjú, J. Dassow, C. Ferretti, R. Freund, P. Frisco, T. Head, H.J. Hoogeboom, M. Ito, J. Kelemen, V. Manca, S. Marcus, M. Margenstern, C. Martín-Vide, A. Mateescu, G. Mauri, V. Mitrana, J. Pazos, A. Păun, M. Pérez-Jiménez, A. Rodríguez-Patón, Y. Rogozhin, G. Rozenberg, Y. Sakakibara, A. Salomaa, F. Sancho-Caparrini, Y. Suzuki, H. Tanaka, T. Yokomori, S. Yu, C. Zandron. Several of them have read (parts of) the manuscript of the present book, making valuable observations.

To all of them, the warmest thanks and indebtedness.

Of great help also have been the participants to the annual workshops on membrane computing held in Curtea de Argeş, Romania, since 2000.

Special thanks are due to Claudio Zandron, the webmaster taking care of the Internet page mentioned above.

Thanks are offered in advance to readers who take up the present invitation to membrane computing, to send their comments about the text or, mainly, contributions to the contents of a possible future version of the book.

The writing of this book has benefited from the wonderful working conditions provided by the Research Group on Mathematical Linguistics, of Rovira i Virgili University, Tarragona, Spain, where I work as a researcher on the Ramon y Cajal programme of the Spanish Ministry of Research.

Last but not least, I should mention the pleasant and efficient collaboration with Springer-Verlag. Mrs Ingeborg Mayer deserves a special bow in this respect.

Tarragona, June 2002 *Gheorghe Păun*

Contents

Preface VII

1. Introduction: Membrane Computing – What It Is and What It Is Not 1

2. Prerequisites 7
2.1 The Biological Membrane 7
2.1.1 The Structure of the Plasma Membrane 8
2.1.2 Trans-membrane Transport 10
2.1.3 Cellular Division: Mitosis 14
2.2 The Neuron 15
2.3 Elements of Computability 16
2.3.1 Basic Notions and Notations 17
2.3.2 Operations with Strings and Languages 18
2.3.3 Chomsky Grammars 19
2.3.4 Characterizations, Necessary Conditions 22
2.3.5 Lindenmayer Systems 24
2.3.6 Finite Automata, Turing Machines 26
2.3.7 Regulated Rewriting 29
2.3.8 On the Difference Between CS and RE 39
2.3.9 Universal Turing Machines and Type-0 Grammars 40
2.3.10 Splicing, Insertion–Deletion, Context Adjoining 42
2.3.11 Elements of Complexity 45
2.3.12 Multisets 49
2.4 Bibliographical Notes 50

3. Membrane Systems with Symbol–Objects 51
3.1 A Simple Class 51
3.2 Two Examples 55
3.3 The Power of the Simple Class 58
3.4 Basic Extensions 64
3.4.1 Dissolving a Membrane 64
3.4.2 Priorities Among the Evolution Rules 70
3.4.3 Two Further Examples 71

3.4.4 The Power of Priority ... 74
3.4.5 The Power of Synchronization ... 77
3.5 A Formal Definition ... 85
3.6 Further Extensions ... 91
3.6.1 Weak Target Commands ... 91
3.6.2 Controlling the Permeability of Membranes ... 93
3.6.3 Communication Controlled by Concentration ... 99
3.6.4 Creating Rules During the Computation ... 102
3.6.5 Using Promoters/Inhibitors ... 104
3.7 Systems with External Output ... 114
3.8 Bibliographical Notes ... 125

4. Trading Evolution for Communication ... 129
4.1 Systems with Symport/Antiport ... 130
4.2 Computational Universality ... 133
4.3 Controls on the Use of Rules ... 141
4.4 Following the Traces of Objects ... 144
4.5 Systems with Carriers ... 153
4.6 Bibliographical Notes ... 159

5. Structuring the Objects ... 161
5.1 Rewriting Membrane Systems ... 162
5.2 Some Variants and Their Power ... 180
5.2.1 Rule Creation ... 181
5.2.2 Conditional Rewriting ... 181
5.2.3 Conditional Communication ... 186
5.2.4 Replicated Rewriting ... 199
5.2.5 Parallel Rewriting ... 208
5.3 Splicing Membrane Systems ... 211
5.4 Contextual Membrane Systems ... 223
5.5 Insertion–Deletion Membrane Systems ... 226
5.6 Bibliographical Notes ... 231

6. Networks of Membranes ... 235
6.1 The Splicing Case ... 236
6.2 Using Symport/Antiport Rules ... 238
6.3 Neural-like Networks of Membranes ... 249
6.3.1 Definitions and Examples ... 249
6.3.2 The Computational Power ... 256
6.3.3 The Computational Efficiency ... 267
6.4 Bibliographical Notes ... 269

7. Trading Space for Time ... 271
7.1 Complexity Classes for Membrane Systems ... 271
7.2 Using Membrane Division ... 273
7.2.1 Solving SAT in Linear Time ... 281
7.2.2 Solving the Hamiltonian Path Problem ... 286
7.2.3 Using Cooperative Rules ... 290
7.2.4 Is Membrane Division Necessary? ... 298
7.3 Using Membrane Creation ... 301
7.3.1 Solving SAT ... 311
7.3.2 Solving HPP ... 316
7.3.3 The Case of String–Objects ... 318
7.4 Using String Replication ... 321
7.5 Using Pre-computed Resources ... 323
7.6 Bibliographical Notes ... 327

8. Further Technical Results ... 329
8.1 Decidability Results ... 329
8.2 Unary Systems ... 340
8.3 A Representation of Context–free Languages ... 344
8.4 Valuating the String–Objects ... 348
8.5 Systems with Enhanced Membrane Handling ... 351
8.6 Brief Excursion Through the Literature ... 354
8.6.1 Generalized Sequential Membrane Systems ... 354
8.6.2 Bidimensional Objects ... 357
8.6.3 Membrane Systems and Stream X-machines ... 357
8.6.4 Membrane Systems and Ambient Calculus ... 359
8.6.5 A Direct Construction of a Universal System ... 361
8.6.6 Further Research Topics ... 363

9. (Attempts to Get) Back to Reality ... 367
9.1 Getting Closer to the Cell by Energy Accounting ... 367
9.2 Getting Closer to the Cell by Gemmation ... 373
9.3 Getting Closer to the Cell: Bilayer Membranes ... 375
9.4 *In Silico* Implementations ... 379
9.5 Artificial Life Applications ... 384
9.6 A Simulation of Photosynthesis ... 392

Open Problems ... 399

Universality Results ... 401

References ... 403

Index ... 417

1. Introduction: Membrane Computing – What It Is and What It Is Not

The history of formalizing the notion of algorithm (and of computation) dates back to G.W. Leibniz (1646–1716) when it focused on trying to model the computations performed by humans, for example bank clerks. These efforts culminated in the first part of the 20th century with the formalization in the form of machines, in the work of Turing [238]. These formalizations were very successful, because they led to the construction of the first electronic computers in the 1940′s. These efforts already were using some ideas from biology, for example, the functioning of neurons in neural networks (Kleene [106], McCulloch and Pitts [165]).

Then, in the second part of the 20th century, a great deal of effort was devoted to understanding computations going on in nature, and to benefitting from this understanding by getting better (more efficient) algorithms or even new types of computers. Well-known examples of such efforts are genetic algorithms (or, more generally speaking, evolutionary computing), and neural networks. A more recent example is the area of DNA computing or the area of molecular computing in general, which is based on paradigms from molecular biology.

All these areas – evolutionary computing, neural computing, and molecular computing – belong to the emerging area of *natural computing*, which is concerned with computing which go on in nature or is inspired by nature.

Membrane computing is (proposed here as) one further branch of natural computing. It enriches the models of molecular computing by providing a spatial structure for molecular computations, inspired by the membrane structure of living cells. In this way, membrane computing comes close to the "computing cell" paradigm.

Membrane computing starts from the observation that over billions of years nature has evolved an astonishing piece of machinery, which is both microscopic and very complex, namely the *cell*, the smallest living thing we know. In the structure and the functioning of a cell, *membranes* play an essential role. The cell is separated from its environment by means of the plasma membrane and it is internally compartmentalized by means of internal membranes. The membrane computing model formalizes this fundamental feature of the living cells, namely, the *membrane structure*.

The role of membranes in biochemistry, in general, for defining the phenomenon usually called *life*, cannot be overestimated. We do not enter here into details, but we only recall the assertion of Kauffman [103] that "the secret of life, the wellspring of reproduction, is not to be found in the beauty of Watson–Crick pairing, but in the achievement of collective catalytic closure". Along the same lines, we read in [90]: "If any entity should be thought as a governor of cellular activity, this should certainly be the membrane.[...] All major activities of cells are topologically connected to membranes.[...] Membranes also are the primary organizers of multi-cellular life."

Although we refer to [144] for further discussions of similar ideas, here we just want to stress the fact that membranes play a crucial role in living cells. Not only do membranes have a semiotic–philosophical role of defining the inside–outside frontier and separating a body from its environment, but also – and this is the main interest of this book – they have a biochemical role. The cell membrane separates, and hence protects, the internal space of the cell from the (often harmful) environment. The inner membranes delimit the structure of the cell – in particular, they protect the nucleus, which contains the genetic information. Membranes are essentially involved in many reactions which take place in the compartments of a cell, and they are selective channels of communication (for the transfer of chemicals) between compartments as well as between the cell and its environment. We will return to these topics when we give some biological details about the structure and functioning of the cell membrane.

Bearing in mind that some major developments in computer science have been inspired by nature and proved to be both theoretically appealing and practically interesting, natural questions (for a mathematician) to be asked are: What about the cell as a source of inspiration for computer science? Can we abstract a computing model from the living cell structure and functioning? Is the cell a "computer"? Can we implement computations through living cells, aiming to obtain *wet computers*, or rather should the models inspired by the cell structure be implemented on existing electronic hardware? Figure 1.1 illustrates this last dilemma and gives an informal but hopefully informative picture of the areas of natural computing (inspired by biology) mentioned above.

Although inspired by the biological structure of living cells, the membrane computing model discussed in this book is meant to provide computational models along the lines of computability theory as studied in computer science.

This model can be briefly described as follows. Consider a *membrane structure* consisting of several membranes arranged in a hierarchical structure inside a main membrane (called the *skin*) and delimiting *regions* (each region is bounded by a membrane and the immediately lower membranes, if there are any; a membrane which does not contain any membrane inside is said to be *elementary* and so its region is the space delimited by it). The membranes are understood as three-dimensional vesicles, although an intuitive representation

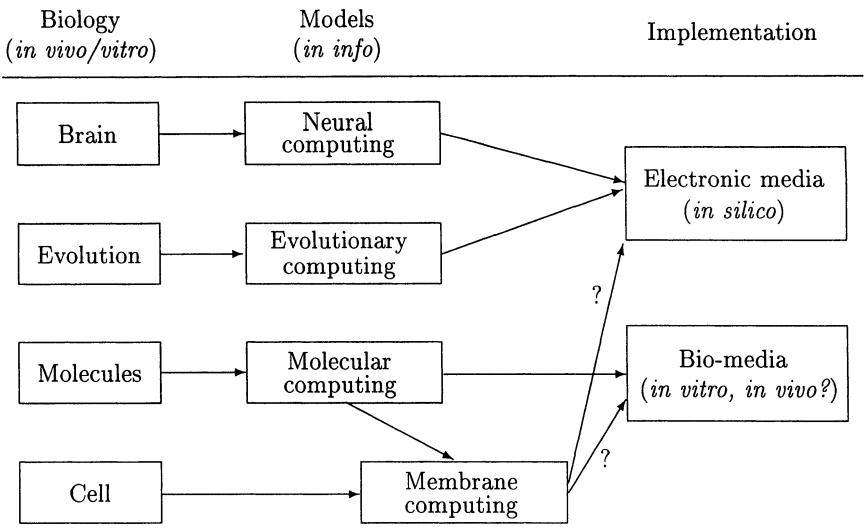


Fig. 1.1. The four bio-domains of natural computing

of a membrane structure is often given in the form of a planar Venn diagram, as that in Figure 1.2, which illustrates the key notions introduced above.

Regions contain *multisets* of *objects*, that is, sets of objects with multiplicities associated with the elements (the objects correspond to chemicals present in the compartments of a cell in certain quantities given by nonnegative integers). The objects are represented by symbols from a given alphabet, hence by their names; they are *atoms* in the etymological sense: undivisible and with no structure. They evolve according to given *evolution rules*, which are also localized, that is the rules are associated with the regions (hence with the membranes, because, obviously, the correspondence membranes–regions is one-to-one). The rules are applied non-deterministically (the rules to be used and the objects to evolve are chosen randomly), in a maximally parallel manner (in each step, all objects which *can* evolve *must* do so). The objects can also be moved (*communicated*) between regions. The membranes can be dissolved, divided, or created. In this way, we get *transitions* from one *configuration* of the system to the next one. This process is synchronized – a global clock is assumed, marking the time units common to all compartments of the system. A sequence of transitions constitutes a *computation*; a computation which reaches a configuration where no rule is applicable to the existing objects is a *halting computation*. With each halting computation we associate a *result* – the number of objects in a specified *output membrane*.

Hence a computation in a membrane system is structured as follows: it starts with an initial configuration of the system (the initial membrane structure and the initial distribution of objects within regions), then the

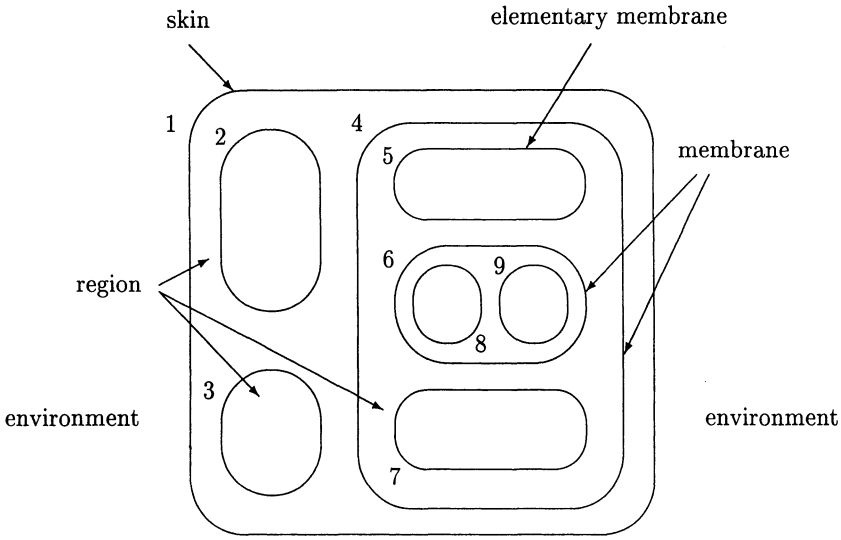


Fig. 1.2. A membrane structure

computation proceeds, and when it stops the result is to be found in a specific output membrane.

Many versions of this basic model were investigated. For instance, for the basic model considered above, where the objects are described by symbols, we can consider priority relations among rules, we can vary the membrane thickness, and hence permeability, we can use the rules in a conditional manner, depending on certain promoters/inhibitors present/absent in the regions, we can consume and create rules (the rules correspond to chemical reactions taking place in regions of the cell, and many such reactions are mediated by enzymes), and so on and so forth.

Moreover, many chemicals from a cell have an inner structure and may be described by strings of symbols rather than by single symbols (for example, DNA molecules are composed from 4 nucleotides, and proteins are composed from 20 aminoacids). Then we have to use string processing evolution rules, which can be based on the classic operation of rewriting, or based on genome specific operations such as splicing, insertion–deletion, crossover, and replication. When we deal with string–objects, the result of a computation is the set of strings present in the halting configuration in the ouput membrane, hence the result of a computation is a language. Another fruitful idea is to consider multi-cellular arrangements of cells rather than a single cell, and to investigate tissue-like networks of (elementary) membranes. A wealth of possibilities indeed exists.

The passing from symbol–objects to string–objects suggests a nice analogy. Recall that Lindenmayer systems (introduced in [125], for an overview see, for instance, [89, 218]) are grammatical models of the development of

multi-cellular organisms which consider cells as basic atomic objects. Therefore, in Lindenmayer systems, the cells are identified by symbols, without any structure. In membrane systems with symbol–objects we operate a first level of zooming which shows the membrane structure of a cell as well as chemicals (molecules) present in the regions; however, the chemicals are "atomic entities" with no internal structure and therefore are represented by single symbols. If we zoom in once again, then we can distinguish a structure also for these objects, obtaining in this way structured objects. Since the structure of many molecules may be represented by strings, we consider membrane systems with string–objects.

Clearly, the operations applied on a specific zooming level correspond to the given zooming. When we see only symbol–objects our operations are operations on symbols representing these objects; when we see also the string structure of objects then the operations on this level of zooming are string processing operations.

From the computability point of view most of the classes of membrane systems obtained have two rather attractive properties: *computational completeness* (they are equivalent in power to Turing machines), and *computational efficiency* (by making use of an exponential workspace created in a natural manner, various types of membrane systems are able to solve **NP**-complete problems in polynomial – often linear – time).

Two remarks are in order here. The membrane systems are finite, discrete computing models, hence their work can be simulated by a Turing machine; in this way we cannot break "the Turing–Church barrier". Then, the "solutions" to **NP**-complete problems are obtained in an ideal framework, where, for instance, we can repeatedly divide membranes, and hence replicate cells, in order to obtain 2^n "processors" in n steps. Dividing cells is a common biological phenomenon, but up to now no computation has been implemented on a bio-ware in the form of a membrane system. Currently, we have no idea where to look for an implementation of membrane computing of practical interest, in biology or on existing electronic computers. We have returned to the dilemma pointed out in Figure 1.1, but it is too early to ask such questions and, especially, to answer them; serious interdisciplinary investigations must precede any sensible answer.

As is also the case with the investigations carried out so far, these new investigations can/must be motivated from various directions, as the models we look for should be mathematically as elegant as possible, computationally as powerful as possible, computationally as efficient as possible (the last two criteria can be contradictory in themselves, as pointed out by M. Conrad [43] – actually, M. Conrad states the incompatibility of three desirable features of computing models/devices: universality/programmability, efficiency, and adaptability/evolvability), and, finally, biologically as realistic as possible. These goals are seemingly contradictory and may motivate many different versions, attempts, and combinations of ingredients.

To conclude, membrane computing is a branch of (theoretical) computer science, an area of mathematical investigations, but not yet an area of practical computing applications (certain applications, of a scientific nature, can already be counted, because concepts and results from membrane computing were borrowed or translated into other areas of research – the last chapter of the book provides some illustrations of this assertion).

It is important to mention something else that membrane computing *is not* (yet): it is not a framework for modelling the "real" cell in such a way as to obtain conclusions relevant for biology. A beautiful book on neural networks, [5], starts with the following: "It is not absolutely necessary to believe that neural network models have anything to do with the nervous system, but it helps. Because, if they do, we are able to use a large body of ideas, experiments, and facts." This assertion can be adapted to membrane computing models, with a suitable grain of humour. Membrane computing abstracts from cell structure and functioning, with the aim of finding computability-relevant ideas, but in this abstraction process we leave biology far behind. The models themselves and the results are relevant for computer science, and we have no intention here of making any claim of direct interest to the (traditional) biologist. This does not mean, of course, that such a goal, of returning some meaningful information to biology, is not a possible important future target of investigations in this area. This is somewhat necessary, taking into account the relative disappointment of biology with respect to the models based on analytic, continuous mathematics, generally used so far. Important progress in computability, which makes it possible to handle huge volumes of data, as already available in biology, suggests that a new age of mathematization of biology might be possible by means of discrete, computer-science-inspired models. Membrane systems are discrete computer science models, but it is also necessary that they return to the "real cell" before hoping to become relevant for biology.

In the meantime, many things should be done, starting with clarifying the many mathematical problems which decorate the landscape of membrane computing, and ending with the practical problem of implementations/applications. The framework is rather general and versatile – compartmental computation, with the separate "processors" communicating in various ways, synchronized, non-deterministic, parallel, dealing with multisets (a data structure often appearing in computer science, but never systematically investigated) – and this makes a non-limited range of branches, analogies, cooperation with other areas, and applications possible.

After all, this is what membrane computing *is*: a new territory for computer theorists to explore.

2. Prerequisites

The main paradigms of membrane computing are abstracted from the structure and the functioning of living cells. That is why we start by briefly recalling some information about the plasma membrane, the cell compartmentalization, the trans-membrane transfer of chemicals, and the inter-cellular communication. To this end we use standard biological books, mainly [4] (but also other sources, such as [9, 127, 129, 237]). The presentation will be schematic, bearing in mind the models we will define in the subsequent chapters of the book, and having as our audience the computer scientist.

Then, we will briefly describe the neural cell and the way the neurons cooperate.

The last sections of this chapter are devoted to computability prerequisites, especially to elements of automata and language theory necessary in the book, as well as to some general elements of complexity theory.

Of course, the reader familiar with these topics does not need to go through all these prerequisite sections – or, at most, (s)he only needs to check some mathematical notations (for instance, the conventions on the way the matrix grammars are presented, because these conventions will be implicitly or explicitly used in the whole book).

2.1 The Biological Membrane

The cell is the fundamental unit of all living organisms, and, at the same time, the living cell is a very complex and well-organized structure, where a large number of elaborate reactions take place all the time.

There are two types of cells, *eukaryotic* and *prokaryotic*. The eukaryotic cells have a nucleus and a much more complex structure than the prokaryotic cells, and mainly the eukaryotic cells are considered in what follows.

A eukaryotic cell contains many different parts, in general, delimited by *membranes*.

Among the most important components of a cell, we mention: the *plasma membrane* (the external membrane, delimiting the cell from its environment), the *nucleus* (where the genetic information is stored, encoded in DNA and RNA molecules; the nucleus is surrounded by two concentric membranes, and has in the centre the *nucleolus*, where RNA is placed), the *mitochondria* (also

surrounded by a membrane; the mitochondria produce energy for the cell), the *Golgi apparatus* (an intricate protein processing "factory", with a central role in the cell metabolism), the *ribosomes* (involved in protein synthesis), the *endoplasmic reticulum* (a complex membrane system, used in the transport of molecules and the communication among the cell components), and the *lysosomes* (membrane-bound vesicles, mainly involved in the digestion of molecules).

We stress the fact that a cell has a large number of *compartments*, separated (delimited/enclosed/protected) by *membranes*, and that all membranes have, among others, two fundamental functions: *separators* and *channels of communication* among the separated regions; a third important role, also important from the computability point of view, is that of a supporter (and enhancer/catalyst) for most of the reactions which take place in the compartments of a cell.

All internal membranes have in principle the same structure as the plasma membrane. That is why we recall some facts concerning the structure and the functioning of the plasma membrane which will be relevant for the theoretic developments in the subsequent chapters of the book.

2.1.1 The Structure of the Plasma Membrane

There is some variability in the composition of the plasma membrane among different species, but all membranes have the same basic structure. The currently accepted model of this structure is the *fluid–mosaic model*, proposed in 1972 by S. Singer and G. Nicolson [224]. According to this model, a membrane is a phospholipid bilayer in which protein molecules (as well as other molecules, such as cholesterol, steroids, and others) are totally or partially embedded – this is schematically illustrated in Figure 2.1.

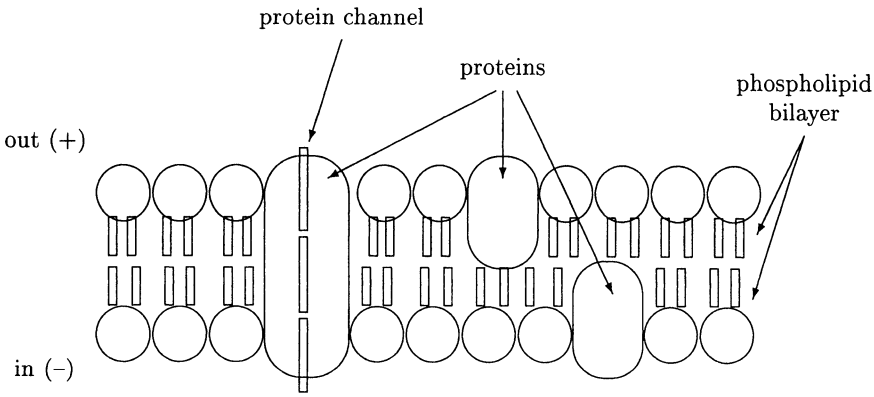


Fig. 2.1. The schematic structure of a membrane

The phospholipid molecules are composed of two main parts: a polar *head* and a non-polar *tail* – see the pictorial representation in Figure 2.2 (a) and the details in Figure 2.2 (b). The head is composed of a phosphate group and a nitrogen group, and the tail consists of two fatty acid chains; the head is bonded to the tail by a glycerol. Consequently, the heads of the molecules in the two layers are hydrophylic, while the tails are hydrophobic. This explains the arrangement of heads against the aqueous solutions from the inner region (plasma) and from outside the cell. Moreover, the polar heads lead to polarizations of the two sides of the membrane: a positive charge in the outside layer and a negative one in the inner layer of molecules. This facilitates the exit of negative ions and the entrance of positive ions. The polarization of the heads (as well as some hydrogen bonding and electrostatic interactions with water molecules from the cytoplasma and from the outer space) stabilizes the bilayer.

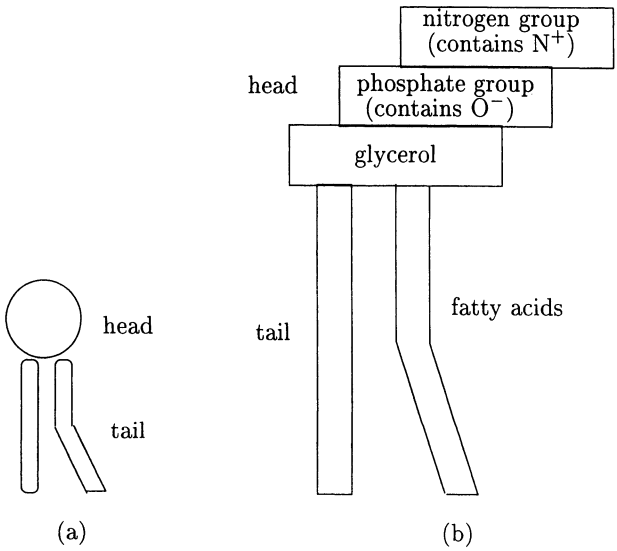


Fig. 2.2. A phospholipid molecule

However, the phospholipid molecules can move in the two layers, but, because of the heads' polarization and the mentioned chemical and electrostatic interactions, they always remain in the planes of the two layers. That is why the model is called the fluid–mosaic one. This movement of molecules on the surface of the membrane makes possible also the movement of proteins (and of other compounds), which is important from the point of view of intra- and inter-cellular communication.

The two layers differ not only in their polarization, but also in their composition. For instance, the glycolipids placed in the external layer have hydrophilic heads which join and form a protecting envelope on the outer surface

of the plasma membrane. In exchange, the proteins placed in the inner layer have a more accentuated catalytic role. Some of these inner proteins serve as links to the cytoskeletal filaments, thus stabilizing mechanically the internal structure of the cell, while the outside proteins serve as links to the extracellular matrix, thus stabilizing the place of cells in tissues.

2.1.2 Trans-membrane Transport

The (plasma) membrane is only partially permeable. For instance, small non-charged molecules, particularly if they are lipid soluble, cross the membrane almost freely. Larger molecules can cross a membrane only with assistance, while charged ions pass selectively from one region to another one.

The trans-membrane transfer of molecules can take place in three main ways: by *passive transport*, by *active transport*, and by *vesicle-mediated transport*.

The passive transport is especially done by diffusion towards a region of lower concentration. It is an effect of the random movement of molecules and proceeds without energy consumption. The speed of transferring molecules through membranes depends on the difference in concentration among the two compartments separated by the membrane, by the temperature (a higher thermal agitation favours the diffusion), the size of molecules (small molecules travel faster), and the characteristics of the membrane. Osmosis is a variant of diffusion, which transports water through membranes, again due to gradient reasons. There is a way to transport molecules by means of certain carrier proteins, also without using metabolic energy, and for this reason is considered a passive transport as well.

The active transport essentially uses energy (provided by adenosine triphosphate molecules, abbreviated ATP molecules) in passing molecules through membranes, and this makes possible the transport of molecules from a compartment of lower concentration to a compartment of higher concentration (hence against the gradient). The most important active membrane transfer is done by *proteins* present in various numbers in the membranes of living cells. For instance, many macromolecules, unable to travel by passive transport, can pass through such "gates".

Actually, there are two main types of proteis involved in trans-membrane transport, some of which just select the moving objects by their size (they are usually called *protein channels*), and others, called *carrier proteins*, which interact with specific molecules, possibly also modifying them when assisting them to cross the membrane.

We emphasize the important fact that the protein channels are *selective*: certain chemicals can pass and certain others cannot pass through these channels. (The specificity of the carrier proteins is very high; for instance, glucose can cross a membrane hundreds of times faster than other sugar molecules of an identical size.) Also, these channels can be closed in the case when an

undesired chemical has appeared in the environment and they can be opened again after the dangerous chemical disappears.

A very interesting situation appears when the transport across a membrane involves two chemicals at the same time, which "collaborate" in their passage through the membrane. There are two possibilities: either the two chemicals are transported in the same direction, or the two chemicals pass in opposite directions. The former case is called *symport*, the latter one is named *antiport*. For uniformity, the case when only one ion/molecule passes through a membrane is called *uniport*.

Figure 2.3 (a) and (b) illustrates the symport and antiport, and also indicates a special case of symport, mainly present in bacteria, where the passage of two chemicals (A and B in figure) through the membrane is possible only in the presence of a promoter (denoted here by C), a protein bound to the membrane in the vicinity of the protein channel (note that this protein does not cross the membrane). (For the sake of accuracy, TRAP means "tripartite ATP periplasmic" transport.)

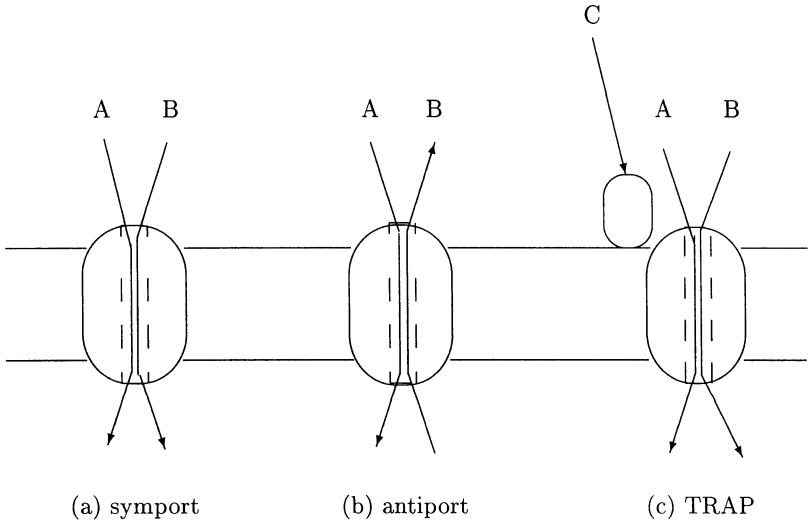


Fig. 2.3. Symport, antiport, promoted symport (TRAP)

An example of antiport is the exchange of Na^+ with K^+ through the so-called *sodium–potassium pump*: the K^+ ion enters the cell at the same time as the Na^+ ion exits, and the process requires energy for a conformational change of the protein channel. This pump is active in all cells, but it is particularly active in nerve and muscle cells. It is interesting to note that this antiport transfer is coupled with a uniport transfer in order to accomplish the important task of transferring salt (Na^+Cl^-) across membranes: first Na^+ positive ions are pumped into a region and then chloride negative ions Cl^- are simply attracted by the positive ones, and they diffuse towards them.

An example of symport appears in the way the bacterium *Escherichia coli* brings lactose from its environment, passing at the same time lactose and protons through the membrane. The process is rather tricky: first, protons are pushed outside the cell by so-called proton pumps, creating a difference of concentration of H^+ between the inner and the outer regions; then the protons are coupled with lactose molecules with the help of certain proteins (lactose permease), and, because of the difference in the concentration of H^+, the lactose–proton pair is transported into the cell.

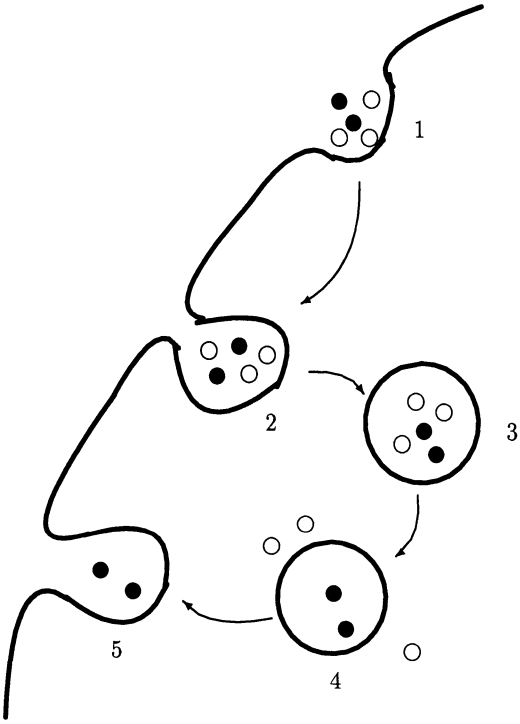


Fig. 2.4. Sequence endocytosis–exocytosis

Many macromolecules are too large to be transported by carrier proteins, but they can be transported through membranes by means of vesicle formation. The process is called *exocytosis* when the macromolecules are sent out of a compartment (this is often the case with secretions sent to the plasma from the Golgi apparatus), and *endocytosis* when substances are introduced in a compartment. Endocytosis is called *phagocytosis* in the case when the material taken in is large (such as a food particle) and *pinocytosis* when vesicles are formed around very small particles, dissolved in a solution. (Phagocytosis can be seen with a light microscope, but pinocytosis can be seen only with an electron microscope.) Besides single chemical transport, by exocytosis and endocytosis we can transport packages of chemicals.

Sometimes, in so-called receptor-mediated endocytosis, we have a sequence of operations, with a sorting of the carried molecules. First, by endocytosis (facilitated by certain chemicals: ligands, receptors, a protein called clathrin, etc.) one brings a variety of molecules into the cell. Then, after their job is done, the clathrin and the ligands are released and made available for a new transport. Part of the contents of the vesicle is also released into the cytoplasm, but part of it will be sent out of the cell, the vesicle will fuse again with the cell membrane and, by exocytosis, will expel the remaining molecules. Figure 2.4 schematically illustrates this sequence of operations.

Besides the trans-membrane transport of chemicals, other important functions of membrane proteins are the *catalytic* activity (certain reactions can take place only in the presence of certain enzymatic proteins), *recognition* and *binding* activities (certain proteins recognize certain molecules or even catch them and keep them bound to the membrane).

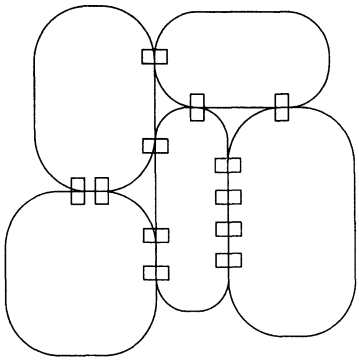


Fig. 2.5. Inter-cellular communication

In animals and plants, the cells are arranged in multi-cellular structures, such as tissues, organs, and organisms, which are held together by connections between the cells. In the case of animal cells there are three types of connections: *adhesion* junctions, *tight* junctions, and *gap* junctions. The first two are meant to keep the cells together – as a matter of fact, tight junctions are much stronger and also serve to form barriers against certain chemicals (this is, for instance, the case of internal cells of the intestine, which keep the gastric juices out of the body). For the purposes of this book, the gap junctions are of more interest.

Because of the fluid nature of the membrane, the protein channels can change their location within the membrane and thus, when two cells come into contact, it may be that identical protein channels from the adjacent membranes combine together, establishing a common channel, which makes possible the transfer of chemical compounds from one cell to another one. After establishing some channels of this type, further channels are enhanced,

as the two membranes remain adjacent, bound by the proteins already in contact. In this way, a complex net of channels can be established, providing the way for inter-cellular communication. Figure 2.5 illustrates this idea. The control of these common protein channels, opening or closing in dependence on the contents of the communicating cells, is a common phenomenon; if a "poison" enters one of the cells, then the "gates" to the neighbouring cells are closed, so as not to allow the poison to spread around, and the channels are reopened when the undesired chemical vanishes.

2.1.3 Cellular Division: Mitosis

One of the most important roles of cells is reproduction, and this is achieved through the division of a cell into two identical copies. The biological term for this process is called *mitosis*, and, in fact, it consists of a sequence of several steps. During the first step, called *interphase*, the cell prepares itself for duplication by replicating its genetic information, as well as replicating all internal organelles. In the next step, the *prophase*, the already-replicated chromosomes are concentrated within the nuclear membranes, and prepared for separation. There follows the *metaphase*, when the chromosomes are separated towards two poles of the cell, which start to get farther apart, a process completed during *anaphase*. During the next step, the *telophase*, new nuclear envelopes start to appear around the two sets of chromosomes; one ends with two separate nuclei, each having its own nucleolus, but still placed in the same cell. The cell itself will split into two identical cells during the next step, the *cytokenesis*, which completes the mitosis. In this way, two cells are created, and each of them can continue the process, starting the interphase again.

This process is very important for membrane computing, as a way to provide exponentially many identical cells in linear time: in n steps we get 2^n identical cells, and, in an idealized framework, n can be arbitrarily large (in principle, bounded only by space restrictions and the supply of necessary nutrients).

Certain cells (this is the case of Baker's yeast) can multiply by *budding*, in such a way that in one step a given cell will produce several new cells. The new cells are not identical to the original cells, but after a specific time they get matured and can produce further cells by budding. In some sense, in this way we get d^n cells in n time units, for d possibly greater than 2; this means that by budding we obtain an exponential working space in a "more efficient" way than by cellular division.

Of course, the biological knowledge of the cell membrane is much richer, but we have recalled here, in this sketched manner, only the facts which seem to be of a direct interest for membrane computing and/or have already been invoked in papers dealing with membrane systems. A closer look at the biology of the membrane would be useful for further developments of membrane computing, especially if aiming to devise a model with some relevance

from a biological point of view (a necessary – but not sufficient! – step towards *relevance* is *adequacy*), and it is necessary in the case of any attempt to implement membrane computing in bio-ware. The books mentioned at the beginning of this section provide much more information about cells and their membranes.

2.2 The Neuron

We mentioned in the Introduction that besides cell-like structures we also consider membrane systems with the membranes arranged in a network, as in a tissue (having in mind inter-cellular communication, as illustrated in Figure 2.5), or the neurons in the brain. That is why we briefly present here the structure and the functioning of a neuron, basically following computer science sources (in particular, [7] and [8]).

In a (very) schematic graphical representation, a neuron looks as sketched in Figure 2.6.

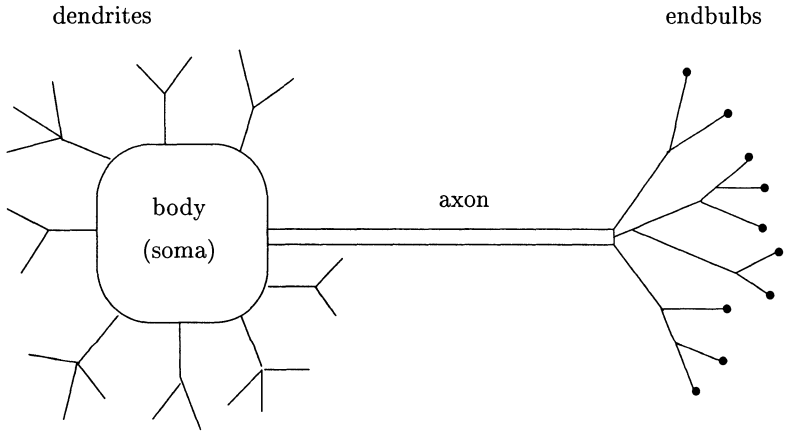


Fig. 2.6. A schematic representation of a neuron

As any cell, a neuron has a *body* (also called *soma*) containing a nucleus (not of interest for what follows), and a membrane. In the case of neurons, the membrane is prolonged by two important classes of fibers, the *dendrites*, which form a very fine filamentary bush around the body of the neuron, and the *axon*, a unique, long filament, which in turn also ends with a fine filamentous bush; each of the filaments from the end of the axon is terminated with a small bulb.

It is by means of these endbulbs and the dendrites that the neurons are linked to each other: the (electrical and chemical) impulses are sent through the axon, from the body of the neuron to the endbulbs, and the endbulbs transmit the impulses to the neurons whose dendrites they touch. Such a

contact between an endbulb of an axon and the dendrites of another neuron is called a *synapse*. A neuron can be linked in this way to several other neurons, hence each neuron can receive impulses from several neurons and can transmit impulses to several neurons.

The transmission of impulses from one neuron to another one is done, roughly speaking, in the following way. A neuron will be "fired" only if it gets sufficient excitation through its dendrites. These excitations should come more or less together, in a short period of time, called the *period of latent summation*. Moreover, the input impulses can be of two types, *excitatory* and *inhibitory*, and in order to get the neuron excited it is necessary that the "algebraic sum" of impulses exceeds a given *threshold* specific to the neuron.

After firing a neuron and exciting it, there is a small interval of time necessary to synthesize the impulse to be transmitted to the neighbouring neurons through the axon; also, there is a small interval of time necessary for the impulse to reach the endbulbs of the axon. After an impulse is transmitted through the axon, there is a time called the *refractory period* during which the axon cannot transmit another impulse.

In short, we have inputs of various types (chemical or electrical, also different in their intensities, possibly also different qualitatively, depending on the source – think of the fact that some neurons can have inputs from various sensory areas, the skin, the eyes, the muscles, etc.), in packages (each neuron has several dendrites and they can get impulses from several other neurons or from several sensory areas), and the neuron synthesizes an impulse which is transmitted to the neurons to which it is linked by synapses; the synthesis of an impulse and its transmission to the adjacent neurons are done according to certain "states" of the neuron. At different moments, depending on its previous actions, the neuron can act differently.

In Section 6.3 we will make essential use of these details.

2.3 Elements of Computability

We recall here several notions and results from automata and formal language theory, as well as from complexity theory. We present all the basic notions and notations, so that, at least in principle, the reader with a minimal mathematical background can understand at least the definitions of models in the subsequent chapters (which are never fully formalized at the time of introducing them), as well as the results – not necessarily the proofs, but the proofs are addressed to the reader with theoretical computer science interests. In certain cases which are more important for membrane computing (such a case is that of matrix grammars), we enter into some detail, but for further information (including precise historical references), the reader is referred to the many monographs in this area, starting with [77], and finishing with the comprehensive handbooks [219, 239]. The Bibliography contains many other

titles which can be useful to the reader who feels the need to enter certain topics in detail: [3, 44, 78, 84, 240].

2.3.1 Basic Notions and Notations

We use standard set-theoretic mathematical notations: $\emptyset$ is the empty set, the fact that an element a belongs to a set M is denoted by $a \in M$, the inclusion of a set M_1 in a set M_2 is written $M_1 \subseteq M_2$, and the strict inclusion is indicated by $M_1 \subset M_2$; the union, intersection, difference, and Cartesian product of two sets M_1, M_2 are written as $M_1 \cup M_2, M_1 \cap M_2, M_1 - M_2$, and $M_1 \times M_2$, respectively. The set of all subsets of a set M (also called the *power set* of M) is denoted by $\mathcal{P}(M)$. The set of natural numbers, $\{0, 1, 2, \ldots\}$, is denoted by $\mathbf{N}$. The cardinality (the number of elements) of a finite set M is denoted by $card(M)$.

An *alphabet* is a finite non-empty set of abstract symbols. For an alphabet V we denote by V^* the set of all strings of symbols from V. The empty string is denoted by λ. Mathematically speaking, V^* is the free monoid generated by V under the operation of *concatenation*. (The unit element of this monoid is λ.) The set of non-empty strings over V, that is $V^* - \{\lambda\}$, is denoted by V^+. Each subset of V^* is called a *language* over V. A language which does not contain the empty string (hence it is a subset of V^+) is said to be *λ-free*. A set/collection of languages is usually called a *family* of languages.

If $x = x_1x_2$, for some $x_1, x_2 \in V^*$, then x_1 is called a *prefix* of x and x_2 is called 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, substrings of a string x are denoted by $Pref(x), Suf(x), Sub(x)$, respectively.

The *length* of a string $x \in V^*$ (the number of occurrences in x of symbols from V) is denoted by $|x|$. The number of occurrences of a symbol $a \in V$ in $x \in V^*$ 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. If FL is a family of languages, then we denote by NFL the family of length sets of languages in FL.

The set of symbols occurring in a string x is denoted by $alph(x)$. For a language $L \subseteq V^*$, we denote $alph(L) = \bigcup_{x \in L} alph(x)$. Observe that $alph(L)$ may be a proper subset of V.

The *Parikh vector* associated with a string $x \in V^*$ with respect to the alphabet $V = \{a_1, a_2, \ldots, a_n\}$ is $\Psi_V(x) = (|x|_{a_1}, |x|_{a_2}, \ldots, |x|_{a_n})$ (note that the ordering of the symbols from V is relevant); this is called the *Parikh mapping* associated with V. For $L \subseteq V^*$ we define $\Psi_V(L) = \{\Psi_V(x) \mid x \in L\}$; we say that $\Psi_V(L)$ is *the Parikh image* of the language L. If FL is a family of languages, then we denote by $PsFL$ the family of Parikh images of languages in FL.

A set M of vectors in $\mathbf{N}^n$, for some $n \geq 1$, is said to be *linear* if there are some vectors $v_i \in \mathbf{N}^n$, $0 \leq i \leq m$, such that

$$M = \{v_0 + \sum_{i=1}^{m} \alpha_i v_i \mid \alpha_1, \ldots, \alpha_m \in \mathbf{N}\}.$$

A finite union of linear sets is said to be *semilinear*..

A language $L \subseteq V^*$ is semilinear if $\Psi_V(L)$ is a semilinear set. The family of semilinear languages is denoted by $SLIN$.

2.3.2 Operations with Strings and Languages

The languages are sets, hence we can perform the usual set operations: union, intersection, difference, and complementation (with respect to V^*, if $L \subseteq V^*$). There are also several operations which are specific to languages as sets of strings (hence specific to strings).

The *concatenation* of L_1, L_2 is $L_1L_2 = \{xy \mid x \in L_1, y \in L_2\}$. This operation can be iterated: take by convention $L^0 = \{\lambda\}$; then, for all $i \geq 0$ we define $L^{i+1} = LL^i$. The union of all languages $L^i, i \geq 0$, is denoted by L^* and is called the *Kleene closure* of L.

Thus, L^* is the set of all strings obtained by concatenating any – finite – number of strings from L, including the case of no-string concatenation, which, by convention, is the empty string. If we ignore the special case of $i = 0$, then we get $L^+ = \bigcup_{i \geq 1} L^i$; this is called the +-Kleene closure of L (observe the consistency of these notations with V^* and V^+).

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

A morphism $h : V^* \longrightarrow U^*$ is called a *coding* if $h(a) \in U$ for each $a \in V$ and a *weak coding* if $h(a) \in U \cup \{\lambda\}$ for each $a \in V$. (Thus, a coding only renames the symbols, while a weak coding can also erase some of them.) If $h : (V_1 \cup V_2)^* \longrightarrow V_1^*$ is the weak coding defined by $h(a) = a$ for $a \in V_1$, and $h(a) = \lambda$ otherwise, then we say that h is a *projection* (associated with V_1) and we denote it by pr_{V_1}.

The *left derivative* of a language $L \subseteq V^*$ with respect to a string $x \in V^*$ is $\partial_x^l(L) = \{w \in V^* \mid xw \in L\}$. The *right derivative* is defined in a symmetric manner: $\partial_x^r(L) = \{w \in V^* \mid wx \in L\}$.

The *mirror image* of a string $x = a_1a_2 \ldots a_n$, where $a_i \in V, 1 \leq i \leq n$, is $mi(x) = a_n \ldots a_2a_1$.

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

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

A language that can be obtained from the letters of an alphabet V and from λ by using finitely many times the operations of union, concatenation, and Kleene $*$ is called *regular*; also the empty language is said to be regular.

2.3.3 Chomsky Grammars

Generally speaking, a *grammar* is a (finite) device *generating* in a well-specified sense the strings of a language (hence defining a set of "syntactically correct" strings). The Chomsky grammars are particular cases of *rewriting systems*, where the operation used in processing the strings is the rewriting (the replacement of a "short" substring of the processed string by another short substring), and they have been introduced as a possible model of the syntax of natural languages (and developed mainly in relation with the syntax of programming languages).

In order to specify a rewriting system we have to specify an *alphabet* V, a set of *axioms*, in the form of a subset A of V^*, and a set R of *rewriting rules*, in the form $r : u \to v$, for u, v strings from V^*. For a string $w = w_1uw_2$ we can rewrite u by means of v by using the rule r, and we get the string $z = w_1vw_2$. One writes $w \Longrightarrow_r z$, or simply $w \Longrightarrow z$ if r is understood, and one calls this operation a *direct derivation step*. The reflexive and transitive closure of this relation is denoted by $\Longrightarrow^*$. Thus, 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 $z_1, \ldots, z_k \in V^*$, $k \geq 1$. If the rewriting system is denoted by $G = (V, A, R)$, then the *language generated* by G is defined by $L(G) = \{z \in V^* \mid w \Longrightarrow^* z \text{ for some } w \in A\}$. That is, $L(G)$ contains all axiom strings, as well as all strings which can be obtained from them by iteratively applying the rules from R.

A rewriting system as above is also called a *pure grammar*. A Chomsky grammar is not "pure", in the sense that in the alphabet V one distinguishes two types of symbols, terminal and non-terminal symbols. Specifically, we write a *Chomsky grammar* as a quadruple $G = (N, T, S, R)$, where N, T are disjoint alphabets, $S \in N$, and R is a finite set of *rewriting rules* (we also say *productions*) of the form $u \to v$, with $u, v \in (N \cup T)^*$ and u contains at least one non-terminal symbol.

The alphabet N is called the *non-terminal alphabet*, T is the *terminal alphabet*, and S is the *axiom*.

The derivation relation $\Longrightarrow$ is defined as in any rewriting system. Each string $w \in (N \cup T)^*$ such that $S \Longrightarrow_G^* w$ is called a *sentential form*.

The language generated by G, denoted by $L(G)$, is defined by

$$L(G) = \{x \in T^* \mid S \Longrightarrow^* x\}.$$

Thus, the important difference with the general case of pure grammars is the fact that only strings consisting of terminal symbols are accepted in the generated language.

Two grammars G_1, G_2 are called *equivalent* if $L(G_1) - \{\lambda\} = L(G_2) - \{\lambda\}$ (the two languages coincide modulo the empty string).

This **convention** will be adopted throughout the book: when comparing the languages generated (or accepted) by two devices we ignore the empty string.

If in $x \Longrightarrow y$ above we have $x = x_1 u x_2$, with $x_1 \in T^*$, then the derivation step is said to be done in the *leftmost* manner and we write $x \Longrightarrow_{left} y$. The language generated by the grammar G by means of derivations where every step is leftmost is denoted by $L_{left}(G)$.

Example 2.3.1. For the grammar

$$G_1 = (\{S\}, \{a, b\}, S, \{S \to aSb,\ S \to ab\})$$

we obtain

$$L(G_1) = \{a^n b^n \mid n \geq 1\}.$$

Indeed, each derivation in G_1 consists of $m \geq 0$ applications of the rule $S \to aSb$, and this produces the string $a^m S b^m$, and ends by using the rule $S \to ab$, which leads to the terminal string $a^{m+1} b^{m+1}$ to which no rule can be applied. Thus, each string generated by G is of the form $a^n b^n$, for some $n \geq 1$; conversely, each string of this form can be generated by G_1.

Example 2.3.2. The grammar

$$G_2 = (\{S, B\}, \{a, b, c\}, S, \{S \to aSBc,\ S \to abc,\ cB \to Bc,\ bB \to bb\})$$

generates the language

$$L(G_2) = \{a^n b^n c^n \mid n \geq 1\}.$$

The proof of this assertion is left to the reader.

According to the form of their rules, the Chomsky grammars are classified as follows. A grammar $G = (N, T, S, R)$ is called:

- *length-increasing*, if for all $u \to v \in R$ we have $|u| \leq |v|$.
- *context-sensitive*, if each $u \to v \in R$ has $u = u_1 A u_2, v = u_1 x u_2$, for $u_1, u_2 \in (N \cup T)^*, A \in N$, and $x \in (N \cup T)^+$.
- *context-free*, if each production $u \to v \in R$ has $u \in N$.
- *linear*, if each rule $u \to v \in R$ has $u \in N$ and $v \in T^* \cup T^* N T^*$.
- *right-linear*, if each rule $u \to v \in R$ has $u \in N$ and $v \in T^* \cup T^* N$.
- *left-linear*, if each rule $u \to v \in R$ has $u \in N$ and $v \in T^* \cup N T^*$.
- *regular*, if each rule $u \to v \in R$ has $u \in N$ and $v \in T \cup TN$.

In context-free grammars (hence also in linear, right- and left-linear, and in regular grammars) we also allow rules of the form $A \to \lambda$, but only when this is useful for simplifying the grammar: because of the convention that the empty string is not counted when comparing the languages generated by two grammars, such rules are not necessary in context-free grammars (or in their restricted subclasses).

The arbitrary, length-increasing, context-free, and regular grammars are also said to be of *type* 0, *type* 1, *type* 2, and *type* 3, respectively.

Note that the grammar G_1 from Example 2.3.1 is linear (but not regular), while the grammar G_2 from Example 2.3.2 is length-increasing (but not context-free).

A context-free grammar $G = (N, T, S, R)$ is said to be *reduced* if all symbols from N are accessible from S and all of them can be rewritten into a terminal string; that is, for each $A \in N$ there is a derivation of the form $S \Longrightarrow^* uAv \Longrightarrow^* uxv$, where x is a terminal string.

The family of languages generated by length-increasing grammars is equal to the family of languages generated by context-sensitive grammars; the families of languages generated by right- or by left-linear grammars coincide and they are equal to the family of languages generated by regular grammars, as well as to the family of regular languages (as defined at the end of Subsection 2.3.2).

We denote by *RE, CS, CF, LIN,* and *REG* the families of languages generated by arbitrary, context-sensitive (hence also by length-increasing), context-free, linear, and regular (hence also by right- and left-linear) grammars, respectively (RE stands for *recursively enumerable*). By FIN we denote the family of finite languages.

Therefore, for the grammars from Examples 2.3.1 and 2.3.2 we have $L(G_1) \in LIN$ and $L(G_2) \in CS$.

The following strict inclusions hold:

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

This is *the Chomsky hierarchy*, the constant reference for investigations related to the power of membrane systems (and of any new types of computing devices). This important role of Chomsky hierarchy is due to several reasons: (1) the family RE of languages generated by type-0 Chomsky grammars is exactly the family of languages which are recognized by Turing machines (see below), and according to the Turing–Church thesis this is the maximal level of algorithmic computability; (2) the Chomsky hierarchy is well structured, hence we have a detailed classification of computing machineries (in between the above-mentioned classes of grammars there are other types of grammars which were considered in the literature); (3) the formal language theory is a well-developed fundamental branch of theoretical computer science, very well investigated (some decades ago it was said that formal language theory is "the flower of theoretical computer science" – Aho, Salomaa, etc.).

Table 2.1 indicates some basic closure properties of the families of languages from the Chomsky hierarchy.

Because RE is the family of languages recognized (computed) by Turing machines, NRE, the family of length sets of languages in RE, is the family of sets of numbers recognized (computed) by Turing machines.

When considering the length sets of languages, the Chomsky hierarchy becomes

$$NFIN \subset NREG = NLIN = NCF \subset NCS \subset NRE,$$

because the length set of a context-free language can be obtained as the length set of a regular language.

Table 2.1. Closure properties of the families in the Chomsky hierarchy

	RE	*CS*	*CF*	*LIN*	*REG*	*FIN*
Union	Yes	Yes	Yes	Yes	Yes	Yes
Intersection	Yes	Yes	No	No	Yes	Yes
Complement	No	Yes	No	No	Yes	No
Concatenation	Yes	Yes	Yes	No	Yes	Yes
Kleene closure $*$	Yes	Yes	Yes	No	Yes	No
Intersection with regular languages	Yes	Yes	Yes	Yes	Yes	Yes
Morphisms	Yes	No	Yes	Yes	Yes	Yes
λ-free morphisms	Yes	Yes	Yes	Yes	Yes	Yes
Left/right derivative	Yes	Yes	Yes	Yes	Yes	Yes
Mirror image	Yes	Yes	Yes	Yes	Yes	Yes
Permutation	Yes	Yes	No	No	No	Yes

We mention here a major feature of the formal grammar approach to computability, which can be considered a drawback from the point of view of practical computer science: the language generated by a grammar consists of all strings obtained at the end of "successful" (terminal) derivations; the "unsuccessful" derivations are ignored, hence we adopt a positivistic/opportunistic approach, which, because of the non-deterministic behaviour of grammars (the rule to be used at a given step and the substring to be rewritten by that rule are non-deterministically chosen), is easy to handle at the mathematical level, but not at the practical level.

2.3.4 Characterizations, Necessary Conditions

From many points of view, in particular in proofs, it is useful to characterize a family of languages by grammars of the corresponding type, but of particular forms, called *normal forms*. Such characterizations (normal forms) exist for all families in the Chomsky hierarchy, but we consider here only some of those referring to context-free and to recursively enumerable languages.

Theorem 2.3.1 (Chomsky normal form). *For every context-free grammar G an equivalent context-free grammar $G' = (N, T, S, R)$ can be effectively constructed, with the rules from R of the forms $A \to BC$ and $A \to a$, where $A, B, C \in N$ and $a \in T$.*

Very useful in the subsequent chapters will be the normal forms for type-0 grammars.

Theorem 2.3.2 (Kuroda normal form). *For every type-0 grammar G, an equivalent grammar $G' = (N, T, S, R)$ can be effectively constructed, with the rules in R of the forms $A \to BC$, $A \to a$, $A \to \lambda$, and $AB \to CD$, where $A, B, C, D \in N$ and $a \in T$.*

Theorem 2.3.3 (Penttonen normal form). *For every type-0 grammar G, an equivalent grammar $G' = (N, T, S, R)$ can be effectively constructed, with the rules in R of the forms $A \to BC$, $A \to a$, $A \to \lambda$, and $AB \to AC$, where $A, B, C \in N$ and $a \in T$.*

Similar results hold true for length-increasing grammars, but without using rules of the form $A \to \lambda$.

Theorem 2.3.4 (Geffert normal forms). (1) *Each recursively enumerable language can be generated by a grammar $G = (N, T, S, R)$ with $N = \{S, A, B, C\}$ and the rules in R of the forms $S \to uSv$ and $S \to x$, where $u, v, x \in (T \cup \{A, B, C\})^*$, and only one non-context-free rule, $ABC \to \lambda$.*

(2) *Each recursively enumerable language can be generated by a grammar $G = (N, T, S, R)$ with $N = \{S, A, B, C, D\}$ and the rules in R of the forms $S \to uSv$ and $S \to x$, where $u, v, x \in (T \cup \{A, B, C, D\})^*$, and only two non-context-free rules, $AB \to \lambda, CD \to \lambda$.*

A linear grammar with only one non-terminal is said to be *minimal.* Thus, the Geffert normal forms say that each recursively enumerable language can be obtained from a minimal linear language by applying the reduction rule $ABC \to \lambda$, or the reduction rules $AB \to \lambda, CD \to \lambda$.

We recall now two results useful in proving that a given language is not context-free.

Theorem 2.3.5 (Bar–Hillel/pumping lemma). *If $L \in CF, L \subseteq V^*$, then there are $p, q \in \mathbf{N}$ such that every $z \in L$ with $|z| > p$ can be written in the form $z = uvwxy$, with $u, v, w, x, y \in V^*$, $|vwx| \leq q, vx \neq \lambda$, and $uv^iwx^iy \in L$ for all $i \geq 0$.*

Theorem 2.3.6 (Parikh theorem). *Every context-free language is semilinear. Otherwise written, we have the relation $CF \subset SLIN$.*

Corollary 2.3.1. *Every context-free language over the one-letter alphabet is regular.*

Theorem 2.3.5 has a particular form for regular languages.

Theorem 2.3.7 (Pumping lemma for regular languages). *If $L \in REG$, $L \subseteq V^*$, then there is $p \in \mathbf{N}$ such that every $z \in L$ with $|z| > p$ can be written in the form $z = uvw$, with $u, v, w \in V^*$, $|vw| \leq p, v \neq \lambda$, and $uv^iw \in L$ for all $i \geq 0$.*

The conditions of Theorems 2.3.5 and 2.3.6 (2.3.7) are only necessary, not sufficient, for a language to be context-free (respectively, regular).

Using these necessary conditions (and other tools, not specified here) the following facts can be proved (some of these languages will be used in subsequent chapters of the book):

$$\begin{aligned}
L_1 &= \{a^n b^n \mid n \geq 1\} \in LIN - REG,\\
L_2 &= L_1 L_1 \in CF - LIN,\\
L_3 &= \{a^n b^n c^n \mid n \geq 1\} \in CS - CF,\\
L_4 &= \{xcx \mid x \in \{a,b\}^*\} \in CS - CF,\\
L_5 &= \{a^{2^n} \mid n \geq 1\} \in CS - CF,\\
L_6 &= \{a^n b^m c^n d^m \mid n,m \geq 1\} \in CS - CF,\\
L_7 &= \{a^n b^m \mid n \geq 1, 1 \leq m \leq 2^n\} \in CS - CF,\\
L_8 &= \{a^n b^m c^p \mid 1 \leq n \leq m \leq p\} \in CS - CF,\\
L_9 &= \{x \in \{a,b\}^* \mid |x|_a = |x|_b\} \in CF - LIN,\\
L_{10} &= \{x \in \{a,b,c\}^* \mid |x|_a = |x|_b = |x|_c\} \in CS - CF.
\end{aligned}$$

All these languages are semilinear, with the exception of L_5 and L_7.

The *Dyck language*, D_n, over $T_n = \{a_1, a_1', \ldots, a_n, a_n'\}$, $n \geq 1$, is the context-free language generated by the grammar

$$G = (\{S\}, T_n, S, \{S \to \lambda,\ S \to SS\} \cup \{S \to a_i S a_i' \mid 1 \leq i \leq n\}).$$

Intuitively, the pairs $(a_i, a_i'), 1 \leq i \leq n$, can be viewed as left and right parentheses, of different kinds (with different labels). Then D_n consists of all strings of correctly nested parentheses.

When $n = 1$ one also can write $T_1 = \{a, b\}$ and the corresponding Dyck language is denoted by $D_{\{a,b\}}$.

2.3.5 Lindenmayer Systems

Lindenmayer systems – usually called L systems – are (pure) rewriting systems introduced with the aim of modelling the development of multi-cellular organisms. The main difference with Chomsky grammars is the parallelism: in a derivation step one rewrites all symbols of the string.

A 0L (zero-interactions Lindenmayer) system is a construct $G = (V, w, R)$, where V is an alphabet, $w \in V^*$ (axiom), and R 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 R (we say that R is *complete*). For $w_1, w_2 \in V^*$ we write $w_1 \Longrightarrow w_2$ if $w_1 = a_1 \ldots a_n, w_2 = v_1 \ldots v_n$, for $a_i \to v_i \in R, 1 \leq i \leq n$. The language generated by G is $L(G) = \{x \in V^* \mid w \Longrightarrow^* x\}$.

If for each rule $a \to v \in R$ we have $v \neq \lambda$, then we say that G is *propagating* (non-erasing); if for each $a \in V$ there is only one rule $a \to v$

in R, then G is said to be *deterministic*. If we distinguish a subset T of V (then we write $G = (V, T, w, R)$) and we define the generated language as $L(G) = \{x \in T^* \mid w \Longrightarrow^* x\}$, then we say that G is *extended*. The family of languages generated by 0L systems is denoted by $0L$; we add the letters P, D, E in front of $0L$ if propagating, deterministic, or extended 0L systems are used, respectively.

A *tabled* 0L system, abbreviated T0L, is a system $G = (V, w, R_1, \ldots, R_n)$, such that each triple $(V, w, R_i), 1 \leq i \leq n$, is a 0L system; each R_i is called a *table*, $1 \leq i \leq n$. The language generated by G is defined by

$$L(G) = \{x \in V^* \mid w \Longrightarrow_{R_{j_1}} w_1 \Longrightarrow_{R_{j_2}} \cdots \Longrightarrow_{R_{j_m}} w_m = x, \\ m \geq 0, 1 \leq j_i \leq n, 1 \leq i \leq m\}.$$

(Each derivation step is performed by the rules of the same table.)

A T0L system is deterministic/propagating/extended when each of its tables is deterministic/propagating/extended.

The family of languages generated by T0L systems is denoted by $T0L$; the families $ET0L, EDT0L$, etc., are obtained in the same way as $E0L, ED0L$, etc., by adding the letters P, D, E in front of $T0L$.

Example 2.3.3. The D0L system

$$G = (\{a\}, a, \{a \to aa\})$$

generates the language $\{a^{2^n} \mid n \geq 1\}$, which is not context-free (note that this language is also non-semilinear).

Example 2.3.4. The E0L system

$$\begin{aligned} G &= (V, \{a, b, c\}, ABC, R), \\ V &= \{A, A', B, B', C, C', F, a, b, c\}, \\ R &= \{A \to AA',\ A \to a,\ A' \to A',\ A' \to a,\ a \to F, \\ &\quad B \to BB',\ B \to b,\ B' \to B',\ B' \to b,\ b \to F, \\ &\quad C \to CC',\ C \to c,\ C' \to C',\ C' \to c, \\ &\quad c \to F,\ F \to F\}, \end{aligned}$$

generates the non-context-free language $\{a^n b^n c^n \mid n \geq 1\}$ (the reader is asked to check this non-trivial assertion).

The $D0L$ family is incomparable with FIN, REG, LIN, CF, whereas $E0L$ strictly includes the family CF; $ET0L$ is the largest family of Lindenmayer languages (with 0-interactions), it is strictly included in CS, and it is closed under union, intersection with regular languages, arbitrary morphisms, concatenation, and Kleene closure $*$.

Here are some languages which are not in given L families (they will be useful in the subsequent chapters of the book):

$$
\begin{aligned}
L_{11} &= \{a, aa\} \notin D0L, \\
L_{12} &= \{a^m b^n a^m \mid 1 \leq m \leq n\} \notin E0L, \\
L_{13} &= \{a^{2^n 3^m} \mid n, m \geq 1\} \notin E0L, \\
L_{14} &= \{a^n \mid n \text{ is a prime number}\} \notin ET0L, \\
L_{15} &= \{x \in \{a,b\}^* \mid |x|_b = 2^{|x|_a}\} \notin ET0L, \\
L_{16} &= \{x \in \{a,b\}^* \mid |x|_a = 2^n, |x|_b = 3^n, n \geq 1\} \notin ET0L, \\
L_{17} &= \{(ab^n)^n \mid n \geq 1\} \notin ET0L.
\end{aligned}
$$

The language L_{13} is in $ET0L$, hence in this way we obtain the strictness of the inclusion $NE0L \subset NET0L$, while L_{14} illustrates the strictness of the inclusion $NET0L \subset NCS$. Thus, in terms of length sets, we have $NCF \subset NE0L \subset NET0L \subset NCS$, $ND0L - NCF \neq \emptyset$.

A very useful *normal form* for ET0L systems says that for each language $L \in ET0L$, $L \subseteq T^*$, there is an ET0L system with only two tables, $G = (V, T, w, R_1, R_2)$, such that $L = L(G)$.

2.3.6 Finite Automata, Turing Machines

Automata are computing devices which start from the strings over a given alphabet and *analyse* them (we also say *recognize*), telling us whether or not they belong to a specified language.

The five basic families of languages in the Chomsky hierarchy, REG, LIN, CF, CS, RE, are also characterized by recognizing automata. These automata are: the finite automaton, the one-turn pushdown automaton, the pushdown automaton, the linearly bounded automaton, and the Turing machine, respectively. We present here only two of these devices, those which, in some sense, define the two poles of computability: finite automata and Turing machines.

A (non-deterministic) *finite automaton* is a construct $A = (Q, V, s_0, F, \delta)$, where Q and V are disjoint alphabets, $s_0 \in Q, F \subseteq Q$, and $\delta : Q \times V \longrightarrow \mathcal{P}(Q)$; Q is the set of states, V is the alphabet of the automaton, s_0 is the initial state, F is the set of final states, and δ is the transition mapping. If $card(\delta(s,a)) \leq 1$ for all $s \in Q, a \in V$, then we say that the automaton is *deterministic* (in this case, δ is a function from $Q \times V$ to Q). A relation $\vdash$ is defined on the set $Q \times V^*$ in the following way: for $s, s' \in Q, a \in V, x \in V^*$, we write $(s, ax) \vdash (s', x)$ if $s' \in \delta(s,a)$; by definition, $(s, \lambda) \vdash (s, \lambda)$. If $\vdash^*$ is the reflexive and transitive closure of the relation $\vdash$, then the language of the strings recognized by automaton A is defined by

$$L(A) = \{x \in V^* \mid (s_0, x) \vdash^* (s, \lambda), s \in F\}.$$

It is known that both deterministic and non-deterministic finite automata characterize the same family of languages, namely REG. The power of finite automata is not increased if we also allow λ-*transitions*, that is, if δ is defined on $Q \times (V \cup \{\lambda\})$ (the automaton can also change its state when reading no

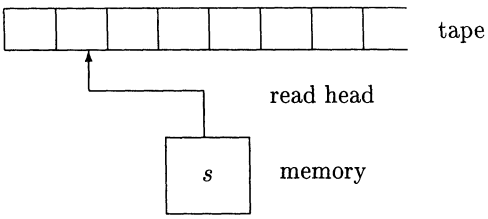


Fig. 2.7. A finite automaton

symbol on its tape), or when the input string is scanned in a two-way manner, going along it to the right or to the left, without changing its symbols.

A useful way to write the transitions of a finite automaton is to use rewriting-like rules: instead of $s' \in \delta(s,a)$, we write $sa \to s'$. Then, the automaton is given in the form $A = (Q, V, s_0, F, R)$, where R is a set of rules of the form $sa \to s'$, for $s, s' \in Q, a \in V$, and the recognized language is defined by $L(A) = \{x \in V^* \mid s_0 x \Longrightarrow^* s_f$, for some $s_f \in F\}$, where $\Longrightarrow$ is defined in the natural way.

We can imagine a finite automaton as in Figure 2.7, where we distinguish the input tape, on whose cells we write the symbols of the input alphabet, the read head, which scans the tape from the left to the right, and the memory, able to hold a state from a finite set of states.

A *Turing machine* is a construct

$$M = (Q, V, T, \flat, s_0, F, \delta),$$

where Q, V are disjoint alphabets (the set of states and the tape alphabet), $T \subseteq V$ (the input alphabet), $\flat \in V - T$ (the blank symbol), $s_0 \in Q$ (the initial state), $F \subseteq Q$ (the set of final states), and δ is a partial mapping from $Q \times V$ to the power set of $Q \times V \times \{L, R\}$ (the move mapping; if $(s', b, d) \in \delta(s, a)$, for $s, s' \in Q, a, b \in V$, and $d \in \{L, R\}$, then the machine reads the symbol a in state s and passes to state s', replaces a with b, and moves the read–write head to the left when $d = L$ and to the right when $d = R$).

If $card(\delta(s,a)) \leq 1$ for all $s \in Q, a \in V$, then M is said to be *deterministic*.

An *instantaneous description* of a Turing machine as above is a string xsy, where $x \in V^*, y \in V^*(V - \{\flat\}) \cup \{\lambda\}$, and $s \in Q$. In this way we identify the contents of the tape, the state, and the position of the read–write head: it scans the first symbol of y. Observe that the blank symbol may appear in x, y, but not in the last position of y; both x and y may be empty. We denote by ID_M the set of all instantaneous descriptions of M.

On the set ID_M one defines the *direct transition* relation $\vdash_M$ as follows:

$$xsay \vdash_M xbs'y \text{ iff } (s', b, R) \in \delta(s, a),$$
$$xs \vdash_M xbs' \text{ iff } (s', b, R) \in \delta(s, \flat),$$

$$xcsay \vdash_M xs'cby \text{ iff } (s', b, L) \in \delta(s, a),$$
$$xcs \vdash_M xs'cb \text{ iff } (s', b, L) \in \delta(s, \flat),$$

where $x, y \in V^*, a, b, c \in V, s, s' \in Q$.

The language recognized by a Turing machine M is defined by

$$L(M) = \{w \in T^* \mid s_0 w \vdash_M^* xsy \text{ for some } s \in F, x, y \in V^*\}.$$

(This is the language of all strings such that the machine reaches a final state when starting to work in the initial state by scanning the first symbol of the string.)

It is also customary to define the language accepted by a Turing machine as consisting of the input strings $w \in T^*$ such that the machine, starting from the configuration $s_0 w$, reaches a configuration where no further move is possible (we say that the machine *halts*); in this case, the set F of final states is no longer necessary. The two modes of defining the language $L(M)$ are equivalent, the identified family of languages is the same, namely RE, and this is true for both deterministic and non-deterministic machines.

The power of Turing machines is not increased if we use several tapes instead of only one; the string to be recognized is written on one of these tapes (called the input tape) and the others are used as working tapes; the same state controls the work of all tapes. Such multi-tape Turing machines are useful when dealing with the complexity of a computation (see Subsection 2.3.11), where it might be necessary to distinguish between the time spent reading the input and the time necessary to actually recognize/process the input.

Graphically, a Turing machine (in the standard form, with only one tape) can be represented as a finite automaton (Figure 2.7). The difference between a finite automaton and a Turing machine is visible only in their functioning: the Turing machine can move its head in both directions and – this is a crucial difference – it can rewrite the scanned symbol, possibly erasing it (replacing it with the blank symbol).

A Turing machine can be also viewed as a mapping-defining device, not just as a mechanism defining a language, but we do not enter into details here (roughly speaking, the computed mapping links two instantaneous descriptions of the tape, an input one and a halting one which gives the value of the function; also, one can use an input tape for the argument and an output tape for the value of the function for the argument written in the first tape; additional working tapes may be used).

When working on an input string a Turing machine is allowed to use as much tape as it needs. Note that finite 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 string is called a *linearly bounded automaton.* These machines characterize the family CS.

There are many models of computability equivalent to Turing machines, but the Turing machine is considered as a standard, because of its elegance and because of its robustness (no reasonable extension can increase its power). Actually, according to the *Turing–Church thesis*, whatever can be algorithmically computed can be also computed by a Turing machine.

A very powerful technique in automata and language theory is to consider the strings over an alphabet with k symbols as numbers in base $k+1$ (none of the symbols stands for zero), to work with numbers, and eventually to return to strings. This is the basic idea of register machines, already investigated in the 1960s, and it is also the proof idea of a central result in regulated rewriting, namely, of the characterization of recursively enumerable languages by context-free matrix grammars with appearance checking – see definitions in the corresponding section below. This idea is useful also in membrane computing, which is why we briefly recall here the definition of *register machines*.

An *n-register machine*, $n \geq 1$, is a construct $M = (R, i_0, f)$, where R is a finite set of *labelled program instructions* of the form $k : (op(i), l, m)$ such that $op(i)$ is an operation on register i of M and k, l, m are labels from a given set of labels for the program instructions of M, $k \neq f$; f is the *final label*, to which we assign the instruction *end*, which halts the program of the register machine M, and i_0 is the *initial label* to start the program. Given an instruction $k : (op(i), l, m)$, if operation $op(i)$ can be applied to register i, then one continues with the instruction with label l, otherwise one continues with the instruction with label m.

The program instructions $(op(i), l, m)$ are of the following forms:

- $(S(i), l, m)$: if possible (that is, if the contents of register i are greater than zero), subtract 1 from register i and go to label l, otherwise skip, that is, do not change the contents of register i, and continue with the instruction with label m;
- $(A(i), h, h)$: add 1 to register i and continue the program with the instruction at label h; obviously, the operation $A(i)$ is always possible, hence both labels where to continue have to be the same.

An n-register machine M recognizes the set $N(M)$ of all natural numbers p such that M starts with p in its first register and halts in state f.

Register machines (with two registers) are equivalent to Turing machines, in the sense that they recognize the same family of sets of numbers, namely, the family NRE.

2.3.7 Regulated Rewriting

The context-free grammars are not powerful enough to cover most of the important syntactic constructions from natural and artificial languages, while the context-sensitive grammars are too powerful (for instance, the family

CS has many negative decidability properties, the derivations in a non-context-free grammar cannot be described by a tree, etc.). One way round this difficulty is to restrict the freedom of using the rules of a context-free grammar, and this has motivated the introduction of several dozens of controls/regulations of the derivation in context-free grammars. The oldest one, the matrix grammar, was introduced, in a particular form, in 1965, and it turns out to be very useful for membrane computing. We present it in a gradual (historical) manner, first in its particular and weaker form "without appearance checking", and then in the general form, the one which is equal in power to Chomsky type-0 grammars.

A context-free *matrix grammar* (without appearance checking) is a construct $G = (N, T, S, M)$, where N, T are disjoint alphabets (of non-terminals and terminals, respectively), $S \in N$ (axiom), and M is a finite set of *matrices*, that is sequences of the form $(A_1 \to x_1, \ldots, A_n \to x_n)$, $n \geq 1$, of context-free rules over $N \cup T$. For a string x, a matrix $m : (r_1, \ldots, r_n)$ is executed by applying the productions $r_1, \ldots, r_n$ one after the other, following the order in which they appear in the matrix. Formally, we write $w \Longrightarrow_m z$ if there is a matrix $m : (A_1 \to x_1, \ldots, A_n \to x_n)$ in M and the strings $w_1, w_2, \ldots, w_{n+1}$ in $(N \cup T)^*$ such that $w = w_1, w_{n+1} = z$, and for each $i = 1, 2, \ldots, n$ we have $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$. If the matrix m is understood, then we write $\Longrightarrow$ instead of $\Longrightarrow_m$. As usual, the reflexive and transitive closure of this relation is denoted by $\Longrightarrow^*$. Then, the language generated by G is $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages generated by context-free matrix grammars is denoted by MAT^λ (the superscript indicates that λ-rules are allowed); when only λ-free rules are used, we denote the corresponding family by MAT. (As we will see below, using λ-rules or not makes an essential difference in this area, hence we cannot avoid λ-rules as is the case for context-free grammars.)

Example 2.3.5. The non-context-free language $L_3 = \{a^n b^n c^n \mid n \geq 1\}$ mentioned in Subsection 2.3.4 can be generated by the matrix grammar

$$\begin{aligned} G &= (\{S, A, B, C\}, \{a, b, c\}, S, M), \text{ with the matrices} \\ M &= \{(S \to ABC), \\ &\quad (A \to aA, B \to bB, C \to cC), \\ &\quad (A \to a, B \to b, C \to c)\}. \end{aligned}$$

The reader can easily check this assertion: the use of each matrix increases synchronously the number of occurrences of symbols a, b, c.

It is also easy to construct matrix grammars for the non-context-free languages L_4, L_6 considered in Subsection 2.3.4. This is not so easy for language L_7, hence we give a matrix grammar for it.

Example 2.3.6. Consider the matrix grammar

$$G = (\{S, A, B, C, D, E\}, \{a, b\}, S, M),$$

with M containing the following matrices:

$$\begin{aligned}
m_1 &: (S \to AE),\\
m_2 &: (A \to A, E \to DD),\\
m_3 &: (A \to aB),\\
m_4 &: (B \to B, D \to EE),\\
m_5 &: (B \to aA),\\
m_6 &: (A \to C),\\
m_7 &: (B \to C),\\
m_8 &: (C \to a),\\
m_9 &: (C \to C, E \to b),\\
m_{10} &: (C \to C, D \to b),\\
m_{11} &: (S \to ab).
\end{aligned}$$

Excepting the derivation $S \Longrightarrow ab$ (which uses the matrix m_{11}), all derivations in G begin by using the matrix m_1, and contain two main phases: one which uses the matrices m_8, m_9, m_{10} (in the presence of C), and an initial one which uses the matrices m_2, m_3, m_4, m_5. These phases are separated by a step which uses either m_6 or m_7. In the first phase, the matrices m_2 and m_4 can double the number of occurrences of the symbols D and E. This is possible only in the presence of symbols A and B, respectively. The change from A to B and conversely introduces an occurrence of the terminal a. Since it is not necessary to double all occurrences of D and E by matrices m_2 and m_4, the number of occurrences of D and E is smaller than or equal to 2^n, where n is the number of occurrences of the symbol a. The second phase of the derivation replaces each occurrence of D and E by b. Therefore, the generated string belongs to the language L_7.

Conversely, each string in L_7 can be generated by the grammar G, hence we have the equality $L(G) = L_7$.

We leave to the reader the task to write a grammar for the language L_8, and we only produce a grammar for the language L_{10}. Actually, because L_{10} is the permutation closure of the language L_3, we only indicate a way of constructing a grammar $G' = (N', T, S, M')$ which generates the permutation closure of the language generated by a given matrix grammar $G = (N, T, S, M)$. The idea is rather simple. For each terminal symbol a of G we introduce the symbol a' which is a non-terminal for G'. Then, G' contains all matrices of G, as well as the matrices

$$\begin{aligned}
&(a' \to b', b' \to a'),\ a, b \in T,\\
&(a' \to a),\ a \in T.
\end{aligned}$$

(Clearly, $N' = N \cup \{a' \mid a \in T\}$.) The matrices from $M' - M$ can be used at any time, and they permute the terminal symbols of G in an arbitrary manner, hence producing all permutations of strings in $L(G)$.

In this way we have obtained a proof for the important observation that the families MAT, MAT^λ are closed under permutation.

The following results on matrix languages are known:

1. $CF \subset MAT \subseteq MAT^\lambda \subset RE$.
2. $MAT \subset CS$, $CS - MAT^\lambda \neq \emptyset$.
3. The families MAT, MAT^λ are closed under union, concatenation, intersection with regular languages, λ-free morphisms, right and left derivatives, mirror image, and permutation; MAT^λ is closed under arbitrary morphisms.
4. Each language $L \in MAT^\lambda, L \subseteq a^*$, is regular. (This assertion, together with the closure properties of the family MAT^λ, implies that the above-mentioned languages $L_5, L_{13} - L_{17}$ are not in MAT^λ.)

The last assertion has some interesting consequences:

5. $NMAT^\lambda = NMAT = NCF = NREG$.
6. $D0L - MAT^\lambda \neq \emptyset$, $ND0L - NMAT^\lambda \neq \emptyset$.

The questions whether or not the inclusion $MAT \subseteq MAT^\lambda$ is proper and whether or not MAT^λ contains languages which are not context-sensitive are old *open problems* of this area.

In a matrix grammar as above (without appearance checking), when using a matrix, *all* rules are used, in the order imposed by the matrix. A powerful extension is provided by the possibility to skip certain rules, and this leads to the definition of matrix grammars *with appearance checking.*

Such a grammar is a construct $G = (N, T, S, M, F)$, where (N, T, S, M) is a matrix grammar as above, and F is a set of occurrences of rules in the matrices of M.

For $w, z \in (N \cup T)^*$ we write $w \Longrightarrow z$ if there is a matrix $(A_1 \to x_1, \ldots, A_n \to x_n)$ in M and the 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 (1) $w_i = w_i' A_i w_i'', w_{i+1} = w_i' x_i w_i''$, for some $w_i', w_i'' \in (N \cup T)^*$, or (2) $w_i = w_{i+1}$, A_i does not appear in w_i, and the rule $A_i \to x_i$ appears in F.

Therefore, the difference between a matrix grammar with appearance checking and one without appearance checking is the fact that in the former case we have at our disposal the set F, consisting of *occurrences* of rules in the matrices of M (that is, if the same rule, say $A \to x$, appears several times in the matrices, only some of these occurrences can be present in F; we may interpret them as "marked", having some "flags" which distinguish them from the other rules); the rules of a matrix are applied in order, as usual, possibly skipping the rules in F if they cannot be applied. Thus, if a rule not in F is met, then it has to be used. If a rule from F is met, then

we have two cases: if it *can* be applied, then it *must* be applied; if it cannot be applied (the non-terminal from its left-hand member is not present in the current string), then the rule may be skipped. That is why the rules from F are said to be applied in the *appearance checking* mode. The information given by such an application of a rule from F is rather useful, because in the case of skipping the rule we get "negative information": a certain symbol *is not* present in the string. Example 2.3.7 given below will illustrate both the definition of the derivation in a matrix grammar with appearance checking and the power of using the rules from the set F.

The language generated by G is defined by $L(G) = \{w \in T^* \mid S \Longrightarrow^* w\}$. The family of languages of this form is denoted by MAT_{ac}^{λ} when using λ-rules and by MAT_{ac} when only λ-free rules are allowed.

The following results are known:

1. $MAT \subset MAT_{ac} \subset CS$.
2. $MAT^{\lambda} \subset MAT_{ac}^{\lambda} = RE$.
3. The family MAT_{ac} is closed under union, concatenation, Kleene closure, intersection with regular languages, λ-free morphisms, left and right derivatives, mirror image, and permutation (but not under arbitrary morphisms, because its morphic closure is $MAT_{ac}^{\lambda} = RE$).

We have mentioned above that the language $L_5 = \{a^{2^n} \mid n \geq 1\}$ is not in the family MAT^{λ}. In contrast, this language can be generated by a matrix grammar which uses the appearance checking feature. Actually, we will produce a λ-free matrix grammar with appearance checking for the language $L_5' = \{ba^{2^n} \mid n \geq 1\}$. The initial symbol b can be erased by a left derivative; because the family MAT_{ac} is closed under this operation, we obtain the fact that $L_5 \in MAT_{ac}$, hence a λ-free matrix grammar for L_5 can also be obtained – but it will be a little bit more complex and here we want to have a clearer example.

Example 2.3.7. Consider the grammar

$$G = (\{S, A, B, X, Y, Z, \#\}, \{a, b\}, S, M, F),$$

with the following matrices:

$$\begin{aligned}
&m_1 : (S \to XA),\\
&m_2 : (X \to X, A \to BB),\\
&m_3 : (X \to Y, A \to \#),\\
&m_4 : (Y \to Y, B \to A),\\
&m_5 : (Y \to X, B \to \#),\\
&m_6 : (Y \to Z, B \to \#),\\
&m_7 : (Z \to Z, A \to a),\\
&m_8 : (Z \to b, A \to a).
\end{aligned}$$

The set F consists of all rules having the symbol # on their right-hand side.

Let us examine the work of this grammar. First, let us observe that # is a trap symbol; once introduced, it cannot be removed, hence the sentential form will not lead to a terminal string. That is, the rules from F should be always "applied" by skipping them, hence when the sentential form does not contain the corresponding symbols A, B from the rules in the second positions of matrices m_3, m_5, m_6. Second, one sees that the symbols X, Y, Z are a sort of control symbols, the "real work" is done by the rules from the second place of matrices, under the control of the rules from the first position. Specifically, in the presence of X one doubles the length of the string (passing from a string of the form XA^m to a string of the form XB^{2m}), in the presence of Y one returns to a string of the form XA^p, while in the presence of Z we can pass to a terminal string.

We start by using m_1, hence producing the string XA. Assume that we have a sentential form XA^m, for some $m \geq 1$. As we have mentioned above, if we use m_3 in order to rewrite a string which contains at least one occurrence of A, then the trap symbol # is introduced and the derivation will not lead to a terminal string. Thus, we have to use m_2 as much as possible; that is, until producing the string XB^{2m} (each use of the matrix replaces one occurrence of A by two occurrences of B). To such a string we can apply (only) m_3, and we obtain the string YB^{2m}: X was replaced by Y, but the second rule is applied in the appearance checking mode, skipping it. Observe the important contribution of the rule $A \to \#$: if we do not introduce the symbol #, then we are sure that *all* symbols A were rewritten, the doubling was complete. Now, matrix m_4 should be used as long as at least one occurrence of B is present: if at least one B is present, then the matrices m_5, m_6 will introduce the trap symbol #. When we get the string YA^{2m} we can use either the matrix m_5, or the matrix m_6. In the former case we get the string XA^{2m}, hence the process can be repeated. In the latter case we get the string ZA^{2m}. From now on, only matrices m_7, m_8 can be used. Of course, m_8 should be used when only one occurrence of A is present, otherwise the string can no longer be processed, in spite of the fact that it is a non-terminal one. Thus, we can repeatedly double the number of occurrences of A, and at any time we can turn to a terminal string. This means that the language L_5' is indeed generated.

If matrix m_8 is replaced by $m_8' : (Z \to a, A \to a)$, then we generate the one-letter non-regular language $\{a^{2^n+1} \mid n \geq 1\}$.

The previous grammar has matrices of a rather restricted form. Actually, for each language in MAT_{ac}^{λ} (hence for each recursively enumerable language) one can find a grammar of this form. Specifically, the following very useful result holds.

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 these three sets mutually disjoint, and the matrices in M are in 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 $X \in N_1, A \in N_2$, and $x \in T^*, |x| \leq 2$.

Moreover, there is only one matrix of type 1 and F consists exactly of all rules $A \to \#$ appearing in matrices of type 3; $\#$ is a trap symbol – once introduced, it is never removed. A matrix of type 4 is used only once, in the last step of a derivation.

This normal form can be formulated also for matrix grammars without appearance checking, a case when there is no matrix of type 3 (also, the symbol $\#$ is missing).

Theorem 2.3.8. *For each matrix grammar – with or without appearance checking – there is an equivalent matrix grammar in the binary normal form.*

For an arbitrary matrix grammar $G = (N, T, S, M, F)$, let us denote by $ac(G)$ the cardinality of the set $\{A \in N \mid A \to \alpha \in F\}$.

The following result will be used in many proofs.

Theorem 2.3.9. *For each language $L \in RE$ there is a matrix grammar with appearance checking G, in the binary normal form, such that $L(G) = L$ and $ac(G) \leq 2$.*

Consequently, to the properties of a grammar G in the binary normal form we can add the fact that $ac(G) \leq 2$. One says that this is *the strong binary normal form* for matrix grammars with appearance checking.

It is an *open problem* whether or not the result from Theorem 2.3.9 can be improved. It was conjectured that the answer is negative, namely, that $ac(G) \leq 1$ implies $L(G) \in MAT^{\lambda}$. As we will see later, this question has a direct relevance also for membrane computing.

In many proofs in the subsequent chapters, another version of the binary normal form for matrix grammars will be useful.

We say that a matrix grammar with appearance checking $G = (N, T, S, M, F)$ is in the *Z-binary normal form* if $N = N_1 \cup N_2 \cup \{S, Z, \#\}$, with these three sets mutually disjoint, and the matrices in M are in 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, and, if a sentential form generated by G contains the symbol Z, then it is of the form Zw, for some $w \in T^*$ (that is, the appearance of Z makes sure that, except for Z, all symbols are terminal). As above, $\#$ is a trap symbol, and the (unique) matrix of type 4 is used only once, in the last step of a derivation.

Theorem 2.3.10. *For each language $L \in RE$ there is a matrix grammar with appearance checking G in the Z-binary normal form such that $L = L(G)$.*

Note that in the case of the Z-binary normal form we do not have a bound on the number of non-terminals used in the appearance checking mode, but – and this is rather important in applications – we have a special non-terminal, Z, which appears if and only if the derivation is terminal (modulo the erasing of this special symbol).

Important Convention (Notation). In many proofs in this book we will start from matrix grammars in the (strong) binary normal form or in the Z-binary normal form. In order not to repeat again and again the same notations, we consider the following *standard* presentation (notations and labelling) for such grammars.

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 $m + 1$ matrices in M, injectively labelled 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 $m_i, 1 \leq i \leq k$, will be called *matrices of type 2*, the matrices $m_i, k + 1 \leq i \leq n$, will be called *matrices of type 3*, and the terminal matrices $m_i : (X \to \lambda, A \to x), 1 \leq i \leq k$, will be called *matrices of type 4*.

In the case of grammars in the strong binary normal form we have (at most) two symbols $B^{(1)}, B^{(2)}$ in N_2 such that the rules $B^{(j)} \to \#$ appear in matrices $m_i, k + 1 \leq i \leq n$. We denote $lab_j = \{i \mid$ the matrix $m_i, k + 1 \leq i \leq n$, contains the rule $B^{(j)} \to \#\}$, $j = 1, 2$. For uniformity, we also denote $lab_0 = \{1, 2, \ldots, k\}$ and $lab = lab_0 \cup lab_1 \cup lab_2 = \{1, 2, \ldots, n\}$ (note that $0 \notin lab$). Clearly, the sets lab_0, lab_1, lab_2 are disjoint.

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 $m_{n+1} : (Z \to \lambda)$.

For all matrix grammars in the binary normal form we may assume that all symbols $X \in N_1$ appear as the left-hand side of a rule from a matrix: if this is not the case, then the derivation is blocked after introducing such a symbol, hence we can remove these symbols (and all matrices involving them).

Thus, when we say "a matrix grammar in the (strong) binary normal form, or Z-binary normal form, *in the standard notation/presentation*" we will always mean the notations and the conventions given here.

As we have mentioned, there are many ways to control the derivation of a context-free grammar in order to generate non-context-free languages. Details can be found in [47]. We only recall here some basic definitions, for the types of controls which are of interest for membrane computing.

A context-free *programmed* grammar is a construct $G = (N, T, S, R)$, where N, T, S are as above, the set of non-terminals, the set of terminals, and the start symbol, and R is a finite set of rules of the form $(b : A \to z, E, F)$, where b is a label, $A \to z$ is a context-free rule over $N \cup T$, and E, F are two sets of labels of rules of G. (E is said to be the *success field*, and F is the *failure field* of the rule.) A rule $(b : A \to z, E, F)$ is applied as follows: if the context-free rule $A \to z$ can be used, then it is used and the next rule to be applied is chosen from those with the label in E, otherwise, we choose a rule labelled by some element of F, and try to apply it. This type of programmed grammars is said to be with *appearance checking*; if no failure field is given for any of the rules, then we obtain a programmed grammar without appearance checking. If for all rules we have $E = F$, then the grammar is said to be *with unconditional transfer* (the next rule to be applied is the same, irrespective of whether the current rule can be effectively used or not).

Sometimes it is useful to write a programmed grammar in the form $G = (N, T, S, R, \sigma, \varphi)$, where N, T, S are as above, R is a set of usual context-free rules and σ, φ are mappings from R to the power set of R; $\sigma(p)$ is the success field of the rule p (a rule in $\sigma(p)$ must be used after successfully applying the rule p), and $\varphi(p)$ is the failure field (a rule from $\varphi(p)$ must be considered when p cannot be applied). This way of writing a programmed grammar emphasizes the important feature of these devices: the control is imposed on the next rule to be used, on the sequence of rules, from one step to the next one (not by prescribing blocks of rules, as in a matrix grammar).

The family of languages generated by programmed grammars with arbitrary context-free rules is denoted by PR_{ac}^{λ}; when only λ-free rules are used one removes the superscript λ; when all failure fields are empty, then one removes the subscript ac. One knows that $MAT_{\beta}^{\alpha} = PR_{\beta}^{\alpha}$ for all combinations of α and β (of course, α can be λ or can be missing, while β can be ac or can be missing).

A context-free *ordered* grammar is a system $G = (N, T, S, R, >)$, where N, T, S are as above, R is a finite set of context-free productions, and $>$ is a partial order relation over R. A production p can be applied to a sentential form x only if it can be applied as a context-free rule and there is no production $r \in R$ such that r is applicable and $r > p$ holds.

The family of languages generated by ordered grammars is denoted by ORD when only λ-free rules are used, and by ORD^{λ} when arbitrary context-free rules are allowed. The following relations are known:

1. $ET0L \subset ORD \subset MAT_{ac}$.
2. $ORD^{\lambda} \subset RE$.

Regulated applications of productions can also be based on checking the contents of the string to which a rule is to be used. We present here only the *random context grammars*, which are constructs $G = (N, T, S, R)$, where N, T, S are as above and R is a finite set of triples of the form $p : (A \to w; E, F)$, where $A \to w$ is a context-free production over $N \cup T$ and E, F are subsets of N. Then, p can be applied to a string $x \in (N \cup T)^*$ only if $x = x_1 A x_2$, $E \subseteq alph(x_1 x_2)$, and $F \cap alph(x_1 x_2) = \emptyset$. If E or F is the empty set, then no condition is imposed by E or F, respectively. E is said to be the set of *permitting* and F is said to be the set of *forbidding* context conditions of p.

The elements of E, F can also be strings, and then all the strings from E should appear in the sentential form to be rewritten and no string from F may appear in this sentential form. Such grammars (called *semi-conditional*) generate all recursively enumerable or all context-sensitive languages, depending on whether λ-rules are used or not, respectively.

For random context grammars, we have the following results. Denote by $RC^{\lambda}_{p,f}$ the family of languages generated by random context grammars with erasing rules; the subscript f is removed when all rules $(A \to z, E, F)$ have the set F empty, and p is removed when all rules $(A \to z, E, F)$ have the set E empty; when only λ-free rules are used we remove the superscript λ. The following results are known:

1. $CF \subset RC_p \subseteq MAT \subset RC_{p,f} = MAT_{ac}$.
2. $RC_f = ORD$, $RC^{\lambda}_f = ORD^{\lambda} \subset RE$.
3. $RC_p \subseteq RC^{\lambda}_p \subseteq MAT^{\lambda} \subset RC^{\lambda}_{p,f} = RE$.

Another very fruitful idea for increasing the power of context-free grammars (in certain cases, also of regular grammars) is to consider distributed generative devices: constructs composed of several grammars working together according to a well-specified cooperation protocol. This leads to the idea of a *grammar system*. Two main classes of grammar systems have been investigated, the sequential ones (known under the name of *cooperating distributed grammar systems*) and the *parallel communicating grammar systems*. We briefly introduce the first class here, because it will soon be useful; details – including bibliographical references – can be found in [44] and [219].

A *cooperating distributed* (in short, CD) grammar system of degree $n \geq 1$ is a construct

$$\Gamma = (N, T, S, R_1, R_2, \ldots, R_n),$$

where N, T are disjoint alphabets (the non-terminal and the terminal alphabet, respectively), $S \in N$ (the axiom), and $R_1, \ldots, R_n$ are finite sets of context-free rules over $N \cup T$. For two sentential forms w, w' over $N \cup T$, we write $w \Longrightarrow_i^* w'$, $w \Longrightarrow_i^{=k} w'$, $w \Longrightarrow_i^{\leq k} w'$, $w \Longrightarrow_i^{\geq k} w'$, $w \Longrightarrow_i^{t} w'$, for some $k \geq 1$, if w' can be obtained from w (1) by any number of derivation steps, (2) by k derivation steps, (3) by at most k derivation steps, (4) by at least k derivation steps, (5) by a maximal number of derivation steps,

using rules from R_i. (In the case of the t mode, no further derivation step $w' \Longrightarrow w''$ is possible by means of rules from R_i.) The language $L_\alpha(\Gamma)$, generated by Γ in the mode $\alpha \in \{*, t\} \cup \{\leq k, = k, \geq k \mid k \geq 1\}$, consists of all strings $x \in T^*$ such that $S \Longrightarrow_{j_1}^{\alpha} w_1 \Longrightarrow_{j_2}^{\alpha} \ldots \Longrightarrow_{j_m}^{\alpha} w_m = x$, for some $m \geq 1, 1 \leq j_s \leq n, 1 \leq s \leq m$. (That is, the "components" $R_1, \ldots, R_n$ of the system work in turn, according to the cooperation protocol α, on a common sentential form. When a terminal string is obtained it is said to be generated by the system.)

We denote by $CD_n(\alpha)$ the family of languages generated by CD grammar systems of degree at most $n \geq 1$, in the mode α; when the degree is not bounded the subscript n is replaced by $*$. We recall here only a few results on the power of CD grammar systems:

1. $CF = CD_*(*) = CD_1(\alpha) = CD_*(= 1) = CD_*(\leq k) = CD_*(\geq 1)$, for all $k \geq 1, \alpha \in \{t\} \cup \{= j, \geq j \mid j \geq 1\}$.
2. $CF = CD_1(t) = CD_2(t) \subset CD_3(t) = ET0L$.
3. $CF \subset CD_*(\alpha) \subseteq MAT^\lambda$, for all $\alpha \in \{= k, \geq k \mid k \geq 1\}$.

2.3.8 On the Difference Between CS and RE

At first sight, there is a "big difference" between the family of context-sensitive languages and that of recursively enumerable languages, between the power of grammars which use erasing rules and λ-free grammars. However, if we closely examine this relationship, then we find that this difference is not so large. For instance, the following result is valid:

Theorem 2.3.11. *For every language $L \subseteq T^*, L \in RE$, there are $L' \in CS$ and $c_1, c_2 \notin T$, such that $L' \subseteq L\{c_1\}\{c_2\}^*$, and for each $w \in L$ there is $i \geq 0$ such that $wc_1c_2^i \in L'$. (Thus, L is equal to L' modulo a tail of the form $c_1c_2^i$ for some $i \geq 0$.)*

Corollary 2.3.2. *Each recursively enumerable language is the projection of a context-sensitive language.*

Of course, the assertion from Theorem 2.3.11 is valid also in a "mirrored" version, with $L' \subseteq \{c_2\}^*\{c_1\}L$.

These results prove that the families RE and CS are "almost equal", the difference lying in a tail of arbitrary length to be added to the strings of a language; because this tail is of the form $c_1c_2^i, i \geq 1$, it carries no other information than its length, hence from a syntactical point of view the two languages L and L' from Theorem 2.3.11 can be considered indistinguishable.

Intuitively speaking, in order to obtain a device of the power of type-0 grammars (of Turing machines, computationally complete), it is sufficient to have (1) context-sensitivity and (2) erasing. Of course, this is a very general formulation. For instance, we have to be careful what context-sensitivity

means. As we will see when presenting the splicing operation, it is not sufficient to "sense the neighbourhood" of the site where we process a string, but we need "enough context-sensitivity to send signals at an arbitrarily long distance". In its turn, the erasing capability should be sufficient in order to be able to use an arbitrarily large workspace and to delete the useless symbols at the end of a computation.

Admittedly, these considerations are rather vague, but the reader will better understand them when looking for characterizations of recursive enumerability in any given framework.

2.3.9 Universal Turing Machines and Type-0 Grammars

A computer is a *programmable* machine, able to execute any program it receives. From a theoretical point of view, this corresponds to the notion of a *universal Turing machine*, and, in general, to the notion of a machine which is universal for a given class, in the following sense.

Consider an alphabet T and a Turing machine $M = (K, V, T, \flat, s_0, F, \delta)$. As we have seen above, M starts working with a string w written on its tape and reaches or does not reach a final state (and then halts), depending on whether or not $w \in L(M)$. A Turing machine can be codified as a string of symbols over a suitable alphabet. Let us denote such a string by $code(M)$. Imagine a Turing machine M_u which starts its work from a string which contains both $w \in T^*$ and $code(M)$ for a given Turing machine M, and stops in a final state if and only if $w \in L(M)$.

Such a machine M_u is called *universal*. It can simulate any given Turing machine, providing that a code of the particular machine is written on the tape of the universal one, together with a string to be dealt with by the particular machine.

The parallelism with a computer, as we know computers in their general form, is clear: the code of a Turing machine is its *program*, the strings to be recognized represent the input data, and the universal Turing machine is the computer itself, with the instructions of the universal Turing machine corresponding to the operating system of a computer.

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

The idea of a universal Turing machine was introduced by Turing himself, who also produced such a machine. Many universal Turing machines are now available in the literature, in general, looking for machines with a small number of states and/or of transitions (instructions).

Given a Turing machine M we can effectively construct a type-0 grammar G such that $L(G) = L(M)$ as follows. Consider a Turing machine $M = (K, V, T, \flat, s_0, F, \delta)$ and construct a non-restricted Chomsky grammar G working as follows: starting from its axiom, G non-deterministically generates a string w over V, then it makes a copy of w (of course, the two copies of w are separated by a suitable marker; further markers, scanners, and other auxiliary symbols are allowed, because they can be erased when they are no longer necessary). On one of the copies of w, G can simulate the work of M, choosing non-deterministically a computation as defined by δ; if a final state is reached, then the witness copy of w is preserved, and everything else is erased.

Details of such a construction can be found in [221], as well as in the prerequisites chapter of [195].

Applying this construction to a universal Turing machine M_u, we obtain a *universal type-0 Chomsky grammar* G_u, a grammar which is universal in the following sense: the language generated by G_u consists of strings of the form, say, $w\#code(M)$, such that $w \in L(M)$. (We can call the language $\{w\#code(M) \mid w \in L(M)\}$ itself universal, and thus any grammar generating this language is universal.) However, we are interested in a "more grammatical" notion of universality, and this leads to the following definition.

A triple $G = (N, T, R)$, where the components N, T, R are as in a usual Chomsky grammar, is called a *grammar scheme*. For a string $w \in (N \cup T)^*$ we define the language $L(G, w) = \{x \in T^* \mid w \Longrightarrow^* x\}$, the derivation being performed according to the productions from R.

A *universal type-0 grammar* is a grammar scheme $G_u = (N_u, T_u, R_u)$, where N_u, T_u are disjoint alphabets, and R_u is a finite set of rewriting rules over $N_u \cup T_u$, with the property that for any type-0 grammar $G = (N, T_u, S, R)$ there is a string $w(G)$ such that $L(G_u, w(G)) = L(G)$.

Therefore, the universal grammar simulates any given grammar, providing that a code $w(G)$ of the given grammar is taken as a starting string of the universal one.

There are universal type-0 grammars in the sense specified above. A concrete construction can be found in Chapter 3 of [195], following [34].

A universal grammar G_u can be constructed by encoding in the "type-0 grammars' programming language" the very way of using a grammar in a derivation process: choose a rule, remove an occurrence of its left-hand member, introduce instead an occurrence of its right-hand member, and check whether or not a terminal string is obtained.

For the membrane computing area it is important to note that when a characterization of recursively enumerable languages/numbers is obtained in

a constructive manner (with the construction started from Turing machines or from Chomsky type-0 grammars), the universality is transferred "for free" from Turing machines/type-0 grammars to membrane systems (the same assertion is true for any type of computing devices). Thus, computational completeness directly means universality, hence programmability. (There is a question here about what the code of a membrane system means and how it is introduced in a universal system; the answer is always provided by the construction from the proof of the completeness result.)

Because in the membrane computing area computational completeness always implies universality, we will say "computational universality", or simply "universality", when having a result which, rigorously speaking, is stated as a computational completeness result.

The universality has an important consequence for membrane computing in what concerns hierarchies of families of languages or of sets of numbers with respect to the number of membranes used by the systems which generate them: all these hierarchies collapse in the case of classes of membrane systems which are computationally complete, because all the proofs of such results start from a given grammar and produce a membrane system equivalent with it; if the grammar is universal, hence fixed, then the obtained system Π_u is fixed, hence it has a precise number of membranes, and also has universality properties. Given another system, Π, it can be simulated by the universal one, after introducing a code of the particular system Π into the universal system Π_u. When the code does not change the number of membranes – and this will always be the case, because the axiom of the grammar simulated by Π_u corresponds to the contents of a membrane of Π_u – then the number of membranes used by Π_u suffices for characterizing the Turing computability. This reasoning gives no precise bound on the number of these membranes, it just guarantees its existence. For precise classes of membrane systems, we will almost always find very low values for the number of membranes which entail universality.

Another important observation is that there is no natural definition of universality, hence no natural universality result, for levels of the Chomsky hierarchy other than the type-0 grammars. This implies the fact that when we have a new computing machinery, in order to have a programmable device, we need a machinery which is computationally complete (hence containing "enough" context-sensitivity and "sufficient" erasing possibilities).

2.3.10 Splicing, Insertion–Deletion, Context Adjoining

Rewriting is only one (fundamental and well-known) operation by which we can process strings (in particular, in the membrane computing area). In formal language theory there are many other string operations, some of them inspired by the very area where membrane computing has its origins, biology.

This is the case with the *splicing* operation, introduced in [86] as a formal counterpart of the recombination of DNA molecules under the influence of

restriction enzymes. Comprehensive details can be found in [195], hence we recall here only a few ideas.

Consider an alphabet V and two symbols $\natural, \$$ not in V. A *splicing rule* over V is a string $r = u_1 \natural u_2 \$ u_3 \natural u_4$, where $u_1, u_2, u_3, u_4 \in V^*$. For such a rule r and for $x, y, w, z \in V^*$ we define

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

(One cuts the strings x, y in between u_1, u_2 and u_3, u_4, respectively, and one recombines the fragments obtained in this way.)

Based on this operation, one can define a language generating device as follows.

First, the operation is extended to languages. An *H scheme* is a pair $\sigma = (V, R)$, where V is an aphabet and $R \subseteq V^* \natural V^* \$ V^* \natural V^*$ is a set of splicing rules. For an H scheme σ and a language $L \subseteq V^*$ we define

$$\begin{aligned}\sigma(L) = \{z \in V^* \mid & (x, y) \vdash_r (w, z) \text{ or } (x, y) \vdash_r (z, w), \\ & \text{for some } x, y \in L, \text{ and } r \in R\}.\end{aligned}$$

Then, the operation is iterated, passing from $\sigma(L)$ to the closure of L under the splicing with respect to the rules from R.

$$\begin{aligned}\sigma^0(L) &= L, \\ \sigma^{i+1}(L) &= \sigma^i(L) \cup \sigma(\sigma^i(L)), \text{ for } i \geq 0, \\ \sigma^*(L) &= \bigcup_{i \geq 0} \sigma^i(L).\end{aligned}$$

An *extended H system* is a construct $\gamma = (V, T, A, R)$, where V is an alphabet, $T \subseteq V$ (terminal alphabet), $A \subseteq V^*$ (axioms), and $R \subseteq V^* \natural V^* \$ V^* \natural V^*$ is a set of splicing rules over V. The language generated by γ is defined by $L(\gamma) = \sigma^*(A) \cap T^*$, where $\sigma = (V, R)$ is the underlying H scheme associated with γ.

For two families FL_1, FL_2 of languages, we denote by $EH(FL_1, FL_2)$ the family of languages $L(\gamma)$ generated by H systems $\gamma = (V, T, A, R)$ with $A \in FL_2$ and $R \in FL_2$ (note that the splicing rules are written as strings, hence it makes sense to speak about the type of the "language" R in a given hierarchy of languages).

The following are basic results in the splicing area:

1. $EH(REG, FIN) = REG$, $EH(CF, FIN) = CF$.
2. $EH(FIN, REG) = RE$.

Therefore, by using a finite set of splicing rules, when starting from a regular language we get only regular languages. This is a good illustration

of the discussion about the context-sensitivity and the erasing needed for obtaining the power of Turing machines: splicing is context-sensitive, because we cut the strings in a given context; it also has erasing, but this is not sufficient. The explanation lies in the fact that by cutting (and recombining fragments of) strings we lose information about the fragments, so that we cannot "send signals at long distance".

However, there are more than a dozen controlled H systems, with rather weak regulations about the use of the splicing rules, which can characterize RE; the feeling is that one needs rather weak additional context-sensitivity features in order to lead H systems to computational universality (and this will be the case also when using the splicing in the membrane computing area).

Other operations with strings investigated in formal language theory mainly with a motivation coming from linguistics are those dealing with inserting or adjoining strings, or pairs of strings, to given strings; dually, single strings or couples of strings can be deleted from given strings.

For instance, an insertion rule is of the form (u, x, v), with the meaning that a string $w = w_1uvw_2$ is transformed by such a rule to $w' = w_1uxvw_2$ (the string x was inserted in the context (u, v)). A rule of the same form can be used in the deletion manner: from $z = z_1uxvz_2$ we pass to $z' = z_1uvz_2$ (the substring x was deleted from the context (u, v)). Now, an *insertion–deletion system* can be defined as a construct $\gamma = (V, T, A, I, D)$, where V is an alphabet, $T \subseteq V$ (the terminal alphabet), $A \subseteq V^*$ (a finite set of axioms), and I, D are finite sets of insertion and deletion rules, respectively. The generated language consists of all strings over T^* which can be obtained by starting from the strings from A and using iteratively rules from I and from D. Such devices can generate all recursively enumerable languages.

A variant of interest is that of *Marcus contextual grammars*, where one deals with pairs of strings, called *contexts*, which are inserted in or deleted from given strings. We only present here the so-called *internal contextual grammars with selection*, which are constructs of the form $G = (V, A, R)$, where V is an alphabet, A is a finite set of strings over V (axioms), and R is a finite set of pairs of the form $(C, (u, v))$, with $C \subseteq V^*$ and $(u, v) \in V^* \times V^*$; C is called the *selector* and (u, v) is called the *context* of the *rule* $(C, (u, v))$.

For two strings $w, z \in V^*$ we write $w \Longrightarrow z$ if $w = w_1xw_2, z = w_1uxvw_2$, for some $(C, (u, v)) \in R$, with $x \in C$, and $w_1, w_2 \in V^*$ (the context (u, v) is adjoined to the string x from the selector C associated with the context). The language generated by G is defined by

$$L(G) = \{z \in V^* \mid w \Longrightarrow^* z \text{ for some } w \in A\}.$$

Details about insertion–deletion systems and about Marcus contextual grammars (of various forms: for instance, in the *external* variant, the contexts are always adjoined at the ends of the processed string) can be found in many places, in particular in [183] and [219].

2.3.11 Elements of Complexity

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

Actually, in order to define a complexity class we need to specify several parameters. First, we have to choose a given *model of computation*, a way to define "algorithms". The notion of an algorithm is a general, undefined one: a precise recipe, which can be followed mechanically, and which, in a finite time, can solve a problem. In order to precisely deal with algorithms we need to choose a mathematical definition for what we understand by this notion, and the standard one is the Turing machine (the multi-tape version, with the input tape being a read-only tape, is the one customarily chosen in complexity theory, but this is not essential for our needs). Second, for defining a complexity class we need to fix a *mode of computation.* Turing machines can be deterministic or non-deterministic, hence we have at least these two modes. The Turing machine is a single-device "computer", executing one instruction in each time unit; it is a *sequential* computing model. There are several possibilities to imagine *parallel* computing devices, containing several "processors" which work simultaneously (cooperating at each time unit or only at given times, sharing or not sharing a common memory, etc.), executing several instructions at the same time. Third, we have to specify the *resource* we want to deal with; *time* is perhaps the most important, but the *space* used for computing is also important. One can imagine several others, related to the "effort" of computing and on the effort of programming the computing device.

Having specified these elements, a complexity class is defined with respect to functions $f : \mathbf{N} \longrightarrow \mathbf{N}$ (with certain minimal properties, for instance non-decreasing, computable, etc.) in the following way: the complexity class $\mathbf{md\text{-}res}_M(f)$ consists of all problems which can be solved in mode *md* by devices D from class M such that for any input x, device D needs at most $f(|x|)$ units from the resource *res*.

Turing machines are the standard models of computing used in complexity theory not only because they are mathematically elegant and historically important, but also because they are robust from the complexity classes' points of view. Roughly speaking, this means that the complexity classes remain the same (for "good" functions f) if we define them by taking the class M of devices to be the class of deterministic Turing machines, and then

changing the type of the Turing machine (with several tapes, for instance) or even the model of computation (but not the mode of computing).

The most important complexity classes are **DTIME**(f), **DSPACE**(f), which contain the problems which can be solved in a deterministic mode in a time, respectively, a space bounded by the function f, and **NTIME**(f), **NSPACE**(f), which contain the problems which can be solved in a non-deterministic mode in a time, respectively, a space bounded by the function f.

The difference between deterministic and non-deterministic algorithms (solutions to problems) is crucial. The standard problem in complexity is a decidability problem, a question with two possible answers, "yes" and "no", about a given "piece of reality". Here is an example: given a graph G, does it contain a Hamiltonian path (a path which passes through all vertices exactly once)? A deterministic algorithm (Turing machine) for solving this problem will take an instance of the problem (a graph) and after a while will provide one of the two possible outputs, "yes" or "no", telling us whether or not the graph contains a Hamiltonian path. The size of the input is described by the number n of the nodes of G; if the time needed to solve the problem is bounded by $f(n)$, then we say that the problem belongs to **DTIME**(f). In the non-deterministic case, we say that a problem is in **NTIME**(f) if, for every input of size n (any graph with n nodes, in the case of the Hamiltonian path problem), there is at least one computation of the algorithm (of the non-deterministic Turing machine, for instance) which gives us the answer "yes" after $f(n)$ time units. Note the important detail that it is sufficient for at least one computation to exist which gives us the positive answer; it does not matter how many computations fail to give this answer (or simply fail to stop). If no computation exists which gives the answer "yes", then we conclude that the problem has the answer "no". Therefore, the conclusion "no" is obtained at the limit, knowing that no computation provides the answer "yes". The difference from the deterministic case is obvious.

A very illustrative reformulation of the difference between deterministic and non-deterministic complexity classes is the following one. In the deterministic case one has to explore all branches of the computability tree, giving the unique answer one can get after completing this exploration, "yes" or "no". In the non-deterministic case one has only to guess a successful root-to-leaf branch, hence to explore a single branch of the computability tree. In the first case the time of computing is proportional to the number of nodes in the whole computability tree; in the second case the time is proportional to the depth of the tree, the length of the longest branch.

Obviously, deterministic classes are included in the non-deterministic classes.

Now, the important step is to choose the mapping f. The most widely investigated complexity classes start from the constant, logarithmic (sometimes, polylogarithmic), linear, polynomial, and exponential functions. In general, it is considered that the borderline between *tractable* and *intractable*

is that between *polynomial* and *exponential*. That is why polynomial functions are very important – also mentioning that "constants and lower monomials do not count", a polynomial of degree k is identified in this framework with n^k. Then, the union of all classes $\mathbf{DTIME}(n^k)$, $k \geq 1$, is usually denoted by $\mathbf{P}$, while the union of all classes $\mathbf{NTIME}(n^k), k \geq 1$, is usually denoted by $\mathbf{NP}$. These are the classes of problems which can be solved in a deterministic polynomial time, respectively of problems which can be solved in a non-deterministic polynomial time. The inclusion $\mathbf{P} \subseteq \mathbf{NP}$ is clear; whether or not this is an equality is probably the best known and the most important open problem in computer science today, the so-called $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem.

The problems from the class $\mathbf{P}$ are considered computationally tractable, that is one can afford to solve them in a feasible time, while the problems from $\mathbf{NP}$ (which are not known to be in $\mathbf{P}$) are considered intractable. The best-known deterministic algorithms for problems in $\mathbf{NP} - \mathbf{P}$ are of an exponential time complexity, and the exponentials grow "too fast" in order to wait an exponential time for a solution. On the other hand, a computer should be a deterministic machine, otherwise we will never know how long we have to wait for a non-deterministic algorithm to determine that a problem has the answer "no": either the computer has not yet correctly guessed a way towards the positive answer, or such an answer does not exist at all.

Thus, it is important to know whether a given problem lies in the class $\mathbf{P}$ or not. However, is not enough to know that a problem is not in the class $\mathbf{P}$. Is the problem inherently difficult, hence in $\mathbf{NP} - \mathbf{P}$ (providing that, as widely expected, $\mathbf{P} \neq \mathbf{NP}$), or are we not yet able to find a deterministic polynomial solution to it? Answering this question for a particular problem would mean solving the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, but there are easier ways to assess the complexity of a problem and conclude that it is "almost surely" intractable. Namely, the problems can be *reduced* from one problem to another one. We say that problem B reduces to problem A if there is a transformation R which, for any input x of B, produces an input $R(x)$ to A such that the answer of A to the input $R(x)$ is the answer of B to the input x. Thus, if B reduces to A, in order to solve B it is enough to compute $R(x)$ and to solve A for the input $R(x)$. Of course, a reduction is acceptable if it is less complex than the problem B itself – with respect to classes $\mathbf{P}$ and $\mathbf{NP}$, a reduction R is acceptable if it can be done in polynomial time. In such a case, when B reduces to A, we may say that A is at least as complex as B, hence it is important to know whether a problem A has the property that "many" (all) problems from a given class can be reduced to A. In such a case we may say that A is among the hardest problems of that class. Nicely enough, for most natural complexity classes, $\mathbf{P}$ and $\mathbf{NP}$ included, there are problems such that *all* problems from that class can be reduced to them! Such problems are called *complete* for the respective classes. Of particular interest are the problems which are $\mathbf{NP}$-complete. All problems for which we do not have (yet) a deterministic polynomial algorithm can be reduced to

them, hence the **NP**-complete problems are "the most difficult from **NP**". This is considered a sufficient guarantee that such a problem is not tractable.

There are lots of **NP**-complete problems, many of them with an innocent formulation, and many of them of direct practical interest (in some sense, the converse is also true: most practical problems are intractable). For instance, the above-mentioned problem about the existence of a Hamiltonian path in a given graph is **NP**-complete. In Chapter 7 we will deal with several others, but we will introduce them at that time.

Classic electronic computers are "von Neumann computers", executing one instruction at a time, inspired from the Turing machine (from the Turing universality theorem). Complexity classes **P** and **NP** are defined starting from sequential models of computing (Turing machines), and we have mentioned that many real-life problems cannot be efficiently solved in this framework. (Well, this is only a partially valid assertion, omitting many details. In many circumstances we can be satisfied with approximate solutions; there are random, probabilistic, evolutionary algorithms, sometimes working unexpectedly well. On the other hand, knowing that a problem is in **P** does not automatically have a practical importance. In theory we only consider the degree of the polynomial function f, the coefficients "do not matter", which is not at all true for a real computation. Furthermore, the theory deals with the worst-case complexity, while for real problems the average-case complexity might be more relevant, etc.) So, an important challenge is to define computing models – and then to implement them – able to overcome this difficulty. The classic hope in this respect is *parallel computing*, with its modern "incarnation", molecular computing.

In some sense, the parallelism tries to simulate the non-determinism: if we had enough processors, then we could explore all branches of a computation tree at the same time, and thus we could get the answer "yes" or "no" – hence the answer provided by a deterministic computation – after exhausting the tree, in a time equal to the depth of the tree, that is, in the time needed by a non-deterministic computation.

Rather attractive, but "if we had enough processors". This is a very delicate point. How many processors, how is their work organized, how are they obtained? The processors themselves should be "computed" (their instructions and programs should be written, starting from a given instance of a problem) in a reasonable time, the processors spend time in coordinating their work, for communication – implicitly, their number should be "reasonable".

The widely used computing model for defining parallel computing complexity classes are the Boolean circuits – roughly speaking, families of ascending nets with Boolean gates in the nodes, starting from given truth values and computing a complex logical function, in parallel; each circuit in the family starts from an input of a given size, hence such a family is of the form $\mathcal{C} = (C_1, C_2, \ldots)$, such that C_i has i inputs. In order to accept such a family

as a good parallel "computer", we have first to ensure that the circuits C_i are constructed in a *uniform manner* (a Turing machine should exist, able to construct C_i starting from 1^i in polynomial time – starting from 1^i is meant to ensure that the circuits indeed form a family, they are related to the same problem, and the difference from C^i and C^j lies in the size of the input they can handle). The complexity classes are then defined by taking into account both the time necessary for computing a function with n inputs and the number of gates from C_n. The time is equal to the depth of the circuit. Thus, one compares the depth of circuits C_n with their number of nodes.

Practically tractable for parallel computing (by Boolean circuits) are considered the functions that can be computed in a polylogarithmic time, $\log^k n$, by using a polynonial number of gates, n^k. One says that the time is *parallel time*, while the number of gates is called the (amount of) *work* (power), which leads to the notation $\mathbf{NC} = \bigcup_k \mathbf{PT/WK}(\log^k n, n^k)$. This class has nice properties, similar to those of classes **P** and **NP**: it is stable under changing the computing model (for instance, from Boolean circuits to PRAMs – parallel random access machines), it is closed under reductions, but many important problems about it are still open. Perhaps the most relevant is the question whether or not the inclusion $\mathbf{NC} \subseteq \mathbf{P}$ is proper or not. (The inclusion is clear, as we deal with a polynomial number of processors working a polynomial number of steps.) Otherwise stated: are all sequentially polynomial problems efficiently parallelizable (solvable in polylogarithmic time on a polynomial number of processors), or are there inherently sequential problems? As in the case of the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem, the conjecture widely accepted is that the answer is negative, and the most probable candidate problems from **P** which are not in **NC** are the **P**-complete problems. There are many such problems, so it seems that Boolean circuits, PRAMs, and other related parallel models of computing are not able to handle them in an efficient manner.

As basic references in the computational complexity area we mention the excellent book [174], as well as [73] (for a large collection of **NP**-complete problems), and the corresponding chapters from [219] and [239].

2.3.12 Multisets

Membrane computing essentially deals with multisets, which are sets with multiplicities associated with their elements.

Let U be an arbitrary set. A *multiset* (over U) is a mapping $M : U \longrightarrow \mathbf{N}$; $M(a)$, for $a \in U$, is the *multiplicity of a in the multiset M*. We indicate this fact also in the form $(a, M(a))$. Of course, in this set-up the multiplicity of each object with respect to any multiset is finite. If the set U is finite, $U = \{a_1, \ldots, a_n\}$, then the multiset M can be explicitly given in the form $\{(a_1, M(a_1)), \ldots, (a_n, M(a_n))\}$. The *support* of a multiset M is the set $supp(M) = \{a \in U \mid M(a) > 0\}$. A multiset M is empty when its support is empty (it is then denoted by $\emptyset$).

Let $M_1, M_2 : U \longrightarrow \mathbf{N}$ be two multisets. We say that M_1 is included in M_2, and we write $M_1 \subseteq M_2$, if $M_1(a) \leq M_2(a)$, for all $a \in U$. The inclusion is strict, and we write $M_1 \subset M_2$, if $M_1 \subseteq M_2$ and $M_1 \neq M_2$. The union of M_1 and M_2 is the multiset $M_1 \cup M_2 : U \longrightarrow \mathbf{N}$ defined by $(M_1 \cup M_2)(a) = M_1(a) + M_2(a)$, for all $a \in U$. The difference $M_1 - M_2$ is defined here only when M_2 is included in M_1 and it is the multiset $M_1 - M_2 : U \longrightarrow \mathbf{N}$ given by $(M_1 - M_2)(a) = M_1(a) - M_2(a)$, for all $a \in U$. If $n \in \mathbf{N}$ and $M : U \longrightarrow \mathbf{N}$ is a multiset, then the amplification of M by n (the scalar product by n) is the multiset $n \otimes M : U \longrightarrow \mathbf{N}$ defined by $(n \otimes M)(a) = n \cdot M(a)$, for all $a \in U$.

For $Y \subseteq X$ and M a multiset over X, we define the *projection* of M on Y by

$$pr_Y(M)(a) = \begin{cases} M(a), & \text{if } a \in Y, \\ 0, & \text{otherwise,} \end{cases}$$

for each $a \in X$.

A multiset M of finite support, $\{(a_1, M(a_1)), \ldots, (a_n, M(a_n))\}$, can also be represented by a string: $w = a_1^{M(a_1)} a_2^{M(a_2)} \ldots a_n^{M(a_n)}$ and all permutations of this string precisely identify the objects in the support of M and their multiplicities. Otherwise stated, the Parikh image of w, $\Psi_U(w)$, is exactly the vector $(M(a_1), \ldots, M(a_n))$ of multiplicities. In the subsequent chapters we will frequently use this more compact representation of multisets of a finite support, thus speaking about "the multiset w", where w is a string (and meaning "the multiset represented by w"), or writing $w(a)$ for $|w|_a$, or even concatenating multisets (for w_1, w_2 strings representing two multisets, $w_1 w_2$ will represent the multiset obtained by the union of the two multisets).

Note the differences (and the similarities) between sets (no multiplicity, no order), multisets (multiplicity but no order), and strings (multiplicity and linear order).

Further details about the mathematics and the applications of multisets in various branches of computer science can be found in [37]; sometimes a multiset is called a *bag*, see, for example, [241].

2.4 Bibliographical Notes

We have already mentioned several bibliographical sources, but we still want to add some specific details. Closure properties for ordered languages can be found in [55]. The fact that one-letter languages from MAT^λ are regular was proved in [85] – after remaining for many years as a conjecture/open problem. Lindenmayer systems were introduced in [125]. Insertion grammars were first considered in [72], and Marcus contextual grammars (with the contexts adjoined in the external mode) are introduced in [143].

A more detailed introduction to formal language theory, also containing other notions and results which might be of interest for membrane computing, can be found in [154].

3. Membrane Systems with Symbol–Objects

We proceed now to introduce the basic model of a membrane system (P system), the one with the membranes arranged in a hierarchical structure, as in a cell, and processing multisets of symbol–objects. We discuss first the simplest model, and the definition will be only partially formalized, in order to introduce the main ideas in a friendly manner. We then prove that this class of systems is not powerful enough, an observation which makes necessary the introduction of further features (such as a priority relation among rules and the membrane dissolving action). This is the first significant class of membrane systems, and we investigate it in several details. First, we illustrate the definition with several examples, then, in Section 3.3, we investigate the power of these basic extensions. A complete formalization is given in Section 3.5. Several further extensions of the model, with additional features, are discussed and investigated in Section 3.6.

3.1 A Simple Class

The main ingredients of a membrane system are the *membrane structure*, the *multisets* of *objects*, and the *evolution rules*.

A membrane structure should be understood as a three-dimensional arrangement of vesicles, such that the size and the spatial placement (distances, positions with respect to any coordinates) do not matter; what is important is the mutual relationship between membranes, the relations of adjacency, of being in or out. Always there is a unique external membrane, the *skin*. The reader is asked to look again at Figure 1.2, where several notions are illustrated: *elementary membrane* (a membrane without any membrane inside), adjacent membranes, *region* enclosed by a membrane (this is clear for elementary membranes, and is the space between a membrane and the immediately inner membranes in the case of non-elementary membranes), *label* of a membrane (and of the corresponding region), *environment*.

It is important to note the one-to-one correspondence between membranes and the regions they enclose. Because of this precise correspondence, we may speak about membranes and understand regions, about their labels, about the contents of membranes as being the contents of their regions, etc.

Note also that from now on the notion of a membrane is idealized, thus departing from the biological membrane: it is just the frontier between an "inside" and an "outside", without a material consistence and without any internal structure.

A membrane structure can be pictorially represented by a Venn diagram, like that in Figure 1.2 (thus, although we have started from a three-dimensional intuition, we can safely work with a two-dimensional representation).

Since a membrane structure corresponds to a directed unordered tree (the root is the skin membrane, the nodes are the membranes, the relation "to be directly inside" defines the edges), it can be mathematically represented by a tree. For instance, the tree in Figure 3.1 corresponds to the membrane structure from Figure 1.2.

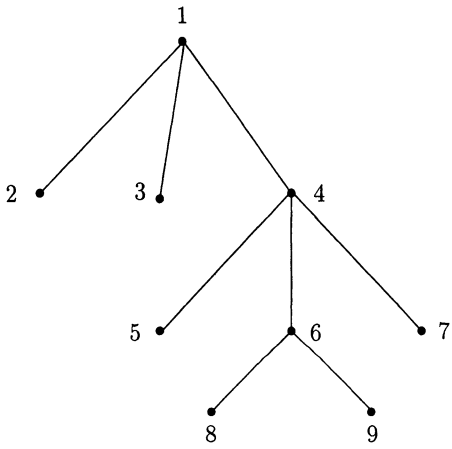


Fig. 3.1. The tree representing the membrane structure from Figure 1.2

The representation by a tree loses the biochemical intuition, but it makes clear the fact that the positions of membranes do not matter, but so do only their containment relationships.

A membrane structure can also be represented by a string of matching parentheses. For instance, the membrane structure from Figure 1.2 is represented by the following parenthetic expression:

$$[_1[_2\ ]_2[_3\ ]_3[_4[_5\ ]_5[_6[_8\ ]_8[_9\ ]_9]_6[_7\ ]_7]_4]_1.$$

Because the membranes have labels, the pairs of corresponding parentheses also have labels. Of course, the same membrane structure may be represented by different parenthetic expressions (the order of parentheses corresponding to neighboring membranes placed in the same upper membrane does not matter).

The second important ingredients are the multisets of objects, which are represented by strings of symbols associated with the objects, in the way discussed in Subsection 2.3.12. In turn, the evolution rules for objects will be given as multiset rewriting rules, of the form $u \to v$, where u and v are strings (understood as representations of multisets). It is important to note that, working with multisets, we work with multiple *copies* of objects; when saying, for instance, that "an object a is processed" (for example, passed through a membrane), we understand "a copy of the object a is processed".

With these minimal clarifications, and with the intuition the reader should already have from the previous sections (from the Introduction and from the biological elements recalled in Section 2.1) we can now introduce the main object of our investigation, the *symbol–object membrane systems* (also called *P systems* or, shortly, *systems* when no danger of confusion arises).

Such machinery is a construct

$$\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o),$$

where:

(i) O is an alphabet – its elements are called *objects*;
(ii) μ is a membrane structure consisting of m membranes, with the membranes (and hence the regions) injectively labelled with $1, 2, \ldots, m$; m is called the *degree* of Π;
(iii) $w_i, 1 \leq i \leq m$, are strings which represent multisets over O associated with the regions $1, 2, \ldots, m$ of μ;
(iv) $R_i, 1 \leq i \leq m$, 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 $u \to v$, where u is a string over O and v is a string over O_{tar}, where $O_{tar} = O \times TAR$, for $TAR = \{here, out\} \cup \{in_j \mid 1 \leq j \leq m\}$;
(v) $i_o \in \{1, 2, \ldots, m\}$ is the label of an elementary membrane (the *output membrane*).

The symbols *here, out*, $in_j, 1 \leq j \leq m$, are called *target commands* or *target indications*, and the length of u is called *the radius* of the rule $u \to v$. To simplify the notation, the target indication "here" for letters (objects) in evolution rules will be mostly omitted.

If Π contains rules of radius greater than one, then we say that Π is a system *with cooperation*. Otherwise, it is a *non-cooperative* system. A particular class of cooperative systems is that of *catalytic* systems. In such a system, a subset $K \subseteq O$ is given (its elements are called catalysts), hence the system is written in the form $\Pi = (O, K, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$. In a catalytic system we can have either rules $a \to v$, with $a \in O, v \in (O - K)_{tar}$, or rules (of a radius greater than one) of the form $ca \to cv$, where $c \in K, a \in O - K$, and v is a string over $(O - K)_{tar}$, where $(O - K)_{tar} = (O - K) \times TAR$. Note that no other evolution rules contain catalysts (that is, there are no rules of the form $c \to v$ or $a \to v_1 c v_2$, with $c \in K$ and $a \in O - K$, $v, v_1, v_2 \in O^*_{tar}$).

The rules from the set $R_i, 1 \leq i \leq m$, are applied to objects from the region i of μ *synchronously, in a non-deterministic maximally parallel manner.* The meaning of this phrase is explained below.

We assume that a global clock exists, marking the time for the whole system (for all compartments of the system). The m-tuple of multisets of objects present at any moment in the m regions of Π constitutes the *configuration* of the system at that moment. The m-tuple $(w_1, \ldots, w_m)$ is the *initial configuration* of Π.

For two configurations $C_1 = (w'_1, \ldots, w'_m)$, $C_2 = (w''_1, \ldots, w''_m)$ of Π we write $C_1 \Longrightarrow C_2$, and we say that we have a *transition* from C_1 to C_2, if we can pass from C_1 to C_2 by using the evolution rules from $R_1, \ldots, R_m$ in the regions $i, \ldots, m$, in the following manner.

Consider a rule $u \to v$ from a set R_i. To apply this rule in region i means to remove the multiset of objects specified by u from region i, and to introduce the objects specified by v, *in the regions indicated by the target commands associated with the objects from* v. Specifically, if v contains a pair $(a, here)$, then the object a will be placed in the same region i where the rule is applied. If v contains a pair (a, out), then the object a will be moved to the region immediately outside membrane i – this region can be the environment in the case when i is the skin membrane; in this case, the object leaves the system and it never comes back. Finally, if a pair (a, in_j) appears in v, then the object a should be moved in the membrane with label j, providing that this membrane is immediately inside membrane i (that is, region i is adjacent to membrane j); if membrane j is not directly accessible from region i, inside region i, then the rule $u \to v$ cannot be applied.

Therefore, by using the rule $u \to v$ in region i we consume the objects specified by the multiset u (we subtract this multiset from the multiset of region i), and we add the objects specified by v to the multiset from region i and to the neighbouring multisets, according to the target commands.

The catalysts do not change the region where they are placed in the initial configuration of the system.

The objects to evolve in a step and the rules by which they evolve are chosen in a non-deterministic manner, but in such a way that in each region we have a maximally parallel application of rules. This means that we assign objects to rules, non-deterministically choosing the rules and the objects assigned to each rule, but in such a way that after this assignation no further rule can be applied to the remaining objects. A rule can be applied in the same step as many times as we want, only the number of copies of objects matters. The objects which remain unassigned are left where they are, and they are passed unchanged to the next configuration.

For example, assume that in a region we have the objects $aaabb$ and the rules $r_1 : aa \to cc$ and $r_2 : ab \to dd$. Then, we can use either the couple r_1, r_2, or twice the rule r_2. In the former case we get the multiset $bccdd$, in the latter case we get the multiset $adddd$. In both cases, one copy of an object remains

unused, first one copy of b, then one copy of a, but no rule can be applied to it; the assignation of objects to rules is maximal. It is not correct to use r_1 only once or r_2 only once, as this is not a maximal application of rules.

A transition from a configuration C_1 of Π to another configuration is obtained by simultaneously evolving all the multisets from C_1. This means that at the same time, all sets of rules $R_i, 1 \leq i \leq m$, are used in the corresponding regions.

It is instructive to see a single step transforming a configuration of the system as a "macro-step" consisting of several "micro-steps" performed after each other. Consider a region i of the system Π. First, we assign (occurrences of) objects from region i to rules from R_i, non-deterministically choosing rules and objects until no further assignment is possible (note that the multiplicity of objects present in region i is crucial in this micro-step). Then, all these "assigned" objects are removed from the current multiset of objects in region i, and (occurrences of) all objects specified by the right hand sides of the chosen rules are added to this multiset, together with their target commands: in_j, *out, here.* Now, all transfers indicated by the commands in_j and *out* are executed, and copies of objects with the transfer command *here* remain in region i. Finally, the target commands are removed, and a macro-step is completed for region i. Since all regions are processed "simultaneously" (with all micro-steps performed synchronously), this completes the global macro-step.

The macro-steps corresponding to the use of evolution rules are performed in parallel, for all possible applicable rules $u \to v$, for all occurrences of multisets u in the regions associated with the rules, and for all regions, following the principles of non-determinism and maximal parallelism.

We emphasize the fact that we have here a double parallelism, once at the level of each region (the rules are used in parallel) and once at the level of the system (all regions evolve concomitantly).

A sequence of transitions between configurations of a given system Π is called a *computation* with respect to Π. A computation is *successful* if and only if it halts: there is no rule applicable to the objects present in the last configuration. The result (output) of a successful computation is the number of objects present in the output membrane in the halting configuration of Π. Note that the objects which are not in the output membrane are not counted. Also, it is important to note that a computation which is not halting yields no output. The set of numbers computed in this way by the system Π is denoted by $N(\Pi)$; (bearing in mind the similarity with the grammars) we also say that the set $N(\Pi)$ is *generated* by Π.

3.2 Two Examples

We now illustrate the definitions from the previous section with two examples.

The first one is rather symple. Consider the (cooperating) system of degree 2

$$\Pi_1 = (O, \mu, w_1, w_2, R_1, R_2, i_o),$$

with the following components:

$$\begin{aligned}
O &= \{a, b, c\},\\
\mu &= [_1[_2\ ]_2]_1,\\
w_1 &= a^2,\\
w_2 &= \lambda,\\
R_1 &= \{a \to a(b, in_2)(c, in_2)^2,\ a^2 \to (a, out)^2\},\\
R_2 &= \emptyset,\\
i_o &= 2.
\end{aligned}$$

The initial configuration of the system is represented in Figure 3.2, where also the rules of R_1 are given.

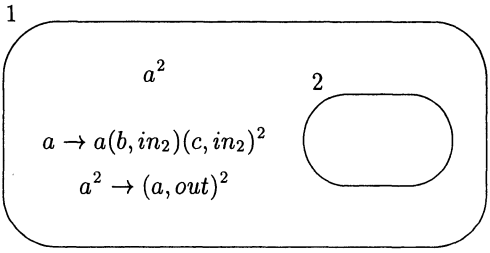


Fig. 3.2. The initial configuration of Π_1

We start with two copies of the object a in membrane 1. In any subsequent configuration – except the halting one – we will also have two copies of a in membrane 1. If one of these copies is processed by the rule $a \to a(b, in_2)(c, in_2)^2$, then the other one also should be processed by the same rule (because of the maximal parallelism). Using this rule means reproducing the object a (in region 1, as in the right-hand side of the rule we have $(a, here)$, with the indication "here" omitted), and sending a copy of b and two copies of c to membrane 2, the output one. No rule is available in membrane 2. The process continues until using the rule $a^2 \to (a, out)^2$, which sends both copies of a outside the system, and thus the computation halts. After $n + 1$ steps, for some $n \geq 0$ (one step is that when the object a is sent out), in the output membrane we have the objects $b^{2n}c^{4n}$, therefore, we have

$$N(\Pi_1) = \{6n \mid n \geq 0\}.$$

The same result is obtained for the simpler, non-cooperating system

$$\Pi_1' = (\{a,b\}, [_1\ ]_1, a, \{a \to ab^6, a \to (a, out)\}, 1).$$

(This time we have given the system in a compact way, directly indicating all its components.)

The system Π_1 suggests an interesting technical point: we can distinguish among the objects b, c and consider as the result of a halting computation the vector of multiplicities of both b and c in membrane 2. In this way, Π_1 generates all pairs $(2n, 4n)$, for $n \geq 0$.

In general, for any symbol–object membrane system Π, instead of the set of numbers $N(\Pi)$ we can consider as the result of computations the set $Ps(\Pi)$, of all vectors of numbers representing the multiplicities of objects present in the output membrane in the halting configuration ("Ps" comes from "Parikh set", as the result of a computation is the Parikh image of the string representing the multiset of objects from membrane i_o). In this book we choose the slightly simpler case of sets $N(\Pi)$, but most of the results remain true also for sets $Ps(\Pi)$.

Let us now consider a little more complex system. Let

$$\Pi_2 = (\{A, B, D, E, X, Y, Z, a, \#\}, [_1[_2\ ]_2]_1, XADE, \lambda, R_1, \emptyset, 2),$$

with the set R_1 containing the following rules (no rule is associated with the output membrane):

$$\begin{aligned}
&XADE \to XBBDE,\\
&XE \to YE,\\
&AD \to \#,\\
&\# \to \#,\\
&YBDE \to YADE,\\
&YD \to XD,\\
&BE \to \#,\\
&YD \to Z,\\
&ZA \to Z(a, in_2).
\end{aligned}$$

The reader should compare this system with the matrix grammar with appearance checking in Example 2.3.7, Subsection 2.3.7, as the work of the two devices is similar: in the presence of the object X, each copy of A is doubled, replaced by two copies of B; in the presence of the object Y, each copy of B returns to A; the passing from X to Y is possible only when no copy of A is present, while the passing from Y to X (or to Z, at the end of a computation) is possible only if no copy of B is present. This "appearance checking" operation is made possible by the maximally parallel manner of using the rules, and it is controlled by the objects D, E and by the trap object $\#$: if any copy of $\#$ is produced, then the system will evolve forever

by means of the rule $\# \to \#$ present in the skin region, hence the computation will never be halting, and it will give no output. At the end of a correctly proceeding computation (that is, not producing the trap object), each A is replaced by a, which is sent to the output membrane.

Here are some details about the work of Π_2. We start with the objects $XADE$ in the skin region. Assume that we are in a configuration when we have here a multiset of the form XA^nDE, for some $n \geq 1$; initially, $n = 1$. If the rule $AD \to \#$ is used, then the computation will never halt, hence we have to "keep busy" the object D by using the rule $XADE \to XBBDE$. Assume that we use this rule for a while, hence we have a multiset XA^mB^pDE in the skin membrane (clearly, $p = 2(n - m)$). If any of the rules $AD \to \#$ and $BE \to \#$ is used, then the trap object is produced, hence we have to use again the rule $XADE \to XBBDE$. That is, we have to use this rule as much as possible, until exhausing the copies of A and producing the multiset $XB^{2n}DE$.

Again, the use of the rule $BE \to \#$ introduces the trap object, but we can "keep busy" the object E by using the rule $XE \to YE$. In a way similar to the doubling of the number of copies of A, we now proceed to replace each copy of B by A, in the presence of Y. Only when this operation is complete can we replace Y with X (by the rule $YD \to XD$, hence "keeping busy" the object D), so that the process can be iterated.

When we have a multiset YA^mDE we can also use the rule $YD \to Z$, which will entail the end of the computation: from now on, the only applicable rule is $ZA \to Z(a, in_2)$, which will replace each copy of A by the "terminal object" a, which will be sent to the output membrane. When all copies of A are used, no rule can be applied, and the computation stops. (The objects Z, E remain in the skin region.)

Consequently, we have $N(\Pi_2) = \{2^n \mid n \geq 1\}$.

3.3 The Power of the Simple Class

We briefly investigate here the generative capacity of the class of systems considered above, mainly in order to prove that they are not powerful enough, hence further ingredients should be considered.

Let us denote by $NOP_m(\alpha, tar)$ the family of sets of natural numbers of the form $N(\Pi)$, generated by symbol–object P systems of degree at most $m \geq 1$, using rules of type α, where $\alpha = coo$ indicates the use of cooperating rules, $\alpha = ncoo$ means that all rules of Π are non-cooperating, and $\alpha = cat$ means that non-cooperating rules or rules with catalysts are used. When the degree of the system is not bounded, the subscript m is replaced with $*$. The symbol *tar* refers to the fact that the communication of objects through membranes uses the addressing in_j, which indicates the *target* membrane (later we will also use other types of addressing, so we need a notation for

this feature). When $m = 1$, hence the only possible target commands are *here* and *out*, we write $NOP_1(\alpha)$ instead of $NOP_1(\alpha, tar)$.

The following relations follow directly from the definitions.

Lemma 3.3.1. (i) $NOP_m(\alpha, tar) \subseteq NOP_{m+1}(\alpha, tar)$, *for all* $m \geq 1$ *and all* $\alpha \in \{ncoo, cat, coo\}$.

(ii) $NOP_m(ncoo, tar) \subseteq NOP_m(cat, tar) \subseteq NOP_m(coo, tar)$, *for all* $m \geq 1$, *and* $NOP_*(ncoo, tar) \subseteq NOP_*(cat, tar) \subseteq NOP_*(coo, tar)$.

By a straightforward construction, or by invoking the Turing–Church thesis, we also have the next inclusion.

Lemma 3.3.2. $NOP_*(coo, tar) \subseteq NRE$.

The previous "proof" of Lemma 3.3.2 is not very honest: a "straightforward construction" here means to construct a Turing machine or a type-0 Chomsky grammar simulating a given system Π, and how to do this is clear in principle, but by no means an easy task, as the construction can be rather long. On the other hand, the Turing–Church thesis is just a conjecture, and strictly mathematically speaking, it proves nothing.

Actually, from now on we will not use such arguments, simply because we will no longer mention any further "straightforward" inclusion like that from Lemma 3.3.2 – while in Section 8.1 we will provide the full details of a construction proving an inclusion into RE (there, not as a goal *per se*, but with other purposes).

We now give a general result, which says that all hierarchies $NOP_m(\alpha, tar)$, $m \geq 1$, collapse at the second level.

Theorem 3.3.1. $NOP_*(\alpha, tar) = NOP_m(\alpha, tar)$, *for all* $\alpha \in \{ncoo, cat, coo\}$ *and* $m \geq 2$.

Proof. Consider a system $\Pi = (O, K, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$ of some degree $m \geq 1$, with the skin membrane labelled with 1. Assume that $m \geq 2$; because i_o should be elementary, it is not the skin membrane, that is $i_o \neq 1$. Let $a_i, 1 \leq i \leq m$, be new symbols associated with each $a \in O$, and h_i be the morphisms defined by $h_i(a) = a_i$, for $a \in O, 1 \leq i \leq m$. Assume that $R_i = \{r_{i,1}, \ldots, r_{i,t_i}\}$, with $t_i \geq 0, 1 \leq i \leq m$. Also, assume that Π does not contain rules which cannot be applied because of wrong target indications: if R_i contains a rule $u \to v$ with v introducing (b, in_j), then j is a membrane placed immediately inside i.

We construct the system

$$\begin{aligned}
\Pi' &= (O', K', [_1[_{i_o}\ ]_{i_o}]_1, w, w_{i_o}, R'_1, R'_{i_o}, i_o),\\
O' &= O \cup \{a_i \mid a \in V, 1 \leq i \leq m\},\\
K' &= \{c_i \mid c \in K, 1 \leq i \leq m\},\\
w &= h_1(w_1) \ldots h_{i_o-1}(w_{i_o-1}) w_{i_o}\ h_{i_o+1}(w_{i_o+1}) \ldots h_m(w_m),
\end{aligned}$$

$R'_1 = \{r'_{i,j} : h_i(u) \to v' \mid$ for $r_{i,j} : u \to v \in R_i, 1 \leq i \leq m, i \neq i_o,$
$1 \leq j \leq t_i,$ where v' is obtained from v in the following way:
each $(b, here)$ from v is replaced with b_i; each (b, out)
from v is replaced with b_j, where j is the membrane outside i,
in the membrane structure μ,
with the exception of $i = 1$, where (b, out) is left unchanged;
each (b, in_s) from v is replaced with b_s, for s a membrane
placed immediately inside i,
with the exception of $s = i_o$, where (b, in_{i_o}) is left unchanged$\}$,
$R'_{i_o} = \{r'_{i_o,j} : u \to v' \mid$ for $r_{i_o,j} : u \to v \in R_{i_o}, 1 \leq j \leq t_{i_o},$ where
v' is obtained from v in the following way:
each $(b, here)$ from v remains unchanged;
each (b, out) from v is replaced with (b_j, out),
where j is the membrane outside i_o in the system $\Pi\}$.

The itinerary of objects through membranes of Π is simulated by the subscripts of symbols used in Π'. The membrane i_o is preserved, with the symbols representing its objects not having subscripts. Thus, the rules in R_{i_o} have the same left-hand multiset (but a modified right-hand side). The rules of Π' corresponding to the rules of Π act "locally", on the objects associated with each membrane of Π, because of the subscripts of the symbols. In this way, any computation in Π can be simulated by an equivalent computation in Π', and conversely; the two computations stop in the same circumstances. In consequence, $N(\Pi) = N(\Pi')$ and we have the inclusion $NOP_m(\alpha, tar) \subseteq NOP_2(\alpha, tar)$ for all $m \geq 2$ and all $\alpha \in \{ncoo, cat, coo\}$ (the form of rules is not changed and if $K = \emptyset$, then also $K' = \emptyset$). The converse inclusion was pointed out in Lemma 3.3.1. □

In the case of non-cooperating systems, the hierarchy on the number of membranes collapses at the first level. Moreover, such systems can generate only the length sets of context-free languages, hence the family of semilinear sets of numbers.

Theorem 3.3.2. $NOP_*(ncoo, tar) = NOP_1(ncoo) = NCF$.

Proof. The inclusion $NOP_1(ncoo) \subseteq NOP_*(ncoo, tar)$ is obvious, while the inclusion $NCF \subseteq NOP_1(ncoo)$ can be easily proved: given a context-free grammar $G = (N, T, S, R)$, in the reduced form (hence all non-terminals can be rewritten into a terminal string), we construct the system $\Pi = (N \cup T, [_1\ ]_1, S, R \cup \{A \to A \mid A \in N\}, 1)$, which is clearly length-equivalent with G (because of rules $A \to A, A \in N$, and because G is reduced, only terminal derivations in G correspond to halting computations in Π; moreover, because of rules $A \to A$, the sequential rewriting in G can be simulated in Π by

computations which are also sequential, in the sense that in each transition one effectively processes only one copy of only one non-terminal symbol).

From Theorem 3.3.1 we know that $NOP_*(ncoo, tar) \subseteq NOP_2(ncoo, tar)$, hence in order to complete the proof it is sufficient to prove the inclusion $NOP_2(ncoo, tar) \subseteq NCF$. To this aim we prove the inclusion $NOP_2(ncoo, tar) \subseteq NCD_2(t)$ and we make use of the equality $CD_2(t) = CF$ pointed out in Subsection 2.3.7.

Consider a non-cooperative P system of degree 2, $\Pi = (O, [_1[_2\ ]_2]_1, w_1, w_2, R_1, R_2, 2)$. Denote $F_i = \{a \in O \mid \text{there is no rule } a \to x \in R_i\}$, $i = 1, 2$. For each object $a \in O$ we consider two new symbols, a_i and a'_i, for each $i = 1, 2$. We denote by h_i the morphisms defined by $h_i(a) = a'_i$, for all $a \in O, i = 1, 2$. If a symbol $a \in F_1$ appears in w_1, then it is removed (it cannot evolve and cannot reach the output membrane).

Define the morphism g_1 on $O \times \{here, out, in_2\}$ by

$$g_1((b, here)) = \begin{cases} \lambda, & \text{if } b \in F_1, \\ b'_1, & \text{otherwise,} \end{cases}$$
$$g_1((b, out)) = \lambda,$$
$$g_1((b, in_2)) = \begin{cases} b, & \text{if } b \in F_2, \\ b'_2, & \text{otherwise,} \end{cases}$$

as well as the morphism g_2 on $O \times \{here, out\}$ by

$$g_2((b, here)) = \begin{cases} b, & \text{if } b \in F_2, \\ b'_2, & \text{otherwise,} \end{cases}$$
$$g_2((b, out)) = \begin{cases} \lambda, & \text{if } b \in F_1, \\ b'_1, & \text{otherwise.} \end{cases}$$

We construct the CD grammar system of degree 2, $\Gamma = (N, O - F_2, S, R'_1, R'_2)$, where

$$N = \{a_i, a'_i \mid a \in O, i = 1, 2\} \cup (O - F_2) \cup \{S\},$$

and with the following components:

$$\begin{aligned} R'_1 &= \{S \to h_1(w_1)h_2(w_2)\} \\ &\quad \cup \{a_i \to g_i(x) \mid a \to x \in R_i, i = 1, 2\}, \\ R'_2 &= \{a'_i \to a_i \mid a \in O, i = 1, 2\}. \end{aligned}$$

The component R'_1 simulates the transitions in the system Π. All symbols a_i such that a rule in R_i can be applied to a have to be rewritten, otherwise the derivation step is not done in the t mode as specific to CD grammar systems. Because the use of a rule introduces primed symbols, we cannot repeatedly rewrite a symbol (this is possible in CD grammar systems, but it is not allowed in a transition in a P system). The subscripts of the symbols indicate the membranes where they are placed in the same way as in the proof

of Theorem 3.3.1 and they select the corresponding rules to be applied to the symbols; the subscripts are changed during the derivation according to the evolution rules of Π. All symbols which cannot evolve or are sent outside of the skin membrane are removed, with the exception of the symbols reaching the output membrane, which are terminals with respect to Γ. The component R_2' replaces primed symbols with non-primed symbols, but preserves their subscripts. Thus, the process can be iterated. The only way to reach a terminal string is to remove all symbols different from those from $V - F_2$ which reach the output membrane. Consequently, $N(\Pi) = lenght(L(\Gamma))$. □

Therefore, the power of symbol–object membrane systems without cooperating rules is rather reduced. As expected, this is not true for the cooperating case. We know already from the second example considered in the previous section that $NOP_2(coo, tar)$ contains sets of numbers that are not in $NMAT^\lambda$: we have pointed out in Subsection 2.3.7 that if $L \in MAT^\lambda, L \subseteq a^*$, then $lenght(L)$ is a semilinear set; however, the system Π_2 from Section 3.2 generates the non-semilinear set $N(\Pi_2) = \{2^n \mid n \geq 1\}$.

Actually, in the cooperative case we can generate all recursively enumerable sets of numbers even by using systems with only one membrane.

Theorem 3.3.3. *$NOP_*(coo, tar) = NOP_m(coo, tar) = NRE$, for all $m \geq 1$.*

Proof. We only have to prove the inclusion $NRE \subseteq NOP_1(coo)$. Clearly, each set from NRE is the length set of a one-letter language from RE, hence a language which can be generated by a matrix grammar with appearance checking $G = (N, \{a\}, S, M, F)$, with only one terminal symbol. Take such a grammar in the Z-normal form, in the standard notation, as agreed in Subsection 2.3.7. We construct the system of degree 1, $\Pi = (O, [_1\ ]_1, HX_{init}A_{init}, R_1, 1)$, with the alphabet of objects

$$O = N \cup \{H, H', H''\} \cup \{H_A \mid A \in N_2\},$$

and with the set R_1 containing the following rules:

(a) $HXA \to HYx$, for all matrices $(X \to Y, A \to x) \in M, x \neq \#$,
(b) $HX \to H'H_AY$,
$H_AA \to \#$,
$\# \to \#$,
$H' \to H''$,
$H''H_A \to H$, for all matrices $(X \to Y, A \to \#) \in M$,
(c) $HZ \to \lambda$.

The matrices of M without rules to be used in the appearance checking mode are simulated by the rules of type (a) of Π. The matrices $(X \to Y, A \to \#)$ are simulated as follows: first, one simulates the rule $X \to Y$ and one introduces the objects H' and H_A (by the rule $HX \to H'H_AY$). If any copy of A is available, then at the next step the rule $H_AA \to \#$ is used, and

the computation will never stop because of the rule $\# \to \#$ which can be used forever. If no copy of A is present, then the object H_A remains unused for one step; in the meantime, H' is replaced with H'', and now the rule $H''H_A \to H$ can be used; it removes the object H_A and reintroduces the object H, hence we can simulate another matrix from M. When the object Z is introduced, hence the derivation in the grammar G is terminal, we remove the last objects different from a by using the rule $HZ \to \lambda$. Therefore, by halting computations in Π we can simulate exactly the terminal derivations in G, hence $N(\Pi) = length(L(G))$, and this concludes the proof. □

Note that a non-context-free grammar $G = (N, T, S, R)$ cannot be simulated by a membrane system Π constructed in the direct manner as that used for context-free grammars, that is taking the rules $u \to v$ from R and introducing them in Π, because the rules of G deal with strings, and hence the ordering of symbols in u and v is crucial, while in a multiset we have no ordering of symbols.

The cases of non-cooperative and of cooperative systems are settled; the intermediate one, of systems using only non-cooperative and catalytic rules, is still *open* (Problem Q1). The difficulty lies in the fact that the catalysts bring context-sensitivity, of a rather restricted type, but still we see no way to extend the proof of Theorem 3.3.2 so that the CD grammar system can also simulate the work of catalysts. Actually, we *conjecture* that $NOP_*(cat, tar)$ contains non-semilinear sets of numbers, hence it is not equal to NCF – but probably it is not equal to NRE either.

Although the biochemistry can be considered as being cooperative (the chemical reactions often involve two or more input compounds), from a mathematical point of view it is attractive (and challenging) to use ingredients which are as restrictive as possible, in particular to use non-cooperative rules. Membrane systems with non-cooperative rules are rather weak from a computational point of view – see Theorem 3.3.2. At the level of context-free grammars we do not have natural universality results, hence a membrane system with non-cooperative rules cannot be universal, and hence programmable. Moreover, the result from Theorem 3.3.2 is "bad" also from another point of view: it says that the hierarchy on the number of membranes collapses at the first level. One membrane means no compartmentalization, just a separation from the environment (which is not even necessary when simulating a context-free grammar), and no compartments means no communication.

We have plenty of reasons to consider extensions of the simple systems discussed up to now. We will deal with such extensions in all the remaining sections of this chapter.

3.4 Basic Extensions

We have mentioned that there are possibly several extensions/modifications of the basic definition of membrane systems with symbol–objects, either with biological motivations or with mathematical motivations – or with both. We now consider the basic extensions; further possibilities will be investigated in Section 3.6.

3.4.1 Dissolving a Membrane

The first extension we introduce is the *membrane dissolving* action. We denote it by the symbol δ which may be added to the rules from a system. That is, the rules can be of the form $u \to v$, as used in the previous sections, or of the form $u \to v\delta$, where $u \to v$ is a usual rule (that is, δ is appended to the multiset from the right-hand side of the rule). The application of a rule $u \to v\delta$ in a region i means to use $u \to v$ in the usual way, then to dissolve membrane i. After dissolving a membrane, all objects and membranes previously present in it become elements of the contents of the immediately upper membrane, while the rules of the dissolved membrane are removed. (The intuition is that the rules describe the biochemistry possible in the region delimited from above by the membrane, and, after breaking this region, the reaction conditions change, they are specific to the membrane covering the dissolved membrane.) The skin membrane is never dissolved; that is, no rule of the form $u \to v\delta$ is applicable in the skin membrane.

There is a subtle point here. We can simply forbid that rules of the form $u \to v\delta$ exist in R_1 – and this is the choice we made in this section. However, we can allow such rules to exist in R_1, and then we have a further choice: to simply forbid their use, that is to ignore them (this is the same as forbidding them to appear in R_1), or to block the computation if we reach a configuration where the only rule which can be applied in the skin region is of the form $u \to v\delta$. This latter variant can be a nice "programming tool", useful for controlling the "correct" development of a computation.

Now, because of the dissolving action, both the definition of a membrane system and the definition of a transition between two configurations of a system need some changes. First, the output membrane is no longer necessary to be elementary in the initial configuration of the system, hence a system (of degree m) is given in the form $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$ (possibly, a set K of catalysts is also specified), where all components are as in the previous sections, excepting the fact that the rules from the sets R_i can also be of the form $u \to v\delta$, and that i_o is not necessarily an elementary membrane of μ. Then, also the very notion of a configuration changes, because by the dissolving action the membrane structure can be modified during the computation, as certain membranes can disappear. That is why a configuration is a $(k+1)$-tuple $C = (\mu', w'_{i_1}, \ldots, w'_{i_k})$, with μ' a membrane structure obtained by removing from μ all membranes different from $i_1, \ldots, i_k$ (of course, the

skin membrane is not removed), with w'_{i_j} strings over O, $1 \leq j \leq k$, and $\{i_1, \ldots, i_k\} \subseteq \{1, 2, \ldots, m\}$. Note that not every configuration may be *reachable* through an evolution of the system. Also, note that if a membrane is present in two different configurations, then it will have the same label, because labels are associated with membranes (and never manipulated during the evolution of the system).

For two configurations $C_1 = (\mu', w'_{i_1}, \ldots, w'_{i_k})$, $C_2 = (\mu'', w''_{j_1}, \ldots, w''_{j_l})$ of Π we write $C_1 \Longrightarrow C_2$, and we say that we have a *transition* from C_1 to C_2, if we can pass from C_1 to C_2 by using the evolution rules from $R_{i_1}, \ldots, R_{i_k}$ in the regions $i_1, \ldots, i_k$, as usual, in the non-deterministic maximally parallel manner.

In terms of the micro-steps which compose a macro-step (a transition), as discussed in Section 3.1, in order to take into account the dissolving action δ we add as the last step of a macro-step the following micro-step: for each region where a rule containing δ was used, the membrane enclosing this region is removed, and consequently the objects of this region will now belong to the region that was enclosing the dissolved membrane. Obviously, if the membrane of this region was also dissolved, then the objects "travel" even further up. Since the skin membrane is never dissolved, there is a limit to this travel.

As usual, we consider as successful only halting computations, but we have now to add a further condition: that the output membrane is an elementary one in the halting configuration. Thus, a halting computation where the output membrane is not elementary provides no output.

We denote by $NOP_m(\alpha, tar, \delta)$ the family of sets of numbers $N(\Pi)$ computed in this way by P systems of degree at most $m \geq 1$, with rules of type $\alpha \in \{ncoo, cat, coo\}$, and using the dissolving action. If no rule contains the symbol δ, that is the dissolving action is not used, then we remove δ from $NOP_m(\alpha, tar, \delta)$ and we get the known families $NOP_m(\alpha, tar)$. When the degree of systems is not bounded, the subscript m is replaced by $*$.

We do not give an example at this moment, because a system essentially using the membrane dissolution will be discussed in the proof of Theorem 3.4.1, and then more complex systems will be examined after introducing a further extension of the model.

Conventions. In general, the family of all sets of numbers computed by systems from a given class of P systems will be denoted by NOP (from "numbers", "symbol–objects", and "P systems"), with a subscript indicating a bound on the number of membranes, followed by a pair of parentheses indicating the type of rules used (*ncoo, cat, coo*, or other types if this is the case), and the type of communication control (up to now, we have mentioned only *tar*); in between these parentheses we will also specify other features which are used, but *only in the case when they are used.* Otherwise stated, *what is not indicated in the parentheses is not used by the systems which characterize the respective family.*

However, when we want to specify *the type* of a given system, we will sometimes use expressions of the form "the system Π is of type $(ncoo,\ tar, \neg\delta)$"; that is, we will also specify features which are not used, negated, together with those which are used, in order to stress a fact that is important in the context (in the example, the fact that the membrane dissolving is not used).

Moreover, from now on we will not state the obvious inclusion relations which follow directly from the definitions, that is relations of the following forms (see also Lemma 3.3.1): $m+1$ membranes are at least as powerful as m membranes (for example, $NOP_m(\alpha, tar, \delta) \subseteq NOP_{m+1}(\alpha, tar, \delta)$); using a feature is at least as powerful as not using it (for example, $NOP_m(\alpha, tar) \subseteq NOP_m(\alpha, tar, \delta)$); cooperative rules are at least as powerful as catalytic rules, which are at least as powerful as non-cooperative rules (for example, $NOP_m(ncoo, tar, \delta) \subseteq NOP_m(cat, tar, \delta) \subseteq NOP_m(coo, tar, \delta)$, and the same for the case when δ is missing).

Similarly, we will not point out the inclusions in the family NRE.

However, we will make implicit use of such obvious relations, and, when giving a result (for instance, a universality theorem), we state it for the most restrictive case for which we know that it is true. For instance, if a result holds for m membranes, then we do not state it also for a larger number of membranes or for arbitrarily many membranes (for the subscript $*$); similarly, if a result is true, say for the case when the dissolving action is not used, then we do not formulate it also for the case when action δ is used.

The use of the membrane dissolving action strictly increases the power of symbol–objects membrane systems.

Theorem 3.4.1. $NE0L \subseteq NOP_2(ncoo, tar, \delta)$.

Proof. Having an E0L system $G = (V, T, w, R)$, we construct the membrane system of degree 2

$$\Pi = (V \cup \{d\}, [_1[_2\ ]_2]_1, \emptyset, dw, R_1, R_2, 1),$$

with the following sets of rules:

$$R_1 = \{a \to a \mid a \in V - T\},$$
$$R_2 = \{d \to d,\ d \to \delta\} \cup R.$$

In the inner membrane one simulates derivation steps in G as long as the symbol d is present; when the membrane is dissolved, the simulation stops. If any symbol not in T is still present, then we can use the corresponding rule $a \to a$ in R_1 forever. Thus, only the terminal derivations in G are simulated by halting computations in Π. □

The result above cannot be improved by taking only one membrane: one membrane means no dissolving action; according to Theorem 3.3.2, we can then compute only context-free length sets.

It is also worth mentioning here the importance of the fact that the output membrane is not necessarily an elementary one in the initial configuration, but it must be elementary in the halting configuration, as is the case for the system Π from the previous proof.

It is not possible to generate all length sets of E0L languages by systems with only two membranes without using this trick of considering the skin membrane as the output membrane. This is again a consequence of Theorem 3.3.2: consider a P system with two membranes such that the output one is the inner membrane; because neither the output membrane nor the skin membrane can be dissolved, the system cannot use the membrane dissolving action at all. Consequently, it generates only length sets of context-free languages.

Theorem 3.3.2 and Theorem 3.4.1 imply the following result.

Corollary 3.4.1. $NOP_*(ncoo, tar) \subset NOP_2(ncoo, tar, \delta)$.

However, the power of P systems which use the membrane dissolving action (and non-cooperating rules) is not too large; they can generate at most the length sets of ET0L languages.

Theorem 3.4.2. $NOP_*(ncoo, tar, \delta) \subseteq NET0L$.

Proof. Let us consider a P system of degree $m \geq 1$, without cooperation, $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$, with the membranes labelled with the numbers $1, 2, \ldots, m$. For each membrane i and for each symbol $a \in V$ we consider a new symbol, a_i, $1 \leq i \leq m$. We define the morphisms h_i by $h_i(a) = a_i$, for all $a \in O, 1 \leq i \leq m$. Denote by T_i the sets $\{a \in O \mid$ there is no rule of the form $a \to x$ in $R_i\}$. We remove from w_i, for $i \neq i_o$, all symbols in T_i (they do not evolve and do not reach the output membrane).

For each $i = 1, 2, \ldots, m$, consider four new symbols $c_i, c_i', \bar{c}_i, d_i$; c_i is meant to tell us that membrane i exists, c_i', d_i will tell us that membrane i was just dissolved, while $\bar{c}_i$ will indicate the fact that membrane i was dissolved at a previous step.

We construct the ET0L system G with the total alphabet V defined by

$$\begin{aligned} V = &\{a_i \mid a \in O, 1 \leq i \leq m\} \cup \{\#\} \\ &\cup \{c_i, c_i', \bar{c}_i, d_i \mid 1 \leq i \leq m\} \cup \{a' \mid a \in T_{i_o}\} \cup T_{i_o}, \end{aligned}$$

the terminal alphabet T_{i_o}, the axiom

$$w = h_1(w_1) \ldots h_m(w_m) c_1 c_2 \ldots c_m,$$

and the tables obtained in the following way. Consider all sets of rules of the following form:

$$\begin{aligned} &\{c_{i_1} \to c_{i_1}, \ldots, c_{i_k} \to c_{i_k}, c_{j_1} \to \#, \ldots, c_{j_l} \to \#, \\ &\bar{c}_{i_1} \to \#, \ldots, \bar{c}_{i_k} \to \#, \bar{c}_{j_1} \to \bar{c}_{j_1}, \ldots, \bar{c}_{j_l} \to \bar{c}_{j_l}, \\ &d_1 \to \#, \ldots, d_m \to \#, c_1' \to \#, \ldots, c_m' \to \#\}, \\ &\text{for}(\{i_1, \ldots, i_k\}, \{j_1, \ldots, j_l\}) \text{ a partition of } \{1, 2, \ldots, m\}. \end{aligned}$$

There are 2^m sets of this form. Let us enumerate them in any given order and denote them by $M_1, M_2, \ldots, M_{2^m}$. Let $E(M_j) = \{i \mid c_i \to c_i \in M_j, 1 \leq i \leq m\}$ (the set of membranes which exist according to M_j).

We consider all sets of rules

$$Q_j = M_j \cup \bigcup_{i \in E(M_j)} R_i, \text{ for } 1 \leq j \leq 2^m.$$

(Because $i \in E(M_j)$, hence the membrane with the label i still exists, the rules in R_i can be applied.) For each such set Q_j we proceed as follows.

For each rule $a \to x \in R_i, i \in E(M_j)$, consider the rule $a_i \to \bar{x}$ constructed in a way similar to that in the proof of Theorem 3.3.1 (with the differences imposed by the use of δ):

(a) if a symbol $(b, here)$ appears in x, then in $\bar{x}$ we put b_i instead of it;
(b) if a symbol (b, out) appears in x, then in $\bar{x}$ we put b_t instead of it, where t is the label of the membrane immediately outside the membrane i; in the case of the skin membrane, instead of (b, out) we put λ;
(c) if a symbol (b, in_t) appears in x, and the set M_j contains the rules $c_t \to c_t$, $\bar{c}_t \to \#$ (that is, membrane t exists), and, moreover, membrane t is adjacent to the rule $a \to x \in R_i$ (this can be checked by examining μ and the information given by M_j about the membranes still existing), then in $\bar{x}$ we put b_t instead of it; if membrane t is not adjacent to the rule, perhaps because it was dissolved, then the rule $a \to \bar{x}$ is removed from the set we construct, because it cannot be applied in the given circumstances.
Moreover, instead of any symbol b_j, for $b \in T_j$, we put λ in $\bar{x}$, with the exception of the output membrane, where we put b', for $b \in T_{i_o}$.

If the rule $a \to x$ introduces the symbol δ, then instead of it in $\bar{x}$ we introduce the symbol d_i.

Let Q'_j be the set of rules obtained in this way. For each rule $c_i \to c_i$ appearing in Q'_j such that R_i contains a rule $a \to v\delta$, we either leave $c_i \to c_i$ unchanged (this is meant to correspond to the case when the rule $a \to v\delta$ is not actually used), or replace it with $c_i \to c'_i$ (this will correspond to the case when the rule $a \to v\delta$ is effectively used, that is the symbol d_i is also introduced in the current string), and we do this in all possible combinations; that is, we produce a series of sets $Q_{j,1}, \ldots, Q_{j,s}$, differing from Q'_j just by rules $c_i \to c_i, c_i \to c'_i$. Of course, for each i, each set $Q_{j,t}$ contains exactly one of these two rules.

To these sets we add the rules $a \to \#$, for $a \in T_{i_o}$, as well as completion rules $a \to a$ for all $a \in V - T_{i_0}$ for which no rule $a \to y$ has already been considered. The obtained sets – we still denote them by $Q_{j,1}, \ldots, Q_{j,s}$ – are tables of our ET0L system. We say that these tables are *of type* Q.

We also consider the following tables:

$$\{c_{i_1} \to c_{i_1}, \ldots, c_{i_k} \to c_{i_k}, c_{j_1} \to \#, \ldots, c_{j_l} \to \#, c_{r_1} \to \#, \ldots, c_{r_s} \to \#,$$
$$\bar{c}_{i_1} \to \#, \ldots, \bar{c}_{i_k} \to \#, \bar{c}_{j_1} \to \#, \ldots, \bar{c}_{j_l} \to \#, \bar{c}_{r_1} \to \bar{c}_{r_1}, \ldots, \bar{c}_{r_s} \to \bar{c}_{r_s},$$
$$c'_{i_1} \to \#, \ldots, c'_{i_k} \to \#, c'_{j_1} \to \bar{c}_{j_1}, \ldots, c'_{j_l} \to \bar{c}_{j_l}, c'_{r_1} \to \#, \ldots, c'_{r_s} \to \#,$$
$$d_{i_1} \to \#, \ldots, d_{i_k} \to \#, d_{j_1} \to \bar{c}_{j_1}, \ldots, d_{j_l} \to \bar{c}_{j_l}, d_{r_1} \to \#, \ldots, d_{r_s} \to \#,$$
$$a_p \to a_q \mid \text{ where } p \in \{j_1, \ldots, j_l\},$$
$$q \text{ is the membrane where a symbol from membrane } p$$
$$\text{will be placed after dissolving membranes } j_1, \ldots, j_l,$$
$$\text{for all } a \in O \text{ and all partitions}$$
$$(\{i_1, \ldots, i_k\}, \{j_1, \ldots, j_l\}, \{r_1, \ldots, r_s\}) \text{ of } \{1, 2, \ldots, m\}\}$$
$$\cup \{a \to \# \mid a \in T_{i_o}\}.$$

(Of course, we also add completion rules $a \to a$ for symbols $a \in V$ for which there is no a-rule already considered.) We say that these tables are *of type* Q$'$.

We also consider the following "final" table:

$$\begin{aligned} Q_f = &\{c_i \to \lambda \mid 1 \le i \le m\} \\ &\cup \{\bar{c}_i \to \lambda \mid 1 \le i \le m, i \ne i_o\} \cup \{\bar{c}_{i_o} \to \#\} \\ &\cup \{a' \to a \mid a \in T_{i_o}\} \\ &\cup \{a \to \# \mid a \in V - (\{b' \mid b \in T\} \cup \{c_i, \bar{c}_i \mid 1 \le i \le m\})\}. \end{aligned}$$

These are all the tables of the ET0L system G.

From the above construction, it is now clear that the use of a table of type Q in the ET0L system G is equivalent to a transition in the system Π. The difference is that after using a table $Q_{j,r}$, in the case of membrane dissolving, the subscripts of the symbols do not correctly indicate the membranes to which they belong. To this end, we have to use a table of type Q$'$ to perform the corrections indicated by the symbols d_t present in it (remember that these symbols tell us which membranes were dissolved during the last transition). On the other hand, as long as symbols c'_t, d'_t are present, no table of type Q can be used: we have introduced rules $c'_t \to \#, d_t \to \#$ in all these tables, hence the trap symbol will be produced. Similarly, for each string there precisely exists only one table of type Q$'$ which can be used without introducing the trap symbol, because of the partition of the set $\{1, \ldots, m\}$ from the definition of tables of type Q$'$, in a set of existing membranes, a set of just dissolved membranes, and a set of membranes which were dissolved at a previous transition.

Thus, we correctly simulate the transitions in Π (by two derivation steps in G), and, conversely, the derivations in G correspond to correct transitions in Π. Moreover, only the halting computations in Π lead to terminal derivations in G: all objects which can evolve in Π are non-terminal symbols in G, hence they should be removed in order to get a terminal derivation. After obtaining a terminal string (that is, after using the table Q_f), no further

derivation step is possible, because of the rules $a \to \#$, for $a \in T_{i_o}$, present in all tables different from Q_f. In the output membrane we always collect terminal objects. If the trap symbol $\#$ is introduced, then it will never be eliminated.

Consequently, $N(\Pi) = length(L(G))$, which concludes the proof. □

Thus, the dissolving action is not sufficient in order to obtain computational completeness (and universality–programmability) in the case of non-cooperative rules. The problem remains *open* for the case when catalytic rules are available, and this is an extension of Problem Q1 (but this time we know that we can generate non-semilinear sets – Theorem 3.4.1).

3.4.2 Priorities Among the Evolution Rules

The universality will be achieved for systems having one more feature: a *priority relation among the rules from each region*, in the form of a partial order relation. Thus, a membrane system with symbol–objects, using both the membrane dissolving action and a priority on the sets of rules, is a construct of the form

$$\Pi = (O, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m), i_o),$$

where all components are as discussed above (also a set K of catalysts can be considered) and ρ_i is a partial order relation on $R_i, 1 \leq i \leq m$. Often, instead of $(r_1, r_2) \in \rho_i$ we will write $r_1 > r_2$.

A rule $r_1 : u_1 \to v_1$ from a set R_i is used in a transition step with respect to Π only if there is no rule $r_2 : u_2 \to v_2$ in R_i which can be applied at the same step and $r_2 > r_1$. That is, at each step, only rules which are maximal among the rules applicable in that region are allowed to be applied.

Note that we interpret the priority in the strong sense (similar to the interpretation used in the ordered grammars considered in Subsection 2.3.7): we have a competition for rule application and not a competition for the objects to be used. An example can illustrate this subtle point: assume that in a region we have the multiset aaa and the rules $r_1 : aa \to bb$ and $r_2 : a \to c$, such that r_1 has a higher priority than r_2. Because r_1 is applicable to the current multiset, it has to be applied, and the obtained multiset is abb; one copy of a remains unused, in spite of the fact that r_2 can evolve it. It is not important that this third copy of a is not used by the rule of a higher priority, what is important is that a rule of a higher priority is applicable (hence applied) and this forbids the use of the rule r_2, of a lower priority.

In biochemical terms, we can interpret this as follows: reaction r_1 is more active than reaction r_2 and it consumes a given resource, say, energy, from the region, hence reaction r_2 cannot take place, in spite of the fact that the necessary input objects are available.

We denote by $NOP_m(\alpha, tar, pri, \delta)$ the family of sets of numbers $N(\Pi)$ generated by P systems of degree at most $m \geq 1$, with rules of type $\alpha \in$

$\{ncoo, cat, coo\}$, using priorities among rules, and the membrane dissolving action. When no bound on the degree of the systems is imposed we replace the subscript m with $*$. As usual, pri or δ is removed when the corresponding feature is not used.

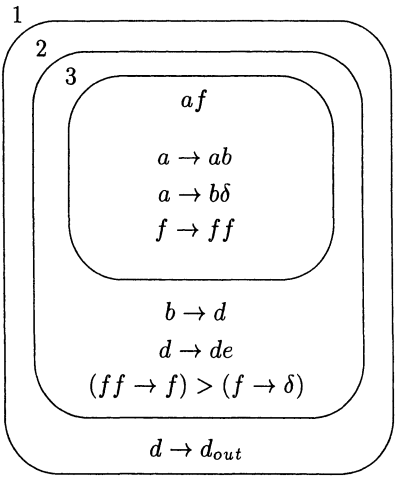


Fig. 3.3. The initial configuration of Π_1 (rules included)

Because the systems of various types $(\alpha, tar, pri, \delta)$ are central in membrane computing, before investigating their power and properties we examine two examples of systems involving both the membrane dissolving action and priorities.

3.4.3 Two Further Examples

We illustrate the functioning of a system of type (coo, tar, pri, δ) with the following system (of degree 3):

$$\begin{aligned}
\Pi_1 &= (O, \mu, w_1, w_2, w_3, (R_1, \rho_1), (R_2, \rho_2), (R_3, \rho_3), 1),\\
O &= \{a, b, d, e, f\},\\
\mu &= [_1[_2[_3\ ]_3]_2]_1,\\
w_1 &= \lambda,\ R_1 = \{d \to d_{out}\},\ \rho_1 = \emptyset,\\
w_2 &= \lambda,\ R_2 = \{b \to d,\ d \to de,\ r_1 : ff \to f,\ r_2 : f \to \delta\},\\
&\quad \rho_2 = \{(r_1, r_2)\},\\
w_3 &= af,\ R_3 = \{a \to ab,\ a \to b\delta,\ f \to ff\},\ \rho_3 = \emptyset.
\end{aligned}$$

The initial configuration of Π_1 (including the rules and the priority relations) is given in Figure 3.3. No objects are present in regions 1 and 2, and so

no rules can be applied in these regions. Hence one has to start in region 3, by using the single copies of objects a and f.

If we iterate the use of rules $a \to ab$ and $f \to ff$ in parallel for all occurrences of a and f currently available, then after n steps, $n \geq 0$, we get n occurrences of b and 2^n occurrences of f. If we then use $a \to b\delta$ instead of $a \to ab$ (note that we always have only one copy of a), then we get $n+1$ occurrences of b and 2^{n+1} occurrences of f, and, moreover, we dissolve membrane 3 (and so region 3 disappears). This means that all the occurrences of objects from region 3 become occurrences of objects from region 2, the rules of region 3 are "lost" (removed), and the rules of region 2 can now be applied to all occurrences of objects present in region 2. As dictated by the priority relation, we have to use the rule $ff \to f$ as much as possible. In one step, we transform b^{n+1} to d^{n+1}, while the number of occurrences of f is halved. In the next step, $n+1$ occurrences of e are produced: each occurrence of d introduces one occurrence of e. At the same time, the number of occurrences of f is halved again.

The priority relation ensures that this step must be iterated n times (each time producing $n+1$ occurrences of e), and then, when only one copy of f will be present, the rule $f \to \delta$ must be used. It dissolves membrane 2 (and so the rules of region 2 are removed), while the objects of region 2 become objects of the skin region, which contains only rules for d. Now, in one step, all copies of this object will be sent out of the system, by using the rule $d \to d_{out}$. No further step is possible, and so the computation stops.

Therefore, we halt with $(n+1)(n+1)$ copies of the object e, for some $n \geq 0$, present in the skin region, the output one. Hence $N(\Pi_1) = \{n^2 \mid n \geq 1\}$.

Up to now we have considered the membrane systems as *generative* devices: one starts from the initial configuration and, because of the non-deterministic evolution, one gets several results, in the form of the number of objects present in the output membrane in the halting configuration. For instance, the system Π_1 discussed above collects in its output membrane different values of $n^2, n \geq 1$. A variant of interest could be a membrane system just *computing* a function f; that is, starting from a configuration which codifies in a given way a number n and halting with $f(n)$ objects in the output membrane. We leave to the reader the task of modifying the system Π_1 in such a way to obtain a system which computes the function $f(n) = n^2$.

Another possible task for a membrane system could be a *decidability* one: start with an instance of a given decidability problem, codified in a certain manner in the initial configuration, and look for the answer to this question, provided in a certain manner at the end of a computation. We illustrate this possibility with the following divisibility problem: we introduce in the input configuration two numbers, n and k, with $k \geq 1$, and we ask whether or not n is a multiple of k. In the affirmative case, we will finish with the object "yes" in the output membrane; in the negative case we will have the object "no" in the output membrane.

The system is the following (of degree 3):

$$\begin{aligned}
\Pi_2 &= (O, \mu, \lambda, a^n c^k d, \lambda, (R_1, \rho_1), (R_2, \rho_2), (\emptyset, \emptyset), 3),\\
O &= \{a, c, c', d, yes, no\},\\
\mu &= [_1 [_2\ ]_2 [_3\ ]_3]_1,\\
R_1 &= \{r_1 : dcc' \to (no, in_3),\ r_2 : d \to (yes, in_3)\},\\
\rho_2 &= \{(r_1, r_2)\},\\
R_2 &= \{r_3 : ac \to c',\ r_4 : ac' \to c,\ r_5 : d \to d\delta\},\\
\rho_2 &= \{(r_3, r_5), (r_4, r_5)\}.
\end{aligned}$$

The structure of Π_2 is better seen in Figure 3.4.

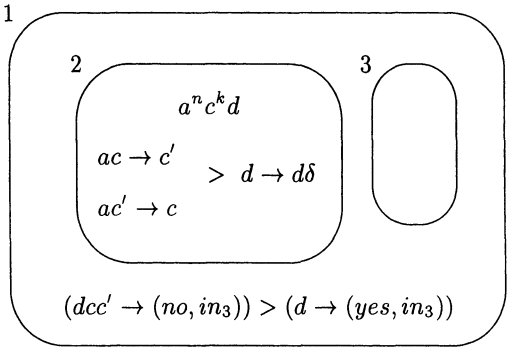


Fig. 3.4. A system deciding whether k divides n

In membrane 2 we subtract k from n, repeatedly (by the rules $ac \to c'$ and $ac' \to c$: at each step, k copies of a disappear, while c is reproduced, primed or not primed, alternating the priming from one step to another one).

The rules $ac \to c'$ and $ac' \to c$ have priority over the rule $d \to d\delta$, therefore we can dissolve membrane 2 only after exhausting the n occurrences of a. If n is a multiple of k – and only in this case – then we never have both occurrences of c and c' simultaneously present in membrane 2 (or in membrane 1, after dissolving membrane 2). Therefore, the rule $dcc' \to (no, in_3)$ is used in membrane 1 if and only if n is not a multiple of k. If this rule is not applicable, then $d \to (yes, in_3)$ may be used. In both cases, in membrane 3 we get the correct answer.

Note that only one of the rules $dcc' \to (no, in_3)$ and $d \to (yes, in_3)$ can be used, because we have only one occurrence of d, and that the computation stops after using any of these rules.

In conclusion, the computation always stops and the output membrane contains only one object, the correct answer to the question whether or not k divides n.

When we formulated the problem, we imposed that $k \geq 1$. If we start with $k = 0$ copies of c in membrane 2, then the rule $d \to d\delta$ is immediately applied, and $d \to (yes, in_3) \in R_3$ will give a wrong answer. However, this can be easily handled: let us replace the rule $r_2 : d \to (yes, in_3)$ from R_1 with $r_2' : dc \to (yes, in_3)$ and $r_2'' : dc' \to (yes, in_3)$, with the priority relation $\rho_1 = \{(r_1, r_2'), (r_1, r_2'')\}$. If no copy of c is present in the system, then no rule from region 1 can be used, no object will be sent to membrane 3, and hence we get no answer. If we want to be fully accurate, then we can add to R_1 one further rule, $r_0 : d \to (\textit{illegal-operation}, in_3)$, with the lowest priority.

This is just a proof that the "programming techniques" offered by systems of type (coo, tar, pri, δ) are very rich.

The systems considered above were cooperative systems and always the rules were either "propagating", in the sense that each rule $u \to v$ had $|u| \leq |v|$, or easy to modify in order to obtain propagating rules.

Actually, the property of a system to be propagating is easy to achieve by replacing each rule $u \to v$ such that $|u| > |v|$ by a rule of the form $u \to vd^i$, where d is a dummy object, and $i = |u| - |v|$. If the dummy object appears in any membrane different from the output membrane, then it never evolves; in the set R_{i_o} we also consider the rule $d \to (d, out)$, so that the dummy object is not counted in the result of a computation. Because of this easy way of passing from non-propagating systems to propagating systems, we do not pay any attention to the propagation feature. Still, a problem appears here with the very definition of propagation: the previous trick works if the propagation is defined at the level of rules, but sending objects outside the system is also a way to remove objects. However, we can proceed as above and instead of objects (a, out) introduced by rules used in the skin region we introduce again the dummy object d. Thus, if the output membrane is not the skin membrane, then we can enforce the propagation, even in the strong sense, when we are not allowed to send objects out of the system. Consequently, from now on we will no longer discuss propagation.

3.4.4 The Power of Priority

We have seen in Theorem 3.4.2 that the membrane dissolving feature does not entail the computational completeness of symbol–objects membrane systems with non-cooperative rules, while the case of systems with catalytic rules is still unsettled. When making use of a priority relation, the situation is somewhat reversed: we get universality for the catalytic systems, but the problem is still *open* for the non-cooperative systems (Problem Q2).

Theorem 3.4.3. $NOP_2(cat, tar, pri) = NRE$.

Proof. We only have to prove the inclusion $NRE \subseteq NOP_2(cat, tar, pri)$. Consider a matrix grammar with appearance checking $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form. Assume the grammar given in the standard

notation, as agreed in Subsection 2.3.7. In particular, this means that for each $X \in N_1$ there is a matrix $(X \to Y, A \to \alpha)$ in M.

We construct the following system with catalysts and priorities:

$$\Pi = (O, \{c\}, [_1[_2\ ]_2]_1, w_1, \lambda, (R_1, \rho_1), (\emptyset, \emptyset), 2),$$

where

$$O = N_1 \cup N_2 \cup \{a, c, H, Z, \#\},$$
$$w_1 = cX_{init}A_{init}H,$$

and the set R_1 contains the following rules (h is the morphism defined by $h(\alpha) = \alpha$, $\alpha \in N_2$, and $h(a) = (a, in_2)$):

$r_i : X \to Y$,
$r'_i : cA \to c\ h(x)$, for $m_i : (X \to Y, A \to x)$, $1 \leq i \leq k$,
$r_t : cH \to c\#$,
$r_\infty : \# \to \#$,
$r_i : cX \to cY$, for $m_i : (X \to Y, A \to \#)$, $k+1 \leq i \leq n$,
$r_A : A \to \#$, for all $A \in N_2$,
$r_f : H \to \lambda$.

The priorities are as follows:

$r_i > r'_j$, for $1 \leq i, j \leq k$ and $i \neq j$,
$r_i > r_A$, for $1 \leq i \leq k$ and $A \in N_2$,
$r_i > r_f$, for $1 \leq i \leq k$,
$r_i > r_B$, for $k+1 \leq i \leq n$ and $B \neq A$, where $m_i : (X \to Y, A \to \#)$,
$r_i > r_f$, for $k+1 \leq i \leq n$.

All the work is done in the skin membrane; membrane 2 is only used for collecting the result of computations.

Let us assume that in the skin membrane we have a multiset $cXwH$; initially, $X = X_{init}$ and $w = A_{init}$. Because for each $X \in N_1$ there is a rule $X \to Y$ in a matrix from M, there is a rule $r_i, 1 \leq i \leq n$, which can be applied, hence the rule $r_f : H \to \lambda$ cannot be used.

We have two possibilities: to use a rule $r_i : X \to Y$ with $1 \leq i \leq k$, or a rule $r_i : cX \to cY$ with $k+1 \leq i \leq n$.

A rule of the first type forbids the use of any rule $r_B : B \to \#$, $B \in N_2$, as well as of all rules r'_j with $j \neq i$. Thus, if $m_i : (X \to Y, A \to x)$ and A is present in w, then the rule $r'_i : cA \to c\ h(x)$ is used, and in this way we correctly simulate the matrix m_i. If A is not present, then the rule $r_t : cH \to c\#$ must be used (because of the maximal parallelism). The trap object $\#$ will evolve forever by means of the rule $r_\infty : \# \to \#$, hence the computation never stops.

If we start with the rule $r_i : cX \to cY$, for some $m_i : (X \to Y, A \to \#)$, $k+1 \leq i \leq n$, then the only applicable rule is $r_A : A \to \#$, because r_i has

priority over all rules r_B with $B \neq A$. Thus, if A is present, then the trap object is introduced, otherwise not.

Therefore, in both cases we correctly simulate a matrix from M, and the process can be iterated. When the symbol Z is introduced (this means that the derivation in G is terminal), because no rule $r_i, 1 \leq i \leq n$, can be applied, we can use the rule $r_f : H \to \lambda$, and the computation stops. The objects c and Z remain in the skin membrane; all copies of a were sent to the output membrane. (If we "wrongly" use the rule $r_t : cH \to c\#$, instead of $r_f : H \to \lambda$, then the computation never stops, but we can avoid this by choosing the rule r_f).

Consequently, $length(L(G)) = N(\Pi)$, and the proof is complete. □

We do not know whether or not a similar result holds true for the non-cooperative systems (probably not, because of the parallelism, which cannot be controlled without catalysts), but the following result indicates that also in this case the priority adds power.

Theorem 3.4.4. $NET0L \subseteq NOP_1(ncoo, tar, pri)$.

Proof. We have mentioned in Subsection 2.3.5 that each ET0L language can be generated by an ET0L system with two tables only. Let $G = (V, T, w, R_1, R_2)$ be such a system. For each letter $a \in V$, let a' be a new symbol, let V' be the set $\{a' \mid a \in V\}$, and h be the morphism defined by $h(a) = a', a \in V$. Let $d, d_1, d_2, \#$ be new symbols. Assume that $R_1 = \{p_1, \ldots, p_u\}$ and $R_2 = \{q_1, \ldots, q_v\}$.

We construct the P system of degree 1,

$$\Pi = (V \cup V' \cup \{d, d_1, d_2, \#\}, [_1\ ]_1, d\ h(w), (R_1', \rho_1), 1),$$

with the following rules:

$$\begin{aligned}
&r_1 : d \to d_1,\\
&r_2 : d \to d_2,\\
&r_1' : d_1 \to d,\\
&r_2' : d_2 \to d,\\
&r_3 : d \to (d, out),\\
&r_4 : e \to (e, out),\\
&p_i' : a' \to h(x), \text{ for } p_i : a \to x \in R_1, 1 \leq i \leq u,\\
&q_i' : a' \to h(x), \text{ for } q_i : a \to x \in R_2, 1 \leq i \leq v,\\
&r_a : a' \to a, \text{ for } a \in T,\\
&r_a : a' \to \#, \text{ for } a \in V - T,\\
&r_\infty : \# \to \#,
\end{aligned}$$

and the priority relation ρ_1 given as follows:

$$
\begin{aligned}
&r_1 > p'_i,\ r_2 > p'_i,\ r_3 > p'_i, \text{ for } 1 \le i \le u,\\
&r_1 > q'_i,\ r_2 > q'_i,\ r_3 > q'_i, \text{ for } 1 \le i \le v,\\
&r_1 > r_a,\ r_2 > r_a,\ r'_1 > r_a,\ r'_2 > r_a, \text{ for } a \in V,\\
&r'_1 > q'_i, \text{ for } 1 \le i \le v,\\
&r'_2 > p'_i, \text{ for } 1 \le i \le u.
\end{aligned}
$$

The system Π works as follows. To the multiset described by the string $d\ h(z)$ we can apply one of the rules r_1, r_2, r_3. If one of r_1, r_2 is used, then no rule can be applied to the objects in $h(z)$ because of the priority relation. The rules r_1, r_2 select the table to be simulated: in the presence of d_i only rules from R_i can be simulated, $i = 1, 2$. This is done in one step, in the same way as in an ET0L system (note that the completeness of tables ensures the fact that all objects evolve). After such a step, the objects d_1, d_2 return to d and the process can be iterated. When we use the rule r_3, no rule p'_i, q'_i can be used, but now the rules r_a can be used, because no rule with a higher priority can be applied. This means that at the same time when we remove the object d from the system, we have to replace with the trap object # each a' such that $a \in V - T$; for the symbols in T we just remove their primes. Therefore, the computation stops when no symbol from $V - T$ is present and continues forever in the opposite case. This is exactly the condition for including a string in the language generated by an ET0L system. Consequently, $length(L(G)) = N(\Pi)$ and this concludes the proof. □

3.4.5 The Power of Synchronization

The existence of a global clock, the same for all regions of a membrane system, is a fundamental feature of our computing mechanisms, but it is of a theoretical inspiration, not necessarily corresponding to a biological reality. Both for this reason and because such a problem is mathematically natural, the question arises about what happens if we remove the global clock, that is we remove the synchronization.

This means that at each step, *any number of rules* may be used in each region of the system. Instead of the maximal parallelism, we have a sort of "free parallelism", as no rule, one rule, or as many rules as we want can be used at the same time in a region (but not necessarily *one rule*, as in sequential systems which we will briefly mention in Section 8.6). Thus, non-synchronization amounts to supposing that in each region we have a rule of the form $a \to a$ for all symbols of the alphabet. Using these rules prevents the use of other rules, hence removes the maximal parallelism, and the synchronization.

Actually, not too much is known about non-synchronized P systems, it is still a *research topic* (Problem Q3). Here we will consider the particular case of using *bi-stable catalysts*. Such a catalyst has two states, c and $\bar{c}$, and the rules involving catalysts may switch between these states. That is, the allowed rules are of the forms $ca \to cv$, $ca \to \bar{c}v$, $\bar{c}a \to \bar{c}v$, and $\bar{c}a \to cv$.

(Actually, from the proofs below one can see that rules of the forms $ca \to \bar{c}v$ and $\bar{c}a \to cv$, hence with the catalysts always changing their states, suffice.)

We indicate the use of such bi-stable catalysts by writing $2cat$ instead of cat, and thus the family of sets of numbers computed by non-synchronized P systems of degree at most $m \geq 1$, using bi-stable catalysts, priorities, and the membrane dissolution feature, is denoted by $NOuP_m(2cat, tar, pri, \delta)$.

Clearly, bi-stable catalysts are at least as powerful as usual catalysts, and this is true both in the synchronized and in the non-synchronized case.

If no priority is used, then removing the synchronization strictly decreases the power of P systems, but this is not the case when using priorities and bi-stable catalysts.

Theorem 3.4.5. *$NOuP_*(\alpha, tar, \gamma) = NOuP_1(\alpha, \gamma) = NCF$, for all $\alpha \in \{ncoo, cat, 2cat\}$ and $\gamma = \delta$ or γ is missing.*

Proof. The system constructed at the beginning of the proof of Theorem 3.3.2 in order to compute the length set of a context-free language is a non-synchronized one, as rules $A \to A$ are available for all non-terminals. Consequently, $NCF \subseteq NOuP_1(nCoo)$. Because we have the equality $NCF = NMAT^\lambda$ (Subsection 2.3.7), in order to complete the proof it is sufficient to show that $NOuP_*(2cat, tar, \delta) \subseteq NMAT^\lambda$.

Consider a system $\Pi = (O, K, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$, of degree $m \geq 1$, with bi-stable catalysts, without priorities among rules, possibly using the dissolving action, and working in the non-synchronized manner.

In order to keep track of the current state of membranes, we consider the vectors of the form $(\alpha_1, \ldots, \alpha_m)$, where $\alpha_i \in \{i, i', i_\delta\}, 1 \leq i \leq m$, with the following meaning: $\alpha_i = i$ indicates that membrane i exists, $\alpha_i = i'$ indicates that membrane i was dissolved, and $\alpha_i = i_\delta$ says that a rule introducing δ was used in membrane i but we still can use such rules before "effectively" dissolving this membrane (the idea is that several rules which introduce δ can be used at the same step; when simulating them in the sequential manner specific to a matrix grammar, one by one, we have to "destroy" the membrane only after simulating all these rules, otherwise no two rules can be used together if both of them introduce δ; more details will be given below). We denote by SM (from "the state of membranes") the set of vectors as above.

For a membrane i and a vector $(\alpha_1, \ldots, \alpha_m) \in SM$, we denote by $low(i, (\alpha_1, \ldots, \alpha_m))$ the set of membranes still present in the configuration described by $(\alpha_1, \ldots, \alpha_m)$ which are immediately inside membrane i (an object from membrane i can be sent to any of these lower membranes by commands (a, in_j), while objects from these membranes can be sent to membrane i by commands of the form (a, out)). If i is an elementary membrane, then $low(i, (\alpha_1, \ldots, \alpha_m)) = \emptyset$.

Similarly, we denote by $up(i, (\alpha_1, \ldots, \alpha_m))$ the membrane placed immediately outside membrane i in the configuration described by $(\alpha_1, \ldots, \alpha_m) \in SM$; if i is the skin membrane, then $up(i, (\alpha_1, \ldots, \alpha_m)) = \emptyset$.

It is obvious that both these mappings can be computed using the information stored in $(\alpha_1, \ldots, \alpha_m)$.

Consider also the morphisms h_i defined by $h_i(a) = (a, i)$, for all $a \in O$ and $1 \leq i \leq m$.

We construct a matrix grammar $G = (N, T, S, M)$ with

$$\begin{aligned} N &= SM \cup \{\langle \alpha_1, \ldots, \alpha_m \rangle' \mid (\alpha_1, \ldots, \alpha_m) \in SM\} \\ &\quad \cup \{(a, i) \mid a \in O, 1 \leq i \leq m\} \\ &\quad \cup \{X_i \mid 1 \leq i \leq m\} \cup \{S\}, \\ T &= \{\langle a, i \rangle \mid a \in O, 1 \leq i \leq m\} \\ &\quad \cup \{\langle \alpha_1, \ldots, \alpha_m \rangle \mid (\alpha_1, \ldots, \alpha_m) \in SM\}, \end{aligned}$$

and the following matrices:

1. $(S \to (1, 2, \ldots, m) X_1 X_2 \ldots X_m h_1(w_1) h_2(w_2) \ldots h_m(w_m))$.
   (We start with all the m membranes present, with the objects in the initial multisets indexed by the labels of the membranes where they are placed. The symbols X_i are used for introducing new objects of the form (a, i), corresponding to membrane i.)
2. For each $i = 1, 2, \ldots, m$ and for each rule $ca \to c'v$ in R_i, with $c, c' \in K$ (c, c' are either identical, or one is the barred variant of the other), we consider all the matrices containing a rule of the form
   $(\alpha_1, \ldots, \alpha_m) \to (\alpha_1, \ldots, \alpha_m)$, with $\alpha_i = i$ and with
   $j \in low(i, (\alpha_1, \ldots, \alpha_m))$ for all j such that (b, in_j) appears in v,
   as well as all rules
   $(c, i) \to (c', i)$,
   $(a, i) \to \lambda$,
   $X_i \to X_i(b, i)$, for each $(b, here)$ which appears in v,
   $X_j \to X_j(b, j)$, for each (b, in_j) which appears in v,
   $X_j \to X_j(b, j)$, for each (b, out) which appears in v,
   where $j = up(i, (\alpha_1, \ldots, \alpha_m))$.
   (The rule $ca \to c'v$ is simulated by a matrix of this form, for each membrane structure where this is possible; the information about the membrane structure is available in the vector $(\alpha_1, \ldots, \alpha_m)$. Note how the second component of symbols (b, j) is changed according to the target indications given by the elements of v and that when i is the skin membrane and (b, out) appears in v, no symbol (b, j) is introduced.)
3. If we have a rule of the form $a \to v$, then we consider the matrices as above, without the rule $(c, i) \to (c', i)$.
4. If in a rule of any form, with or without catalysts, we also introduce the symbol δ, then in matrices as above we replace the rule
   $(\alpha_1, \ldots, i, \ldots, \alpha_m) \to (\alpha_1, \ldots, i, \ldots, \alpha_m)$ with the rule
   $(\alpha_1, \ldots, i, \ldots, \alpha_m) \to (\alpha_1, \ldots, i_\delta, \ldots, \alpha_m)$.
   Moreover, we also consider all matrices associated with rules $ca \to c'v\delta$ or $a \to v\delta$ from the set R_i, constructed as above, and having the rule

$(\alpha_1, \ldots, \alpha_m) \to (\alpha_1, \ldots, \alpha_m)$ replaced with
$(\alpha_1, \ldots, i_\delta, \ldots, \alpha_m) \to (\alpha_1, \ldots, i_\delta, \ldots, \alpha_m)$.
(When simulating a rule which introduces the dissolving symbol, we proceed as with any rule without δ, but we allow the simulation of other dissolving rules before actually marking the membrane i as being dissolved. The rules which do not introduce δ are not allowed to be simulated in the presence of i_δ, because they can be simulated before the dissolving rule.)

5. We also consider the following matrices, for all $i = 1, 2, \ldots, m$:
   $((\alpha_1, \ldots, i_\delta, \ldots, \alpha_m) \to (\alpha_1, \ldots, i', \ldots, \alpha_m))$.
   (The fact that membrane i was dissolved is registered in the vector describing the membrane structure.)
   $((\alpha_1, \ldots, \alpha_m) \to (\alpha_1, \ldots, \alpha_m),\ (b, i) \to (b, j))$,
   for $\alpha_i = i'$, $b \in O$, and $j = up(i, (\alpha_1, \ldots, \alpha_m))$.
   (After dissolving a membrane, its objects can be renamed in such a way as to assign them to the new region to which they belong.)
6. In order to finish the derivation in G we consider the following matrices:
   $((\alpha_1, \ldots, \alpha_m) \to \langle \alpha_1, \ldots, \alpha_m \rangle')$.
   (From now on no rule from Π can be simulated.)
   $(\langle \alpha_1, \ldots, \alpha_m \rangle' \to \langle \alpha_1, \ldots, \alpha_m \rangle', (b, i) \to \langle b, i \rangle)$,
   for all $(\alpha_1, \ldots, \alpha_m) \in SM$ and $b \in O, 1 \leq i \leq m$.
   (We pass to terminal symbols of the form $\langle b, i \rangle$.)
   $(\langle \alpha_1, \ldots, \alpha_m \rangle' \to \langle \alpha_1, \ldots, \alpha_m \rangle)$.
   (The derivation is finished.)

From the previous explanations and taking into consideration that Π is a non-synchronized system, one can see that the derivations in G correspond to correct computations in Π and, conversely, each computation in Π can be simulated by a derivation in G.

Still, two problems remain. First, we have not checked in G whether the computation halts. Second, it is possible that in the terminal string generated by G the symbols are not "in the right place": symbols of the form $\langle b, i \rangle$ can exist although membrane i was dissolved.

We can solve these problems by intersecting $L(G)$ with appropriate regular languages.

Consider first the language

$$L_1 = \{\langle \alpha_1, \ldots, i', \ldots, \alpha_m \rangle x \langle b, i \rangle y \mid \text{for all } 1 \leq i \leq m, b \in O,\\ x, y \in T^*, \text{ and } \alpha_j \in \{j, j'\}, 1 \leq j \leq m\}.$$

It consists of strings with "wrongly placed symbols". Thus, the complement of this language, $T^* - L_1$, contains only strings which describe configurations where the objects are correctly placed. Intersecting $L(G)$ with $T^* - L_1$ (which is a regular language), we obtain a language L_2 which is still in the family MAT^λ (this family is closed under intersection with regular languages).

Consider now the language

$$\begin{aligned} L_3 = \{&\langle\alpha_1,\ldots,i,\ldots,\alpha_m\rangle x_1\langle b,i\rangle x_2 \mid \text{for all } b\to v\in R_i,\\ &1\le i\le m,\ x_1,x_2\in T^*,\ \text{and } \alpha_j\in\{j,j'\}, 1\le j\le m\}\\ \cup\ \{&\langle\alpha_1,\ldots,i,\ldots,\alpha_m\rangle x_1\langle c,i\rangle x_2\langle b,i\rangle x_3,\\ &\langle\alpha_1,\ldots,i,\ldots,\alpha_m\rangle x_1\langle b,i\rangle x_2\langle c,i\rangle x_3 \mid\\ &\text{for all } cb\to c'v\in R_i, 1\le i\le m, x_1,x_2,x_3\in T^*,\\ &\text{and } \alpha_j\in\{j,j'\}, 1\le j\le m\}. \end{aligned}$$

The strings in L_3 correspond to multisets where at least a rule from Π can still be applied. Thus, the complement of this language, $T^* - L_3$, corresponds to halting configurations. Intersecting L_2 with L_3, we get a language L_4 which is again a matrix language and corresponds to halting configurations with respect to Π.

We still have to extract from the strings of L_4 the output as that associated with computations in Π. To this end, we can use the morphism h defined by

$$h(\langle b,i\rangle) = \begin{cases} b, & \text{if } i = i_o,\\ \lambda, & \text{otherwise,}\end{cases}$$

and

$$h(\langle\alpha_1,\ldots,\alpha_m\rangle) = \lambda,\ \text{for } \alpha_j\in\{j,j'\}, 1\le j\le m.$$

We clearly have $length(h(L_4)) = N(\Pi)$; because the family MAT^λ is closed under arbitrary morphisms, we obtain $N(\Pi)\in NMAT^\lambda$, and this completes the proof. □

The following consequence of this result makes clear the fact that (without priorities) the non-synchronization entails a loss of generative power.

Corollary 3.4.2. $NOP_2(ncoo, tar, \delta) - NOuP_*(2cat, tar, \delta) \neq \emptyset$.

Proof. From Theorem 3.4.5 and the fact that NCF contains only semilinear sets, it follows that each set from the family $NOuP_*(2cat, tar, \delta)$ is semilinear. However, $NOP_2(ncoo, tar, \delta)$ contains non-semilinear sets, as we have seen in Theorem 3.4.1. □

If a priority relation is used, then the loss in power entailed by the non-synchronization is compensated by the power of the bi-stable catalysts: we again get the computational completeness (for systems with only two membranes).

Theorem 3.4.6. $NOuP_2(2cat, tar, pri) = NRE$.

Proof. It is enough to prove the inclusion $NRE \subseteq NOuP_2(2cat, tar, pri)$.

Consider a matrix grammar $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, given in the standard notation, as specified in the convention from Subsection 2.3.7.

We construct the system of degree 2 with bi-stable catalysts

$$\Pi = (O, K, [_1[_2\ ]_2]_1, w_1, \lambda, (R_1, \rho_1), (\emptyset, \emptyset), 2),$$

where

$$\begin{aligned} O &= N \cup T \cup K \cup \{f_i, f_i', f_i'', \bar{f}_i, \bar{f}_i' \mid 1 \le i \le n\} \cup \{a, f, \bar{f}, Z, \#\}, \\ K &= \{c_i, \bar{c}_i, d_i, \bar{d}_i \mid 1 \le i \le n\} \cup \{d_{n+1}, \bar{d}_{n+1}\}, \\ w_1 &= X_{init} A_{init} f \bar{c}_1 \bar{c}_2 \ldots \bar{c}_n \bar{d}_1 \bar{d}_2 \ldots \bar{d}_n \bar{d}_{n+1}, \end{aligned}$$

and the set R_1 contains the following rules:

For each matrix $m_i : (X \to Y, A \to x), 1 \le i \le k$, we introduce the rules
$\bar{c}_i f_i \to c_i \bar{f}_i$,
$c_i X \to \bar{c}_i Y$,
$\bar{c}_i \bar{f}_i \to c_i \bar{f}_i'$,
$c_i \bar{f}_i' \to \bar{c}_i f_i'$,
$\bar{d}_i f_i' \to d_i f_i''$,
$d_i A \to \bar{d}_i x$,
$\bar{d}_i f_i'' \to d_i \bar{f}$,
$d_i \bar{f} \to \bar{d}_i f$.
For each matrix $m_i : (X \to Y, A \to \#), k+1 \le i \le n$, we introduce the rules
$\bar{c}_i f_i \to c_i \bar{f}_i$,
$c_i X \to \bar{c}_i Y$,
$\bar{c}_i \bar{f}_i \to c_i \bar{f}_i'$,
$c_i \bar{f}_i' \to \bar{c}_i f_i'$,
$\bar{d}_i f_i' \to d_i f_i''$,
$d_i A \to \bar{d}_i \#$,
$d_i f_i'' \to \bar{d}_i f$,
with the priority relation $d_i A \to \bar{d}_i \# \ > \ d_i f_i'' \to \bar{d}_i f$.
We also consider the following rules:
$a \to (a, in_2)$,
$f \to f_i$, for each $1 \le i \le n$,
$f_i \to \#$,
$f_i' \to \#$,
$\bar{f}_i' \to \#$,
$\bar{f}_i \to \#$,
$f_i'' \to \#$, for each $1 \le i \le n$,
$\alpha \to \#$, for each $\alpha \in N$,
$\# \to \#$,
$\bar{d}_{n+1} Z \to d_{n+1}$,
$d_{n+1} f \to \bar{d}_{n+1}$.

Initially, only a rule $f \to f_i$ can be used. It selects the matrix m_i to be simulated. The simulation starts by "unlocking" the catalyst c_i, by using the

rule $\bar{c}_i f_i \to c_i \bar{f}_i$; this catalyst helps to simulate the rule $X \to Y$ and gets back the overbar. By using the rules $\bar{c}_i \bar{f}_i \to c_i \bar{f}'_i$ and $c_i \bar{f}'_i \to \bar{c}_i f'_i$, we get the primed variant of f_i, which "unlocks" the catalyst d_i, which helps to simulate the rule $A \to x$. We can return to f if and only if the matrix is correctly simulated. (Note that in each step we can use a rule of the form $\alpha \to \#$, for α a variant of f_i. This will make the computation never halt. Thus, we have to proceed using the rules in the sequence which corresponds to simulating the matrix m_i.) Also the catalysts c_i, d_i return to their barred versions, hence the process can be iterated.

Whenever a copy of a is introduced, it is immediately sent to the output membrane.

In the case of matrices $(X \to Y, A \to \#)$, the simulation of the rule $X \to Y$ is done as above, while the appearance checking application of the rule $A \to \#$ is ensured by the priority relation: if A is present, then the trap object $\#$ is introduced and the computation never stops; if A is not present, then the rule $d_i f''_i \to \bar{d}_i f$ can be used and we return to f (and the barred version of d_i), hence the process can be iterated.

When the object Z is introduced (hence the derivation in G is terminal), we can finish the computation by removing the object f (note that Z is introduced before f, hence we use the rule $\bar{d}_{n+1} Z \to d_{n+1}$ before the appearance of f, and hence we can immediately use the rule $d_{n+1} f \to \bar{d}_{n+1}$).

Consequently, $N(\Pi) = length(L(G))$. At each step, we can use at most one rule of R_1, hence Π works in a non-synchronized manner, and this observation concludes the proof. □

The reader is requested to note that in the previous proof we have used the catalysts in a restricted way: c is always transformed into $\bar{c}$ and $\bar{c}$ is always transformed into c. If we do not want to obtain this "flip–flop normal form", then the construction will be slightly shorter: instead of two rules $\bar{c}_i \bar{f}_i \to c_i \bar{f}'_i$, $c_i \bar{f}'_i \to \bar{c}_i f'_i$ it is enough to use the rule $\bar{c}_i \bar{f}_i \to \bar{c}_i f'_i$.

The use of priorities can be considered restrictive from a biochemical point of view, but we have seen in Theorem 3.4.5 and its corollary that without priorities the power of non-synchronized systems is strictly smaller than the power of synchronized systems also using priorities.

However, in the case of bi-stable catalysts, the standard synchronization or a sort of *minimal synchronization* can compensate for the use of priorities.

Theorem 3.4.7. $NOP_2(2cat, tar) = NRE$.

Proof. Let $G = (N, \{a\}, S, M, F)$ be a matrix grammar in the Z-binary normal form, in the standard notation. We construct the system

$$\Pi = (O, K, [_1[_2\ ]_2]_1, w_1, \lambda, R_1, \emptyset, 2),$$

where (h is the morphism defined by $h(a) = (a, in_2)$, and $h(A) = A$, for $A \in N_2$):

$$
\begin{aligned}
O &= N \cup K \cup \{a, E, \#\} \cup \{X' \mid X \in N_1\},\\
K &= \{c_i, \bar{c}_i \mid 1 \le i \le n\},\\
w_1 &= X_{init} A_{init} E F c_1 c_2 \ldots c_n,\\
R_1 &= \{c_i X \to \bar{c}_i(Y, in_2),\\
&\qquad \bar{c}_i E \to c_i \#,\\
&\qquad \bar{c}_i A \to c_i\ h(x) \mid m_i : (X \to Y, A \to x), 1 \le i \le k\}\\
&\quad \cup \{c_i X \to \bar{c}_i(Y', in_2),\\
&\qquad \bar{c}_i A \to c_i \#,\\
&\qquad \bar{c}_i Y' \to c_i Y \mid m_i : (X \to Y, A \to \#), k+1 \le i \le n\}\\
&\quad \cup \{\# \to \#\},\\
R_2 &= \{X \to (X, out),\\
&\qquad X' \to (X', out) \mid X \in N_1\}.
\end{aligned}
$$

This system works as follows. If in the skin region we have a multiset of the form $XwEc_1 \ldots c_n$, then we have to use a rule $c_i X \to \bar{c}_i(Y, in_2)$, or a rule $c_i X \to \bar{c}_i(Y', in_2)$ (no other rule is applicable). In the first case we simulate the matrix $m_i : (X \to Y, A \to x), 1 \le i \le k$: the only rules from R_1 applicable at the next step are $\bar{c}_i E \to c_i \#$ and $\bar{c}_i A \to c_i\ h(x)$, hence we have to correctly simulate also the second rule of the matrix.

If we start with a rule $c_i X \to \bar{c}_i(Y', in_2)$, $k+1 \le i \le n$, then the only rule from R_1 which can be used at the next step is $\bar{c}_i A \to c_i \#$, providing that A is present. If this is not the case, then $\bar{c}_i$ waits one step, and it returns to c_i at the next step by means of the rule $\bar{c}_i Y' \to c_i Y$. Thus, also the matrix $m_i : (X \to Y,\ A \to \#)$ is correctly simulated. The process can be iterated. Each occurrence of a is sent to membrane 2. When the object Z is produced, the computation stops. The equality $length(L(G)) = N(\Pi)$ is obvious, hence the proof is complete. □

Note that in each step, in each region, at most one rule of the previous system can be used. Therefore, instead of synchronization by means of the maximal parallelism, we can consider a *minimal synchronization* in the following form: in each step, in each region where at least one rule *can* be used, at least one rule *must* pe used. The previous proof shows that bi-stable catalysts ensure the universality (of systems with two membranes) also in the case of this minimal synchronization.

The construction from the proof of Theorem 3.4.7 always changes the barred and the non-barred versions of the catalysts, hence again a "flip–flop normal form" is obtained.

As the previous results illustrate, the bi-stable catalysts are very powerful from a computational point of view, and an "explanation" of this fact is their short-term memory. Switching from c to $\bar{c}$ and conversely gives a piece of information similar to that carried by the states of a finite automaton or a Turing machine, and this is very useful – especially when we have at our

disposal a large number of such "states", as in the proofs of Theorems 3.4.6 and 3.4.7, where we have used sets of catalysts of the form $c_1, \ldots, c_n$.

3.5 A Formal Definition

For the sake of completeness, we now give a mathematical definition of a membrane system (with symbol–objects, using cooperative rules, priorities, and the membrane dissolving action) and of the set of natural numbers generated by it, in spite of the fact that, in order to preserve the biological intuition and to keep the arguments to a resonable length, in the following chapters of the book we will work with the partially formal definition also used in the previous sections.

We start by defining the notion of a membrane structure in terms of parenthetic expressions (this way of presenting the membrane structures has already been used and will be widely used below).

Consider the language MS over the alphabet $\{[,]\}$, whose strings are recurrently defined as follows:

1. $[\,] \in MS$;
2. If $\mu_1, \ldots, \mu_n \in MS$ for some $n \geq 1$, then $[\mu_1 \ldots \mu_n] \in MS$.

Note the difference between MS and the Dyck language over $\{[,\ ]\}$: all strings from MS have an external pair of matching parentheses, while the Dyck language also contains strings of the form $[\ldots][\ldots]$, which are not in the language MS.

Consider now the following relation over the elements of MS: $x \sim y$ if and only if we can write the two strings in the form $x = \mu_1\mu_2\mu_3\mu_4$, $y = \mu_1\mu_3\mu_2\mu_4$, for $\mu_1\mu_4 \in MS$ and $\mu_2, \mu_3 \in MS$ (two neighboring pairs of parentheses are interchanged, together with their contents). We denote by $\sim^*$ the reflexive and transitive closure of the relation $\sim$. This is clearly an equivalence relation. We denote by $\overline{MS}$ the set of equivalence classes of MS with respect to this relation. The elements of $\overline{MS}$ are called *membrane structures*.

Each matching pair of parentheses $[,]$ appearing in a membrane structure is called a *membrane*. The number of membranes in a membrane structure μ is called the *degree* of μ and is denoted by $deg(\mu)$. The external membrane of a membrane structure μ is called the *skin* membrane of μ. A membrane which appears in $\mu \in \overline{MS}$ in the form $[\,]$ (no other membrane appears inside the two parentheses) is called an *elementary* membrane.

The *depth* of a membrane structure μ, denoted by $dep(\mu)$, is defined recurrently as follows:

1. if $\mu = [\,]$, then $dep(\mu) = 1$;
2. if $x = [\mu_1 \ldots \mu_n]$, for some $\mu_1, \ldots, \mu_n \in MS$,
   then $dep(\mu) = \max\{dep(\mu_i) \mid 1 \leq i \leq n\} + 1$.

In general, in a membrane structure with m elements, the membranes are labelled by the elements of the set $\{1, \ldots, m\}$.

The formalization which follows will not use this way of giving a membrane structure, but will use concepts from graph theory.

An *undirected graph* γ is an ordered pair (V, E), where V is a finite set (its elements are called *vertices* or *nodes*) and E is a finite set of unordered pairs of vertices, $\{u, v\}$, such that $u \neq v$. If $\{u, v\} \in E$, then we say that vertices u, v are *adjacent*. For $\gamma = (V, E)$ we sometimes write $\gamma = (V(\gamma), E(\gamma))$.

A *path of length* k from a vertex u to a vertex v in $\gamma = (V, E)$ is a sequence $(x_0, x_1, \ldots, x_k)$ of vertices such that $u = x_0, v = x_k$, and $\{x_j, x_{j+1}\} \in E$ for all $0 \leq j < k$. The length of the path is the number of edges in the path (k in the previous case).

If there is a path from a vertex u to a vertex v, then we say that v is *reachable* from u in γ, and we denote it as $u \hookrightarrow_\gamma v$. Also, we say that u and v are *connected* in γ.

An undirected graph is *connected* if every pair of different vertices are connected.

A path is *simple* if all edges and all vertices on the path, except possibly the first and the last vertices, are distinct. A *cycle* is a simple path of length at least 1 which begins and ends with the same vertex. Note that in an undirected graph, a cycle must be of length at least 3. An undirected graph with no cycles is said to be *acyclic*.

A (*free*) *tree* is a connected, acyclic, and undirected graph. A *rooted tree* is a tree with one of its vertices distinguished from the other ones. The distinguished vertex is called the *root* of the tree.

Let γ be a rooted tree with root r. Given a vertex u, any vertex v different from u on the unique path from r to u is called a *proper ancestor* of u. If v is a proper ancestor of u, then we say that u is a *proper descendant* of v. Any vertex v with no proper descendants is called a *leaf* of the rooted tree.

For every vertex u different from the root we denote by $f(u)$ (the *father* of u) the only proper ancestor of u such that $\{f(u), u\} \in E$. The father of the root is undefined. For any vertex u we denote by $Ch(u)$ (the *children* of u) the set of proper descendants of u such that their father is u.

In a rooted tree $\gamma = (V, E)$ with root r we can define a binary relation $\bar{E}$ on V as follows: $(x, y) \in \bar{E}$ if and only if x is a proper ancestor of y. This binary relation establishes a "direction" over the edges of the rooted tree.

Definition 3.5.1. *A* membrane structure *is a rooted tree, where the nodes are called* membranes, *the root is called* skin, *and the leaves are called* elementary membranes.

In this formalization the labels of the membranes are the vertices of the graph.

In the definition of a membrane system we will use the notion of a *cell*, which is a membrane structure with a multiset of objects associated with

every membrane. Let us denote by $\mathbf{M}(O)$ the family of multisets over a set O. The multisets will be written as vectors, representing the multiplicities of objects from O, in a given ordering of these objects; in this notation, the inclusion among two multisets $\boldsymbol{m}_1, \boldsymbol{m}_2$ is written as an inequality among vectors, $\boldsymbol{m}_1 \leq \boldsymbol{m}_2$.

Definition 3.5.2. *A* cell *over an alphabet (of objects) O is a pair (μ, M), where μ is a membrane structure, and M is an application from $V(\mu)$ to* $\mathbf{M}(O)$.

The *degree* of a cell $C = (\mu, M)$, denoted by $|C|$, is $card(V(\mu))$, that is the number of membranes from the membrane structure.

Definition 3.5.3. *Let (μ, M) be a cell over an alphabet O. An* (evolution) rule *over O is a triple $r = (\boldsymbol{d}_r, \boldsymbol{v}_r, \delta_r)$, where*

1. *$\boldsymbol{d}_r$ is a multiset over O.*
2. *$\boldsymbol{v}_r$ is a function with domain $V(\mu) \cup \{here, out\}$ and range contained in $\mathbf{M}(O)$, where $here, out \notin V(\mu)$.*
3. *$\delta_r \in \{\neg\delta, \delta\}$, with $\neg\delta, \delta \notin O$.*

Finally, to give the syntax of a membrane system we need to assign a set of rules to every membrane of the structure.

Definition 3.5.4. *Let $C = (\mu, M)$ be a cell over an alphabet O. A* collection *R of evolution* rules *associated with C is a function R with the domain $V(\mu)$ such that for any membrane $x \in V(\mu)$ we have $R(x) = \{r_1^x, \ldots, r_{s_x}^x\}$ for some $s_x \geq 0$, where $r_1^x, \ldots, r_{s_x}^x$ are evolution rules over O. Instead of $R(x)$ we sometimes write R_x.*

Definition 3.5.5. *Let $C = (\mu, M)$ be a cell and let R be a collection of evolution rules associated with C. A* priority relation *over R is a function ρ with the domain $V(\mu)$ such that for every membrane $x \in V(\mu)$, $\rho(x)$ (also denoted by ρ_x) is a strict partial order over R_x (possibly empty).*

We interpret the fact that $(r', r) \in \rho_x$ in the sense that the rule r' has a higher priority than the rule r.

Definition 3.5.6. *A symbol–object P system is a 4-tuple $\Pi = (O, C_0, \mathcal{R}, i_o)$, where:*

- *O is a non-empty finite set of objects.*
- *$C_0 = (\mu_0, M_0)$ is a cell over O.*
- *$\mathcal{R}$ is an ordered pair (R, ρ) where R is a collection of evolution rules associated with C_0, and ρ is a priority relation over R.*
- *i_o is a node of the rooted tree μ_0 (the output membrane of Π).*

We now proceed to give the formal definition of the functioning of a membrane system – one can say that this is the "semantic" of a the system. To this end, we need the concepts of configuration, transition, and output.

Definition 3.5.7. *A* configuration *of a system* $\Pi = (O, C_0, \mathcal{R}, i_o)$, *with* $C_0 = (\mu_0, M_0)$, *is a cell* $C = (\mu, M)$ *over* O, *where* $V(\mu) \subseteq V(\mu_0)$, *and* μ *has the same root as* μ_0. *The configuration* $C_0 = (\mu_0, M_0)$ *will be called the* initial configuration of Π.

Now, we have to define the application of a rule in a given region.

Definition 3.5.8. *Let* $C = (\mu, M)$ *be a configuration of a system* $\Pi = (O, C_0, \mathcal{R}, i_0)$, *with* $C_0 = (\mu_0, M_0)$, *and let* $x \in V(\mu_0)$ *be a membrane of* Π. *We say that the (evolution) rule* $r = (\boldsymbol{d}_r, \boldsymbol{v}_r, \delta_r) \in R_x$ *is* semi-applicable *to* C *if:*

- *The membrane associated with node* x *exists in* C; *that is* $x \in V(\mu)$.
- *The dissolution is not allowed in the root node; that is, if* x *is the root node of* μ, *then* $\delta_r = \neg\delta$.
- *The membrane* x *contains all the objects necessary to apply the rule, that is* $\boldsymbol{d}_r \leq M(x)$.
- *The nodes where the rule tries to send objects by means of* in_y *are children of* x; *that is for all* $y \in V(\mu)$, *if* $\boldsymbol{v}_r(y) \neq \mathbf{0}$, *then* $y \in Ch(x)$.

The rule r *is* applicable *to* C *if it is semi-applicable to* C *and there is no semi-applicable rules in* R_x *with a higher priority; that is, there is no* $r' \in R_x$ *such that* $(r', r) \in \rho_x$ *and* r' *is semi-applicable to* C.

We define the *index of applicability of* r *in* C *over the node* x, and we denote it by $N_{Ap}(r, C, x)$, as the maximum number of times the rule r can be applied to C in the node x:

$$N_{Ap}(r, C, x) = \max\{n \mid n \otimes \boldsymbol{d}_r \leq M(x)\}.$$

Of course, an applicable rule may not be effectively applied at a given moment, because of the non-determinism.

Definition 3.5.9. *Let* $C = (\mu, M)$ *be a configuration of a system* Π. *We say that* $\boldsymbol{p}_x = (p_{x,1}, \ldots, p_{x,s_x})$ *is an* applicability vector *over* $x \in V(\mu)$ *for* C, *and we write* $\boldsymbol{p}_x \in \mathbf{Ap}(x, C)$, *if:*

1. *The node is still alive; that is, if* $\boldsymbol{p}_x \neq \mathbf{0}$, *then* $x \in V(\mu)$.
2. *Every rule can be applied as many times as the vector* $\boldsymbol{p}$ *indicates; that is, for all* j *such that* $1 \leq j \leq s_x$ *we have* $p_{x,j} \leq N_{Ap}(r_j^x, C, x)$.
3. *All rules can be applied simultaneously, that is* $\sum_{j=1}^{s_x} p_{x,j} \otimes \boldsymbol{d}_{r_j^x} \leq M(x)$.
4. *The vector* $\boldsymbol{p}_x$ *is maximal; that is, there is no* $\boldsymbol{v} = (q_1, \ldots, q_{s_x})$ *such that* $\boldsymbol{p}_x < \boldsymbol{v}$ *and* $\boldsymbol{v}$ *satisfies the previous conditions 1, 2, 3.*

The rules are simultaneously applied in all membranes, hence the notion of applicability vector should be extended as follows.

Definition 3.5.10. *Let $C = (\mu, M)$ be a configuration of a system $\Pi = (O, C_0, \mathcal{R}, i_0)$ with $C_0 = (\mu_0, M_0)$ such that $V(\mu) = \{x_1, \ldots, x_m\}$. A set $Q = \{\boldsymbol{p}_{x_1}, \ldots, \boldsymbol{p}_{x_m}\}$ of applicability vectors, $\boldsymbol{p}_{x_i} \in \mathbf{Ap}(x, C)$, $1 \leq i \leq m$, is called an* applicability matrix *over C, and we write $Q \in \mathbf{M_{Ap}}(C)$.*

In order to determine how a system Π evolves by means of an applicability matrix Q, we must formalize our intuitive understanding of the way that a system *computes*. To determine a successor configuration of a given configuration $C = (\mu, M)$, we proceed in two stages. In the first stage all the rules are executed without considering the dissolving action. In the second stage we remove the dissolved membranes/nodes, distribute their contents to the region which is immediately external to them, and redefine the connections in the resulting rooted tree.

The first stage produces the configuration $C'' = (\mu, M'')$, where

$$M''(x) = M(x) - \sum_{j=1}^{s_x} p_{x,j} \otimes \boldsymbol{d}_{r_j^x} + \sum_{j=1}^{s_x} p_{x,j} \otimes \boldsymbol{v}_{r_j^x}(here)$$
$$+ \sum_{j=1}^{s_{f(x)}} p_{f(x),j} \otimes \boldsymbol{v}_{r_j^{f(x)}}(x) + \sum_{y \in Ch(x)} \sum_{j=1}^{s_y} p_{y,j} \otimes \boldsymbol{v}_{r_j^y}(out).$$

To get the second stage (redistribution of the contents of the dissolved membranes) we need to characterize the nodes to be dissolved by the execution of an applicability matrix.

Definition 3.5.11. *If $C = (\mu, M)$ is a configuration of a system Π, and $Q \in \mathbf{M_{Ap}}(C)$ is an applicability matrix over C, then we define*

$$\Delta(Q, C) = \{x \in V(\mu) \mid \textit{there is } 1 \leq j \leq s_x \textit{ with } p_{x,j} \neq 0 \textit{ and } \delta_{r_j^x} = \delta\}.$$

That is, $\Delta(Q, C)$ is the set of nodes in the rooted tree determined by the configuration C that have to be removed after the application of the matrix Q. In one step of a transition several membranes/nodes can be dissolved.

Definition 3.5.12. *Let $C = (\mu, M)$ be a configuration of a system Π and let $Q \in \mathbf{M_{Ap}}(C)$ be an applicability matrix over C. For each node $x \in V(\mu)$, we define the* donors *of x for C in the application of Q as follows:*

$$Don(x, Q, C) = \begin{cases} \emptyset, & \textit{if } x \in \Delta(Q, C), \\ \{y \in \Delta(Q, C) \mid x \hookrightarrow_\mu y, & \\ \textit{and, if } x \hookrightarrow_\mu z \hookrightarrow_\mu y, & \\ \textit{then } z \in \Delta(P, C)\}, & \textit{otherwise.} \end{cases}$$

That is, if a membrane x is not dissolved, then a membrane y is a donor for x if y is dissolved in the application of Q, and every membrane which is a proper descendant of x and a proper ancestor of y is also dissolved.

Now, we can define the execution of an applicability matrix over a given configuration.

Definition 3.5.13. *Let $C = (\mu, M)$ be a configuration of a system Π and let $Q \in \mathbf{M_{Ap}}(C)$ be an applicability matrix over C. We define the* execution *of Q over C, denoted $Q(C)$, as the configuration $C' = (\mu', M')$, where:*

- *μ' is the rooted tree obtained from μ as follows:*
  - *$V(\mu') = V(\mu) - \Delta(Q, C)$.*
  - *If $x, y \in V(\mu')$, then*

$$(x,y) \in \bar{E}(\mu') \text{ iff } \begin{array}{l} \text{there are } x_0, \ldots, x_n \in V(\mu) \\ \text{such that } x_1, \ldots, x_{n-1} \in \Delta(Q,C), \\ x_0 = x, x_n = y, \text{ and for all } 0 \leq i < n \\ \text{we have } (x_i, x_{i+1}) \in \bar{E}(\mu). \end{array}$$

- $$M'(x) = \begin{cases} M''(x) \cup \bigcup_{y \in Don(x,Q,C)} M''(y), & \text{if } x \notin \Delta(Q,C), \\ \emptyset, & \text{otherwise.} \end{cases}$$

That is, μ' is the rooted tree obtained from μ by erasing all the nodes dissolved by the application of matrix Q, and restoring the connections: if a node y is dissolved, then the node and all the edges adjacent to it are erased, and every edge between the first non-dissolved ancestor of y and every first non-dissolved descendant of y are added. If a node is dissolved, then its content is removed; if not, then we add to its content the content of every donor node.

We are now ready to define the transition from a configuration of a system to another configuration.

Definition 3.5.14. *We say that a configuration C_1 of a system Π yields a configuration C_2 in a* one-step transition, *denoted $C_1 \Rightarrow_\Pi C_2$, if there exists a non-null applicability matrix Q over C_1 such that $Q(C_1) = C_2$.*

A configuration C yields a configuration C' in $k \geq 1$ transition steps, if there are configurations $C_1, \ldots, C_{k+1}$ such that $C_1 = C, C_{k+1} = C'$, and for all $1 \leq i \leq k$ we have $C_i \Rightarrow_\Pi C_{i+1}$. We also say that $C_1 \Longrightarrow_\Pi C_1$ holds in 0 transition steps. Finally, we say that a configuration C yields a configuration C', denoted $C \Longrightarrow^*_\Pi C'$, if C yields C' in k transition steps, for some $k \geq 0$.

Definition 3.5.15. *The* computation tree of a system Π, *denoted* $\mathbf{Comp}(\Pi)$, *is a rooted labelled maximal tree defined as follows: the root of the tree is the initial configuration, C_0, of Π. The children of a node are the configurations that follow in a one-step transition. Nodes and edges are labelled by configurations and applicability matrices, respectively, in such a way that two labelled nodes C, C' are adjacent in* $\mathbf{Comp}(\Pi)$ *by means of an edge labelled with Q if and only if Q is a non-null applicability matrix from $\mathbf{M_{Ap}}(C)$ and $C' = Q(C)$.*

The maximal branches of $\mathbf{Comp}(\Pi)$ *are called* computations *of Π. We say that a computation of Π* halts *if it is a finite branch. The configurations C such that $\mathbf{M_{Ap}}(C)$ contains only the null matrix are called* halting configurations, *and we denote their set by $Halt(\Pi)$.*

Definition 3.5.16. *We say that a computation* $C_0 \Rightarrow_\Pi C_1 \Rightarrow_\Pi \ldots \Rightarrow_\Pi C_n$ *of a system* $\Pi = (O, C_0, \mathcal{R}, i_o)$ *is* successful *if it halts and* i_o *is a leaf of the rooted tree* μ_n*, where* $C_n = (\mu_n, M_n)$*. We denote by* $S(\Pi)$ *the set of all successful computations of* Π*.*

Definition 3.5.17. *If* $\mathbf{M_{Ap}}(C)$ *is unitary for every configuration* C *of a* P *system, then we say that the system is* deterministic*, otherwise we say that the system is* non-deterministic*.*

In the previous sections, the membrane systems were used in order to compute sets of numbers, but we have mentioned that we can see these devices as generating multisets (the objects present in the output membrane in the halting configurations, identified by their multiplicities), hence we give here definitions also for this general case.

Definition 3.5.18. *Let* $\Pi = (O, C_0, \mathcal{R}, i_o)$ *be a membrane system. The Parikh set generated by* Π*, denoted by* $Ps(\Pi)$*, is*

$$P_s(\Pi) = \{M_C(i_o) \mid C \in S(\Pi) \text{ and } C = (\mu_C, M_C)\}.$$

The set of natural numbers generated by Π*, denoted by* $N(\Pi)$*, is*

$$N(\Pi) = \{|M_C(i_o)| \mid C \in S(\Pi) \text{ and } C = (\mu_C, M_C)\}.$$

3.6 Further Extensions

Actually, what we are going to consider in this section are not only further features to be added to membrane systems with symbol–objects, but also some restrictions. As usual, the motivation comes from two directions: we try to stay as close as possible to the biological reality, especially to the ways of transferring chemical compounds through membranes, and, at the same time, to keep the models as simple/elegant as possible from a mathematical point of view.

3.6.1 Weak Target Commands

In the first subsections we deal with changes in the way of controlling the communication between the regions of a system.

For instance, the target commands *here, out* look quite natural, but this is not necessarily the case with in_j. Specifying the name of a membrane where an object should go is reminiscent of the fact that the protein channels are selective, and they can choose the ions and molecules which can be transported through a membrane, but nevertheless a command in_j looks very programming languages oriented (it resembles "*goto i*" instructions). Similarly, the

priority relation among rules corresponds to the observation that the rules describe chemical reactions and the chemical reactions have different degrees of reactivity, some of which develop faster than others; in given circumstances some reactions are more probable than others, but the similarity with ordered grammars is also visible. Finally, the bi-stable catalysts can be related to the fact that there are enzymes which "get tired" after enhancing a reaction, change their state, and return to the active state after a while, but this is also a rather automata-theoretic point of view.

However, in the universality results from the previous sections, either the priorities or the bi-stable catalysts (or both, in the case of non-synchronized systems) were essentially used, a fact that is not encouraging from the point of view of the bio-realism of these results. This is a good reason to look for classes of membrane systems which do not involve these features, but compensate the expected loss of power by other features, possibly/hopefully more realistic.

Fortunately, in the proofs of the previous universality results we do not essentially use the addressing in_j, because the number of membranes was always one or two, hence no address was necessary when sending an object to an inner membrane; the inner membrane was always unique. Thus, the weaker addressing in, without any label, is sufficient. We can use such an addressing also in the general case, when it is possible to have several membranes placed directly inside a given membrane, with the meaning that the respective object should go to any of these membranes, non-deterministically choosing it.

More specifically, together with $(a, here)$ and (b, out), in the rules associated with a membrane i we can use (c, in); when applying such a rule, the object c should be transferred to any membrane placed directly inside membrane i, choosing it non-deterministically; if membrane i is elementary, then a rule $u \to v$ such that v contains at least a pair (c, in) cannot be applied.

The weak addressing by *here, out, in* will be indicated in the notation of families of sets of numbers computed by P systems by replacing *tar* by *in*.

An intermediate possibility, more specific than *in* and less specific than in_j, is to associate both with objects and membranes (*electrical*) *charges* of three sorts: $+, -, 0$ (positive, negative, neutral). The charges of membranes are given in the initial configuration, and they are not changed during computations, but the charge of objects is given explicitly by the evolution rules that introduce them. For example, the rule $a \to b^+ d^- c$ introduces one copy of b positively charged, one copy of d negatively charged, and a copy of c which is neutral (non-charged; the indication 0 is not written). A charged object will immediately go to one of the directly adjacent lower membranes of the opposite polarization, and the neutral objects remain in the same region or will exit it, depending on whether they are associated with the command *here* or the command *out*, respectively (therefore, the neutral objects will have associated indications *here, out* – usually *here* is omitted). After a charged object crosses a membrane, it becomes neutral.

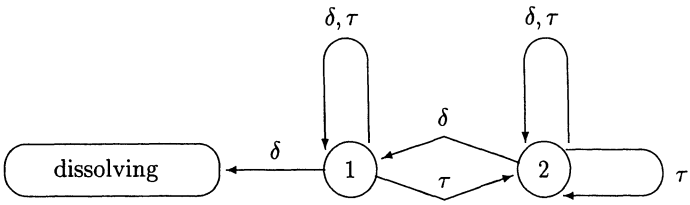


Fig. 3.5. The effect of actions δ, τ

3.6.2 Controlling the Permeability of Membranes

Another way of controlling the passage of objects through membranes is to control the *permeability* of membranes, remembering the fact that the permeability of bio-membranes can be variable. This control of permeability can be implemented in membrane systems by the use of an action opposite to δ, denoted by τ. This action is associated with rules in the same way as δ: we allow rules of the form $u \to v\tau$ (u and v are strings, representing multisets of objects, the objects from v having associated target commands of various types); therefore, we cannot have both δ and τ associated with the same rule. The effect of using such a rule is an increase in the "thickness" of a membrane, possibly making it impermeable, in the following sense.

Let us assume that initially all membranes have thickness 1. If a rule is used in a membrane of thickness 1 and it introduces the symbol τ, then the thickness of this membrane becomes 2. A membrane of thickness 2 does not become thicker through the use of other rules which introduce the symbol τ; however, no object can pass through it (this means that rules which ask for an object to be sent through a membrane of thickness 2 are not allowed to be used, neither in the membrane of thickness 2, if the rule introduces objects (a, out), nor in the upper region, in the case of rules which send objects by commands in_j, or in, or by means of electrical charges to the impermeable membrane). If a rule which introduces the symbol δ is used within a membrane of thickness 1, then the membrane is dissolved; if the membrane had thickness 2, then it returns to thickness 1 (hence at the next step the membrane is again permeable). If at the same step one uses rules which introduce both δ and τ in the same membrane, then the membrane does not change its thickness. The cumulative effect of the actions δ and τ is illustrated by the diagram in Figure 3.5 (the two circles indicate the thickness of membranes).

As expected, and as we will see in Theorem 3.6.1, the actions δ and τ together provide a very convenient/powerful programming device to regulate the passage of objects through membranes and the use of rules.

When the action of increasing membrane thickness is used, we add τ to the notation of families of sets of numbers, thus obtaining families of the form $NOP_m(cat, in, \delta, \tau)$.

Theorem 3.6.1. $NOP_4(cat, in, \delta, \tau) = NRE$.

Proof. As usual, we will prove only the non-trivial inclusion, that is $NRE \subseteq NOP_4(cat, in, \delta, \tau)$.

Let us consider a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the strong binary normal form, given in the standard notation, but with a different labelling. Specifically, all matrices $(X \to Y, A \to x)$, hence without rules to be used in the appearance checking mode, are labelled with $m_{2i}, 1 \leq i \leq k$. The matrices of the form $(X \to Y, B^{(j)} \to \#)$, hence with the second rule to be used in the appearance checking mode, are labelled with m_i with $i \in lab_j$, for $j = 1, 2$, such that lab_1, lab_2, and $\{2, 4, \ldots, 2k\}$ are mutually disjoint sets. We denote by lab the union $lab_1 \cup lab_2$.

Each matrix of the form $(X \to \lambda, A \to x), X \in N_1, A \in N_2, x \in a^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. We continue to label the obtained matrix in the same way as the original one.

We construct the system

$$\Pi = (O, K, \mu, w_1, w_2, w_3, w_4, R_1, R_2, R_3, R_4, 3)$$

with the following components:

$$\begin{aligned}
O &= N_1 \cup N_2 \cup \{\bar{X}, X', X_{2i}, X'_{2i}, (X_{2i}, j) \mid X \in N_1, 1 \leq i, j \leq k\} \\
&\quad \cup \{X_i \mid X \in N_1, i \in lab\} \\
&\quad \cup \{\bar{A}, A', A'', A_{2i}, A'_{2i}, A''_{2i}, (A_{2i}, j) \mid A \in N_2, 1 \leq i, j \leq k\} \\
&\quad \cup \{a, c, f, f', \bar{f}, E, E', Z, Z', Z'', Z_2, Z_3, Z_4, \#\}, \\
K &= \{c\}, \\
\mu &= [_1[_2\ ]_2[_3\ ]_3[_4\ ]_4]_1, \\
w_1 &= \bar{X}_{init}\bar{A}_{init}Ec, \\
w_2 &= Z_2, \\
w_3 &= cZ_3, \\
w_4 &= cZ_4,
\end{aligned}$$

and with the sets R_1, R_2, R_3, R_4 containing the following rules (the membrane with label 1 will be used for simulating matrices $m_{2i}, 1 \leq i \leq k$, while membranes with labels 3, 4 will simulate the matrices with labels in lab_1, lab_2, respectively; in the rules below, for a string $x \in (N_2 \cup \{a\})^*$, x'' represents the string obtained by replacing all non-terminal symbols $A \in N_2$ which appear in x with A'' and leaving unchanged all occurrences of the terminal symbol a; as a shorthand, we will write (x, α) instead of $(b_1, \alpha) \ldots (b_s, \alpha)$, for $\alpha \in \{in, out\}$ and $x = b_1 \ldots b_s$, with $b_i \in N_2 \cup \{a\}, 1 \leq i \leq s$):

$R_1 : E \to (E, in),$
$\bar{X} \to X',$
$X' \to X,$
$X'' \to \bar{X},$ for all $X \in N_1,$
$\bar{D} \to D',$
$D' \to D,$
$D'' \to \bar{D},$ for all $D \in N_2,$
$X \to (X_{2i}, in),$
$cA \to c(A_{2i}, in),$ for all $m_{2i} : (X \to Y, A \to x), 1 \leq i \leq k,$
$X \to (X_i, in),$ for all $m_i : (X \to Y, B^{(j)} \to \#), j = 1, 2, i \in lab,$
$D \to (D, in),$
$D \to \#,$ for all $D \in N_2,$
$X \to \#,$ for all $X \in N_1,$
$E' \to E(Z_2 Z_3 Z_4, in),$
$Z \to \#,$
$\# \to \#,$
$a \to \#,$
$a \to (a, in).$

$R_2 : Z_2 \to Z\tau,$
$E \to \delta,$
$X_{2i} \to (X_{2i}, 1)\tau,$ for all $X \in N_1, 1 \leq i \leq k,$
$A_{2i} \to (A_{2i}, 1),$ for all $A \in N_2, 1 \leq i \leq k,$
$(X_{2i}, j) \to (X_{2i}, j+1),$
$(A_{2i}, j) \to (A_{2i}, j+1),$ for all $1 \leq j < 2k,$
$(X_{2i}, 2i) \to Y'\tau,$
$(A_{2i}, 2i) \to A'_{2i}\delta,$
$Y' \to Y,$
$Y \to (Y''E', out),$
$Y \to \#,$
$A'_{2i} \to A''_{2i}\delta,$
$A''_{2i} \to (x'', out),$ for all $m_{2i} : (X \to Y, A \to x), X \in N_1,$
$Y \in N_1 \cup \{f\},$
$A \in N_2, x \in (N_2 \cup \{a\})^*,$
$D \to D',$
$D' \to (D'', out),$ for all $D \in N_2,$
$X_i \to \#,$ for all $X \in N_1$ and $i \in lab,$

$Z_3 \to \#,$
$Z_4 \to \#,$
$\# \to \#,$
$a \to (a, out).$

$R_3 : Z_3 \to Z'\tau,$
$cZ' \to cZ'',$
$Z'' \to \delta,$
$Z' \to \lambda,$
$cE \to c\delta,$
$X_i \to Y,$
$D \to D',$ for all $D \in N_2, D \neq B^{(1)},$
$Y \to (\bar{Y}E, out)\tau,$ for all $m_i : (X \to Y, B^{(1)} \to \#),$
$B^{(1)} \to \#,$
$X_i \to \#,$ for all $X \in N_1$ and $i \in (lab \cup \{2, 4, \ldots, 2k\}) - lab_1,$
$D_i \to \#,$ for all $D \in N_2$ and $i \in (lab \cup \{2, 4, \ldots, 2k\}) - lab_1,$
$Z_2 \to \#,$
$Z_4 \to \#,$
$\# \to \#.$

$R_4 : Z_4 \to Z'\tau,$
$cZ' \to cZ'',$
$Z'' \to Z\delta,$
$Z' \to Z,$
$cE \to c\delta,$
$X_i \to Y,$
$D \to D',$ for all $D \in N_2, D \neq B^{(2)},$
$D' \to (D'', out),$
$Y \to (Y''E', out),$ for all $m_i : (X \to Y, B^{(2)} \to \#),$
$B^{(2)} \to \#,$
$X_i \to \#,$ for all $X \in N_1$ and $i \in (lab \cup \{2, 4, \ldots, 2k\}) - lab_2,$
$D_i \to \#,$ for all $D \in N_2$ and $i \in (lab \cup \{2, 4, \ldots, 2k\}) - lab_2,$
$Z_2 \to \#,$
$Z_3 \to \#,$
$\# \to \#,$
$a \to (a, out).$

Besides the fact that we handle all matrices which contain rules of the form $B^{(j)} \to \#, j = 1, 2,$ in only one membrane, another important technique

used in this construction, and which has not appeared in the proofs of the universality results from Section 3.4, is the way of simulating the matrices without appearance checking rules by the interplay of membranes with labels 1 and 2 (specifically, by means of the subscripts i and the parameters j in non-terminal symbols of the form (X_{2i}, j) and (A_{2i}, j), for $1 \leq i \leq k, 1 \leq j \leq 2k$). Note the important fact that the labels of matrices m_{2i} have even values for the subscripts.

Let us examine in some detail the way the previous system works.

We start with membranes of thickness 1, but at the first step all membranes $2, 3, 4$ get the thickness 2 by using the rules $Z_2 \to Z\tau$, $Z_3 \to Z'\tau$, $Z_4 \to Z'\tau$, respectively (it is important to note that the object Z' is not introduced in the output membrane, that with label 3). At the same time, in the skin membrane we use the rule $E \to (E, in)$, which sends the object E, non-deterministically, to one of the inner membranes, as well as the rules $\bar{X} \to X', \bar{D} \to D'$, for all $X \in N_1, D \in N_2$.

At the next step, the primed objects from the skin membrane lose their primes and the object E "opens" the membrane where it has been sent. (If E arrives in one of membranes 3, 4 and we use here the rule $cZ' \to cZ''$ instead of $cE \to c\delta$, then the membrane will be opened at the next step and the computation will never halt: the rules of the form $\alpha \to (\beta, in)$ cannot be used, and hence we have to use rules of the form $X \to \#, D \to \#$.) That is, exactly one of the membranes with labels 2, 3, 4 has again the thickness 1, hence it is permeable. Therefore, at the next step, when rules of the form $X \to (X_{2i}, in), X \to (X_l, in), cA \to c(A_{2j}, in), D \to (D, in)$ should be used, all objects X_{2i}, X_l, A_{2j}, D will arrive in the same membrane (if we use a rule of the form $X \to \#$ or $D \to \#$, then the trap object $\#$ is introduced and the computation will never halt).

Let us assume that in the skin membrane we use a rule $X \to (X_{2i}, in)$ for some $1 \leq i \leq k$. If the object X_2 arrives in one of membranes 3, 4, then the trap object is introduced. Thus, we can continue only in the case when X_{2i} is sent to membrane 2. In the first step, the rule $X_{2i} \to (X_{2i}, 1)\tau$ will make this membrane of thickness 2. Simultaneously, all symbols $D \in N_2$ become D'. If no object A_{2j} (actually, only one, because the catalyst is present in only one copy) is present here, then the membrane will remain impermeable, eventually X_{2i} will reach $(X_{2i}, 2i)$, and then Y' and Y will be produced, which will not be able to leave the membrane, hence the rule $Y \to \#$ will be used, and the computation will never finish. If an object A_{2j} is also present, then we have to examine three possibilities:

Case 1: $i < j$. In the first step we get the objects $(X_{2i}, 1), (A_{2j}, 1)$ (and the symbol τ is also introduced), then we grow the second component of these objects until reaching $(X_{2i}, 2i)$ and $(A_{2j}, 2i)$. At the next step we get Y' and $(A_{2j}, 2i+1)$, and we introduce one further symbol τ, which, however, changes nothing, the membrane remains of thickness 2.

Now, because $i < j$ we have $2j \geq 2i + 2 > 2i + 1$, hence at the next step we produce Y and $(A_{2j}, 2i+2)$, and then we have to use the rule $Y \to \#$ (the rule $Y \to (Y''E', out)$ cannot be used, because the membrane has thickness 2), and thus the computation will never halt.

Case 2: $i > j$. After the first step, when we get $(X_{2i}, 1), (A_{2j}, 1)$ (and the membrane is made thicker), we continue until reaching the objects $(X_{2i}, 2j)$, $(A_{2j}, 2j)$. At the next step we produce $(X_{2i}, 2j + 1)$ and A'_{2j}, while also introducing δ, which returns the membrane to thickness 1. At the next step we again introduce the symbol δ, by using the rule $A'_{2j} \to A''_{2j}\delta$, the membrane is dissolved, the object Z arrives in the skin region, and the trap object is again introduced.

Case 3: $i = j$. This time we arrive at the objects $(X_{2i}, 2i), (A_{2i}, 2i)$ at the same time, with the membrane of thickness 2. At the next step we use the rules $(X_{2i}, 2i) \to Y'\tau$ and $(A_{2i}, 2i) \to A'_{2i}\delta$, which do not change the membrane thickness. At the next step we use $Y' \to Y$ and $A'_{2i} \to A''_{2i}\delta$, which return the membrane to thickness 1. This makes possible the use of the rules $Y \to (Y''E', out)$ and $A''_{2i} \to (x'', out)$. In this way, the objects Y'', E' are sent to the skin membrane, together with the objects of x, with the non-terminal symbols double primed. Simultaneously, the rules $D' \to (D'', out)$, for all $D \in N_2$, can be used. In the skin membrane, Y'' introduces $\bar{Y}$, all $D'', D \in N_2$, become $\bar{D}$, and E'' introduces E and sends the objects Z_2, Z_3, Z_4 to the lower membranes. Each Z_i must go to the corresponding membrane $i = 2, 3, 4$, otherwise the trap symbol is introduced. During the work in membrane 2, in membranes 3, 4 we use the rules $Z_i \to Z'\delta$, $cZ' \to cZ''$, $Z'' \to \delta$, $Z'' \to Z\delta$, respectively, and the membrane is again of thickness 1. If instead of $cZ' \to cZ''$ we use $Z' \to \lambda$ in membrane 3 or $Z' \to Z$ is membrane 4, then the membrane remains of thickness 2 forever, and it is never used again. (In particular, if this is the case with membrane 3 and any copy of a is introduced in the skin membrane, then the computation will never stop: either a goes forever through membranes 2 and 4, or the rule $a \to \#$ is used in the skin region.)

We return to a configuration the same as that we started with: all membranes have thickness 1, the objects in the skin membrane are barred, and the object E is again available. Note that in this way the use of the matrix $m_{2i} : (X \to Y, A \to x)$ has been correctly simulated.

Assume now that at the third step considered above we do not use a rule $X \to (X_{2i}, in)$ for $1 \leq i \leq k$, but a rule $X \to (X_i, in)$ for some $i \in lab$, hence corresponding to a matrix $m_i : (X \to Y, B^{(j)} \to \#)$, for some $j = 1, 2$. If the object X_i arrives in the membrane with label 2 (that is, the object E has been sent to this membrane) or if an object A_j has been produced by a rule $cA \to c(A_j, in)$ and it arrives in one of membranes 3, 4, then the trap object is introduced. Therefore, the only correct continuation is to have one of membranes 3, 4 of thickness 1 and to send here both the object X_i and all objects from N_2. If the open membrane is not that with the label $2 + j$, for

$i \in lab_j$, then again the trap object is introduced and the computation never stops.

Thus, assume that in membrane $2+j$ we have both X_i, for $i \in lab_j$, as well as all objects of N_2. At the next step we use the rule $X_i \to Y$, for the corresponding matrix m_i, as well as all rules $D \to D'$, for all $D \in N_2 - \{B^{(j)}\}$. If also $B^{(j)}$ is present, then the rule $B^{(j)} \to \#$ must be used and the trap object is introduced. If $B^{(j)}$ is not present, then we continue with $Y \to (Y''E', out)$ and $D' \to (D'', out)$, hence we return all objects, double primed, to the skin membrane, with X replaced by Y. In this way, the matrix m_i is correctly simulated. (Note that, because we have opened the membrane j by means of the rule $cE \to c\delta$, the rule $cZ' \to cZ''$ cannot be used at the same time, and hence $Z' \to \lambda$ or $Z' \to Z$ should be used; thus, the membrane is not "illegally" dissolved by the rules $Z'' \to \delta$ or $Z'' \to Z\delta$.)

The process can be iterated. All copies of the terminal object a are directed to membrane 3: in membranes 2 and 4 we have the rule $a \to (a, out)$ while in the skin membrane we have the rule $a \to (a, in)$; no rule from membrane 3 involves the object a. Thus, the computation can stop only if all copies of a are "stored" in membrane 3 and no copy of a is present outside this membrane.

As long as at least one non-terminal symbol of G is present, the rules $X \to \#, D \to \#$ from the skin membrane can be used. If we reach a stage when only an object $X \in N_1$ is present in the system, but no object from N_2 is present, then either this object X enters membrane 2, and it will produce the trap object, or it enters one of membranes 3, 4 (if a matrix $m_i : (X \to Y,\ B^{(j)} \to \#)$ exists), it will be changed into Y, and it exits; the process continues. If no such matrix exists, then either $X \to \#$ is used, or the object enters membrane 2. Similarly, if no object from N_1 is present but some objects from N_2 still exist in the skin membrane, then either such objects enter a membrane 3, 4, or they enter membrane 2 unchanged, and are sent out unchanged, or they enter membrane 2 with a subscript, and then the trap object is produced. Consequently, if we have objects either only from N_1 or only from N_2, then the computation continues forever. The only way to stop is to use a matrix $m_t : (X \to f, A \to x)$, with a terminal string x, and to obtain in this way a configuration where no non-terminal symbol of G is still present in the system. The object f will remain unchanged in the skin membrane, all copies of a are sent to membrane 3, and the computation halts. Thus, we can simulate in Π all terminal derivations in G and only such derivations are simulated by halting computations. That is, $length(L(G)) = N(\Pi)$, which concludes the proof. □

3.6.3 Communication Controlled by Concentration

As we have mentioned in Section 2.1, in living cells a significant part of transmembrane transport of chemicals takes place in a passive way, because of the difference in *concentration* between two adjacent compartments of a cell or

between the cell and its environment. Let us also incorporate this way of communication in our systems.

At first sight, this can be done in a very simple way: in the set O of objects we consider a subset, $O_c \subseteq O$, and whenever such objects are introduced, they are immediately redistributed among all regions of the system in such a way that all regions will have the same number of occurrences of each symbol from O_c; a difference of one occurrence is allowed when the total number of occurrences is not divisible by the number of regions. We do not count the environment here, hence for a system of degree m we take into consideration only the m internal regions.

Note that we handle each object from O_c separately and we do not count these objects (and then redistribute them) as being indistinguishable.

A problem appears when the number of copies of an object to be redistributed is not a multiple of m, where m is the number of membranes in the system. In such a case, the "remainder objects" are redistributed according to the following "efficiency principle": we move objects at the lowest distance (crossing the smallest number of membranes). For instance, if we have $m+1$ copies of an object from O_c in a region, then all regions will get one copy of the object, while the region where the objects were produced will remain with two copies. If we started with $m+2$ copies of the object, then each region will get one copy, one further copy will remain in the region where the object was processed, and one further copy will be sent to one of the neighbouring regions, non-deterministically chosen. If only two copies of an object are introduced in a given region, then one of them remains in the same region, and the other one will go to any of the adjacent regions, non-deterministically chosen.

How the redistribution of objects takes place in practical circumstances is not considered here, but we assume that the redistribution is done instantaneously, always correctly, and at the level of the whole system in one step, by a magical mechanism which is not explicit in our model.

Because in the proof of Theorem 3.6.2 below we will always introduce new objects in only one region at a time, the above-mentioned principle will be sufficient to govern the communication, so we ignore the more complex situations which can appear.

It is important to note that in this framework, no indication *here, in, out* is necessary. The objects in O_c will go *in* or *out*, according to their concentration, and the objects in $O - O_c$ remain in the region where they are introduced. Of course, no catalyst can appear in O_c, as the catalysts are not allowed to change their regions.

In the notation of families of sets of numbers computed by P systems of a given type, we write *con* instead of the target mode in order to indicate that we use systems with the communication controlled by concentration.

Theorem 3.6.2. $NOP_2(2cat, con) = NRE$.

Proof. Let $G = (N, \{a\}, S, M, F)$ be a matrix grammar in the Z-binary normal form, in the standard notation. We construct the system

$$\begin{aligned}
\Pi &= (O, K, [_1[_2\ ]_2]_1, w_1, \lambda, R_1, \emptyset, 2),\\
O &= N \cup O_c \cup K \cup \{E, \#\} \cup \{X', X'', X''', X^{iv} \mid X \in N_1\},\\
O_c &= \{a\},\\
K &= \{c_i, \bar{c}_i \mid 1 \le i \le k\} \cup \{d_i, \bar{d}_i \mid k+1 \le i \le n\}\\
w_1 &= X_{init} A_{init} E c_1 c_2 \ldots c_n d_{k+1} d_2 \ldots d_n,
\end{aligned}$$

with the set R_1 containing the following rules.

1. For each matrix $m_i : (X \to Y, A \to x), 1 \le i \le k$, we consider the rules:
   $c_i X \to \bar{c}_i Y'$,
   $Y' \to Y$,
   $\bar{c}_i A \to c_i x'$, where x' is obtained by replacing each occurrence of a from x by (a, in_2),
   $\bar{c}_i E \to c_i \#$.
2. For each matrix $m_i : (X \to Y, A \to \#), k+1 \le i \le n$, we consider the rules:
   $X \to X''$,
   $d_i X'' \to \bar{d}_i Y'''$,
   $Y''' \to Y^{iv}$,
   $\bar{d}_i A \to d_i \#$,
   $\bar{d}_i Y^{iv} \to d_i Y$.
3. We also add the rule $\# \to \#$.

At any moment, only one symbol X from N_1 is present (and for each $X \in N_1$ there is a rule $X \to Y$ in a matrix of G). If we start by using a rule $c_i X \to \bar{c}_i Y'$, then at the next step we have also to simulate the second rule of the matrix $m_i : (X \to Y, A \to \#)$, otherwise the rule $\bar{c}_i E \to c_i \#$ will be used and the trap object $\#$ is introduced. Each copy of a is directly introduced into membrane 2.

If the symbol X becomes X'', then at the next step we have to use a rule $d_i X'' \to \bar{d}_i Y''', k+1 \le i \le n$, and the simulation of a matrix $m_i : (X \to Y, A \to \#)$ is started. At the next step, Y''' becomes Y^{iv} and $\bar{d}_i$ must wait, or, if the symbol A is present, then the trap object is introduced. If A is not present, then at the next step $\bar{d}_i$ returns to d_i with the help of Y^{iv}.

The process can be iterated and it cannot stop before introducing the object Z, because Z appears only when no non-terminal from N_2 is present in the string generated by G.

Therefore, we obtain $length(L(G)) = N(\Pi)$, and this ends the proof. □

Note that in the proof of Theorem 3.6.2 we have not used priorities or the control of the membrane thickness – but we have used the powerful feature of bi-stable catalysts. It is an *open problem* of interest to investigate systems

with the communication based on concentration (possibly also taking into consideration the environment, and distributing copies of objects to $m+1$ regions, m inside the system and one outside) for the case of the usual catalysts, possibly also using priorities, actions δ, τ, etc. (Problem Q4.)

3.6.4 Creating Rules During the Computation

We continue by considering two other bio-inspired extensions.

The first one concerns the possibility to create new evolution rules during the computation. The biochemistry from a living cell is to a great extent managed by the proteins bound to membranes; the proteins are chemicals which, in principle, can be processed as any other chemicals, and in particular, can be consumed during reactions. Let us suppose that when using a rule in a membrane system, the rule is "consumed"; this means that we have to take into consideration the multiplicity of rules, to work with multisets of rules in the same way as we have worked with multisets of objects. In order to perform infinite computations, we have to provide the possibility to increase also the number of available rules.

Formally, we consider a set R_{pos} of all *possible rules*, injectively labelled by the elements of a set *lab*; the rules associated with a membrane i will be of the form $r : u \to v/z$, for $r \in lab, z \in lab^*$. The meaning of $r : u \to v/z$ is that after using this rule, a copy of r is consumed, and all rules indicated by z become available in membrane i. When specifying the initial configuration of a system, for each membrane we indicate a multiset of rules available at the beginning of each computation: $\Pi = (O, \mu, w_1, \ldots, w_m, lab, R_{pos}, R_1, \ldots, R_m, i_o)$ (alphabet, membrane structure, multisets of objects placed in the m regions of μ, labels of possible rules, the form of possible rules, multisets of rules present in the m regions of μ, output membrane, respectively; a set of catalysts can also be considered). The computations proceed as usual, in the non-deterministic maximally parallel manner, with a result associated with a halting computation in the form of the number of objects present in the output membrane.

Consuming/creating rules can be seen as a way of controlling the next rule to be used, as in programmed grammars. Because the maximal parallelism, together with the halting condition, has a certain connection with the appearance checking mechanism from regulated grammars, it is not very surprising that in this case also we obtain the computational completeness, even without using any other feature, such as a priority, actions δ, τ, and even without using catalysts: the fact that we work with multisets of rules ensures the possibility to control/inhibit the parallelism when necessary. This is a rather interesting property of systems with rule creation – the following result is up to now the first universality theorem where non-cooperative rules (hence no catalyst) are used.

In the notation of families of sets of numbers computed by P systems with the rule creation possibility we add the indication *rule* to the other used features.

Theorem 3.6.3. $NOP_1(ncoo, rule) = NRE$.

Proof. Consider a matrix grammar with appearance checking $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, in the standard notation, and construct the system

$$
\begin{aligned}
\Pi &= (O, [_1\ ]_1, X_{init}A_{init}, lab, R_{pos}, R_1, 1),\\
O &= N \cup \{X' \mid X \in N_1\} \cup \{a, D, \#\},\\
lab &= \{r_i, r_i', r_i'' \mid 1 \le i \le k\}\\
&\quad \cup \{r_j, r_{j,1}, r_{j,2}, r_{j,3}, r_{j,4} \mid k+1 \le j \le n\} \cup \{r_{n+1}, r_v\},\\
R_1 &= r_1 r_2 \ldots r_k r_{k+1} \ldots r_n r_{n+1} r_v,\\
R_{pos} &= \{r_i : X \to X'/r_i',\\
&\qquad r_i' : A \to x/r_i'',\\
&\qquad r_i'' : X' \to Y/r_i \mid \text{for } m_i : (X \to Y, A \to x), 1 \le i \le k\}\\
&\quad \cup \{r_{n+1} : Z \to \lambda/\lambda\}\\
&\quad \cup \{r_j : X \to X'D/r_{j,1}r_{j,2},\\
&\qquad r_{j,1} : B \to \#/\lambda,\\
&\qquad r_{j,2} : D \to B/r_{j,3},\\
&\qquad r_{j,3} : \# \to (\#, out)/r_{j,4},\\
&\qquad r_{j,4} : X' \to Y/r_j \mid \text{for } m_j : (X \to Y, B \to \#), k+1 \le j \le n\}\\
&\quad \cup \{r_v : \# \to \#\#/r_v r_v\}.
\end{aligned}
$$

The simulation of matrices $m_i, 1 \le i \le k$, proceeds in a way which is rather similar to the work of a programmed grammar – hence similar to the work of a matrix grammar – so we invite the reader to examine this case.

The case of matrices with rules used in the appearance checking mode is trickier. Assume that we have a multiset of objects described by a string Xw (in the presence of all rules $r_1, \ldots, r_n, r_v$) and that we use a rule $r_j : X \to X'D/r_{j,1}r_{j,2}$. It replaces X with the symbols X' and D, disappears, and introduces the new rules $r_{j,1} : B \to \#/\lambda$ and $r_{j,2} : D \to B/r_{j,3}$. Thus, if the multiset contains the object B, then the rule $r_{j,1} : B \to \#/\lambda$ must be used. It introduces the trap object $\#$ which makes possible the use of the "virus" rule $r_v : \# \to \#\#/r_v r_v$. This last rule will exponentially multiply both the object $\#$ and itself, so the computation will continue forever and no result is obtained.

If the rule $r_{j,1} : B \to \#/\lambda$ cannot be used, then it remains in the system. At the next step, the rule $r_{j,2} : D \to B/r_{j,3}$ previously introduced will replace D with B, also introducing the rule $r_{j,3} : \# \to (\#, out)/r_{j,4}$. This

rule cannot be used immediately, because we have no copy of # available, but the rule $r_{j,1} : B \to \#/\lambda$ can now be applied to the single occurrence of B introduced above. We get one copy of #, hence at the next step we can use either the rule $r_{j,3} : \# \to (\#, out)/r_{j,4}$ or the virus rule. In the latter case, the computation will never finish, but we can avoid this by using the first rule, which sends the object # out of the system. In this way, the virus rule cannot be applied, it remains "silent" in the system. By using the rule $r_{j,3} : \# \to (\#, out)/r_{j,4}$ we have also introduced the rule $r_{j,4} : X' \to Y/r_j$, which returns the configuration to a form the same as that we started with: the symbol from N_1 is again unprimed, and all rules $r_1, \ldots, r_{n+1}$ are again present in one copy. This corresponds to a correct simulation of the matrix $m_j : (X \to Y, B \to \#)$. The process can be iterated.

Note that the rule $r_{j,3} : \# \to (\#, out)/r_{j,4}$ is always introduced one step before introducing the object #, hence we can always avoid the use of the virus rule, but if the rule $r_{j,1} : B \to \#/\lambda$ is applicable, then it will lead to the activation of the virus rule, because it is used before introducing the rule $r_{j,3} : \# \to (\#, out)/r_{j,4}$.

When the rule r_{n+1} is applicable, hence the derivation in G is terminal, no object different from a is present in the system. Thus, we obtain the equality $N(\Pi) = length(L(G))$, which concludes the proof. □

3.6.5 Using Promoters/Inhibitors

We have said several times that biochemistry is cooperative and that many reactions in the cell are enzyme enhanced. Cooperation is similar to context-sensitivity, which is a (too) powerful feature, so we try to avoid it as much as possible in our abstract, symbolic framework. On the other hand, the idea of enzymes and that of catalysts were captured by the catalytic rules. What about *promoters*: that is, the case of reactions which are possible (or only more probable) in the presence of certain chemicals which are not exactly catalysts, and which can separately evolve in parallel with the chemicals whose evolution they support? A similar question concerns *inhibitors*, chemicals which forbid certain reactions (and which do not have any resemblance to the catalysts, as they have a negative influence on a reaction).

There are several possibilities of formalizing these ideas. The first one is to consider the promoters/inhibitors at the level of each rule; that is, to consider rules of the forms $u \to v|_a$ and $u \to v|_{\neg a}$, where a is an object. Assume that in region i we have the multiset of objects w. The rule $u \to v|_a$ is applied in region i only if the object a belongs to the multiset $w - u$; the rule $u \to v|_{\neg a}$ is applied only if the object a does not belong to the multiset $w - u$. That is, the permitting/forbidding objects should appear/not appear in the region where the rule is applied, but without taking into consideration the multiset of objects to which the rule is applied. In parallel with the conditional rules there may exist rules of the standard form, $u \to v$, which are freely applied, as usual.

A promoting rule $u \to v|_a$ is not the same as considering the rule $au \to av$, because the same object a may promote several rules (in particular, several applications of the same rule), and, moreover, the object a may evolve at the same time, by means of other rules (perhaps promoted by objects used by the rule $u \to v|_a$).

We settle here the case of using promoters – and, as expected, we get computational universality (for the case of catalytic rules). The case of inhibitors remains *open* (Problem Q5).

The use of promoters at the level of separate rules is indicated by adding *proR* in the notation of families of sets of numbers computed by P systems.

Theorem 3.6.4. $NOP_2(cat, in, proR) = NRE$.

Proof. Take a matrix grammar $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, in the standard notation, and construct the system

$$\Pi = (O, \{c\}, [_1[_2\ ]_2]_1, X_{init}A_{init}Ed_0, \lambda, R_1, \emptyset, 2),$$

with the alphabet

$$O = N \cup \{d_i, d'_i \mid 1 \leq i \leq n+1\} \cup \{a, c, d_0, E, \#\},$$

and the following rules:

$cd_0 \to cd_i$, for all $1 \leq i \leq n+1$,
$d_i \to \lambda$,
$cX \to cYd'_i|_{d_i}$,
$d'_i \to \lambda$,
$cA \to cxd_0|_{d'_i}$, for all $m_i : (X \to Y, A \to x)$, $1 \leq i \leq k$,
$d_i \to \lambda$,
$cX \to cYd'_i|_{d_i}$,
$cd'_i \to cd_0$,
$A \to \#|_{d'_i}$, for all $m_i : (X \to Y, A \to \#)$, $k+1 \leq i \leq n$,
$d_{n+1} \to \lambda$,
$cZ \to cd'_{n+1}|_{d_{n+1}}$,
$d'_{n+1} \to \lambda$,
$E \to \lambda|_{d'_{n+1}}$,
$cX \to c\#$, for all $X \in N_1$,
$cE \to c\#$,
$\# \to \#$,
$a \to (a, in)$.

The reader can easily check the equality $length(L(G)) = N(\Pi)$, by making use of the following observations: at each step, the catalyst c should be "kept busy", otherwise it will introduce the trap object $\#$ by means of the rule $cE \to c\#$, or of a rule $cX \to c\#$ for some $X \in N_1$; this observation is very useful, both in ensuring that each matrix without an appearance checking

rule is completely simulated (if the rule $cA \to cxd_0|_{d_i'}$ cannot be used, then the catalyst can be involved in a trap rule as above), and in ensuring that the rules $cd_0 \to cd_i$ do not choose, because of their non-determinism, a "wrong" matrix to simulate (if we cannot apply the right rule $cX \to cYd_i'|_{d_i}$ promoted by d_i, then again the catalyst is free for rules which introduce the trap object); note also that the appearance checking rules $A \to \#$ are applicable only in the presence of the corresponding object d_i', but if they cannot be used, we may skip them, exactly as in a matrix grammar; in the meantime, the catalyst is used by the rule $cd_i' \to cd_0$. □

In the previous proof we need membrane 2 only because we cannot remove the catalyst, hence, if we use only one membrane, then we cannot generate the number zero. Thus, if we ignore this number, one membrane suffices. Otherwise stated, if we denote by $N'RE$ the family of recursively enumerable sets of natural non-null numbers, then we have $NOP_1(cat, in, proR) = N'RE$.

Another possibility is to consider the promoters/inhibitors at the level of sets of rules: that is, to have a list of sets of rules associated with each membrane, each set having associated a promoting object (or a multiset of objects); at each step we may use only rules from those sets which are enabled by the permitting/forbidding condition with respect to the current multiset. We introduce this variant of control in the general case of promoters/inhibitors in the form of multisets of objects, because this case gives again a rather interesting result: the universality can be obtained *without catalysts*. After Theorem 3.6.3, this is the second time when a characterization of NRE is obtained by non-cooperative rules, and this time the result is quite unexpected at first sight, because of the way the parallelism is controlled. In Theorem 3.6.3 we worked with multisets of rules, hence the parallelism was restricted by the multiplicity of available rules, but here the rules are not counted, they can be used arbitrarily many times in the same step. In general, the control of the parallelism is ensured by catalysts: if a catalyst c appears in only one copy, then at each step we can process (at most) one copy of an object a by a rule $ca \to cv$. However, in what follows there is some price to pay in order to obtain universality without catalysts: we allow the promoting objects to be processed by the same rules which the objects themselves promote. This is different from the case considered above of rule promoting (we believe that the definition can be modified without changing the results).

A *membrane system with promoters/inhibitors at the level of sets of rules* is a construct

$$\Pi = (O, \mu, w_1, \ldots, w_m, (R_{1,1}, p_{1,1}), \ldots, (R_{1,k_1}, p_{1,k_1}), \ldots, \\ (R_{m,1}, p_{m,1}), \ldots, (R_{m,k_m}, p_{m,k_m}), i_o),$$

where O, μ, $w_1, \ldots, w_m$, and i_o are, as usual, the alphabet of objects, the membrane structure (of degree m), the multisets of objects initially present

in the system, and the output membrane, respectively, while $R_{i,1}, \ldots, R_{i,k_i}$, $k_i \geq 0$, are finite sets of evolution rules present in region i of the system and $p_{i,j}$ are strings over O, called *promoters*, $1 \leq i \leq m, 1 \leq j \leq k_i$. The rules are of the form $a \to v$, for $a \in O, v \in (O \times \{here, out, in\})^*$. We do not have catalysts.

The rules from a set $R_{i,j}$ can be used in region i only if the multiset represented by $p_{i,j}$ is present in that region. Of course, if several multisets $p_{i,j_1}, \ldots, p_{i,j_s}$ are present in region i, then rules from all sets $R_{i,j_1}, \ldots, R_{i,j_s}$ may be used in region i.

The maximum of $k_i, 1 \leq i \leq m$, is called the *promoter diversity* of the system.

We denote by $NOP_m(ncoo, in, proS_p)$ the family of sets of numbers $N(\Pi)$ computed by P systems of degree at most $m \geq 1$, with non-cooperating rules, with the target indications *here, out, in*, and using promoters at the level of sets of rules, with the promoter diversity at most $p \geq 1$.

The following results indicate a trade-off between the degree and the promoter diversity of the systems. We start with the case of the minimal promoter diversity – in exchange, using the largest number of membranes used in any universality result from the previous sections: six!

Theorem 3.6.5. $NOP_6(ncoo, in, proS_2) = NRE$.

Proof. Consider a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the strong binary normal form, in the standard presentation. Each matrix of the form $(X \to \lambda, A \to x), X \in N_1, A \in N_2, x \in a^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. We continue to label the obtained matrix in the same way as the original one.

We construct the system (of degree 6 and promoter diversity 2)

$$\begin{aligned}\Pi = (&O, \mu, w_1, \ldots, w_6, (R_{1,1}, p_{1,1}), (\emptyset, \emptyset), (R_{2,1}, p_{2,1}), (R_{2,2}, p_{2,2}), \ldots,\\ &(R_{6,1}, p_{6,1}), (R_{6,2}, p_{6,2}), 4),\end{aligned}$$

with the following components:

$$\begin{aligned}O &= N_1 \cup N_2 \cup \{X', X_i, X_i', (X_i, j) \mid X \in N_1 \cup \{f\}, 1 \leq i, j \leq k\}\\ &\quad \cup \{X_i \mid X \in N_1, i \in lab_1 \cup lab_2\}\\ &\quad \cup \{A', A_i, A_i', (A_i, j) \mid A \in N_2, 1 \leq i, j \leq k\}\\ &\quad \cup \{a, d, d', f, E, E', F, \#\},\\ \mu &= [_1[_2[_3\ ]_3]_2[_4\ ]_4[_5\ ]_5[_6\ ]_6]_1,\\ w_1 &= X_{init} A_{init} E,\\ w_i &= \lambda, \text{ for } i = 2, \ldots, 6,\end{aligned}$$

and with the sets $R_{i,j}$ containing the following rules (we will give at the same time the promoters $p_{i,j}$; the membranes with labels 2, 3 will be used for simulating matrices $m_i, 1 \leq i \leq k$, while membranes with labels 5, 6 will

simulate the matrices with labels in lab_1, lab_2, respectively; membrane 4 is used to control the parallelism of the system and for collecting the output; in the rules below, for a string $x = a_1a_2$, with $a_1, a_2 \in N_2 \cup \{a\}$, we write (x, out) instead of $(a_1, out)(a_2, out)$):

$$
\begin{aligned}
R_{1,1} \ : \ & F \to (F, in), \\
& X \to (Y_i, in)(E, in)F, \text{ and} \\
& A \to (A_i, in)(d, in)F, \text{ for all } m_i : (X \to Y, A \to x) \in M, \\
& \quad 1 \le i \le k, Y \in N_1 \cup \{f\}, \\
& A \to (A, in)F, \text{ for all } A \in N_2, \\
& X \to (Y_i, in)(E, in), \text{ for all } m_i : (X \to Y, B^{(j)} \to \#), \\
& \quad i \in lab_j, j = 1, 2, \\
& a \to (a, in), \\
& \# \to \#; \\
p_{1,1} = \ & \lambda; \\
R_{2,1} \ : \ & Y_i \to (Y_i, 1), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& A_i \to (A_i, 1), \text{ for all } A \in N_2, 1 \le i \le k, \\
& E \to \lambda, \\
& A \to (A', in), \text{ for all } A \in N_2; \\
p_{2,1} = \ & E; \\
R_{2,2} \ : \ & Y_i \to \#, \text{ for all } Y \in N_1 \cup \{f\}, i \in lab_0, \\
& \quad \text{or } Y \in N_1, i \in lab_1 \cup lab_2, \\
& A \to \#, \text{ for all } A \in N_2, \\
& A_i \to \#, \text{ for all } A \in N_2, i \in lab_0 \cup lab_1 \cup lab_2, \\
& d \to \#, \\
& F \to \#, \\
& \# \to \#, \\
& (Y_i, j) \to (Y_i, j+1), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le j < i \le k-1, \\
& (A_i, j) \to (A_i, j+1), \text{ for all } A \in N_2, 1 \le j < i \le k-1, \\
& (Y_i, i) \to (Y_i, in)(d, in), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& (A_i, i) \to (A_i, in)(d', in), \text{ for all } A \in N_2, 1 \le i \le k, \\
& Y_i' \to (Y, out), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& A_i' \to (x, out), \text{ for all } m_i : (X \to Y, A \to x), 1 \le i \le k, \\
& A' \to (A, out), \text{ for all } A \in N_2, \\
& a \to (a, out); \\
p_{2,2} = \ & \lambda; \\
R_{3,1} \ : \ & Y_i \to (Y_i', out), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k,
\end{aligned}
$$

$$
\begin{aligned}
& A_i \to (A'_i, out), \text{ for all } A \in N_2, 1 \le i \le k, \\
& A' \to (A', out), \text{ for all } A \in N_2, \\
& d \to \lambda, \\
& d' \to \lambda; \\
p_{3,1} = {} & dd'; \\
R_{3,2} : {} & Y_i \to \#, \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& A_i \to \#, \text{ for all } A \in N_2, 1 \le i \le k, \\
& A \to \#, \text{ for all } A \in N_2, \\
& \# \to \#; \\
p_{3,2} = {} & \lambda; \\
R_{4,1} : {} & d \to (\#, out); \\
p_{4,1} = {} & dd; \\
R_{4,2} : {} & F \to \lambda, \\
& d \to \lambda, \\
& Y_i \to \#, \text{ for all } Y \in N_1 \cup \{f\}, i \in lab_0, \\
& \quad \text{or } Y \in N_1, i \in lab_1 \cup lab_2, \\
& A_i \to \#, \text{ for all } A \in N_2, 1 \le i \le k, \\
& A \to \#, \text{ for all } A \in N_2; \\
p_{4,2} = {} & F; \\
R_{4+j,1} : {} & Y_i \to \#, \text{ for all } Y \in N_1 \cup \{f\}, i \notin lab_j, \\
& A_i \to \#, \text{ for all } A \in N_2, i \in lab_0, \\
& Y_i \to (Y, out), \text{ for all } Y \in N_1, i \in lab_j, \\
& A \to (A, out), \text{ for all } A \in N_2 - \{B^{(j)}\}, \\
& B^{(j)} \to \#, \\
& E \to \lambda; \\
p_{4+j,1} = {} & E; \\
R_{4+j,2} : {} & Y_i \to \#, \text{ for all } X \in N_1 \cup \{f\}, i \in lab_0, \\
& \quad \text{or } Y \in N_1, i \in lab_1 \cup lab_2, \\
& A_i \to \#, \text{ for all } A \in N_1, 1 \le i \le k, \\
& A \to \#, \text{ for all } A \in N_2, \\
& d \to \#, \\
& F \to \#, \\
& \# \to \#, \\
& a \to (a, out); \\
& p_{4+j,2}] = \lambda, \text{ for } j = 1, 2.
\end{aligned}
$$

We will now discuss in some detail the way this system works, because it is typical for membrane systems with promoters associated with sets of rules, and because many "tricks" used in this construction will also appear in the proofs of the subsequent theorems.

We start with the objects X_{init}, A_{init} in the skin membrane. Let us assume that we are in a configuration when we have here a multiset described by a string Xw, with $X \in N_1$ and $w \in (N_2 \cup \{a\})^*$. The terminal object a is sent to lower membranes by the rule $a \to (a, in)$, but it comes back from membranes 2, 5, 6, so eventually has to reach the output membrane, that with label 4.

Assume that the unique symbol from N_1 was replaced by Y_i, for some $1 \leq i \leq k$, with the intention to simulate the matrix $m_i : (X \to Y, A \to x)$. At the same time we introduce the objects E and F (the latter object remains in the skin membrane for one step). We have three cases: none, exactly one, or more than one copy of a symbol D from N_2 was processed by a rule $D \to (D_j, in)(d, in)F$. All other symbols $B \in N_2$ should evolve by rules $B \to (B, in)F$.

The object E is meant to *keep together* all the objects, and this is done in the following way. If the promoter E is not present, then the sets of rules $R_{2,1}, R_{5,1}, R_{6,1}$ cannot be used, hence the rules from the sets $R_{2,2}, R_{5,2}, R_{6,2}$ must be used instead (note that these rules can always be used, because of the empty promoter, always present by default in any multiset) and they introduce the trap object #, which evolves forever by means of the rule $\# \to \#$. If at the first step some objects Y_i, A_i, A arrive in membrane 4, then they will wait one step, but when F arrives here, they will introduce the trap object, too. On the other hand, the object F must arrive in membrane 4, because this is the only place where it does not introduce the trap object (rules $F \to \#$ are present in all sets $R_{2,2}, R_{5,2}, R_{6,2}$). Thus, we conclude that *all objects of the forms Y_i, A_i, A should be together with E.* This is very important in what follows.

Now, let us see what happens with the object d. It leads immediately to # in membranes 2, 5, 6, so it must arrive in membrane 4. If there are at least two copies of d, then the promoter dd allows the use of the rule $d \to \#$ (and this rule must be used, because the object F is not yet here). If there is only one copy of d, then the rule $d \to \#$ cannot be used, the object d waits unchanged until F arrives here, then it is removed. At the same time, F is removed. In short, *we ensure in this way that at most one symbol $A \in N_2$ was replaced by A_j, for some $1 \leq j \leq k$.*

If the object Y_i and (at most one copy of) an object A_j arrive in one of membranes 5, 6, then again the computation will never finish (these objects introduce # also in the presence of E), so we have to be in membrane 2 with all objects. Because E is present, we can avoid the use of rules which introduce the trap object. We replace Y_i by $(Y_i, 1)$, A_j by $(A_j, 1)$, and we send to membrane 3 all other objects from N_2, primed; at the same time, the

object E is removed. For a while, these objects will do nothing in membrane 3, because no applicable rule is enabled here.

In membrane 2 we start to check whether or not $i = j$ (and whether or not we have *at least* one object of the form A_j; up to now we only know that there is *at most* such an object). To this end we use the rules $(Y_i, s) \to (Y_i, s+1)$ and $(A_j, s) \to (A_j, s+1)$, repeatedly, until obtaining an object (Y_i, i) or (A_j, j). Such objects produce $Y_i d$ and $A_j d'$, respectively, which are immediately sent to membrane 3. If $i \neq j$, then only one d is sent to membrane 3, primed or not, together with Y_i or A_j; this means that the set $R_{3,1}$ is not enabled, hence the corresponding rule $Y_i \to \#, A_j \to \#$ from $R_{3,2}$ must be used. If we have both d and d' in membrane 3, that is we have $i = j$, and indeed *we had* a copy of A_i in membrane 2, then we use the rules of $R_{3,1}$. We send to membrane 2 primed versions of all objects, while the two objects d, d' are erased. From membrane 2, the primed object Y_i' sends to the skin membrane the object Y, A_i' sends the multiset x, and at the same time all non-terminals B' from membrane 2 are sent to the skin membrane in the non-primed version.

In this way, we have correctly simulated the matrix $m_i : (X \to Y, A \to x)$ (one and exactly one rule $A \to (A_i, in)(d, in)F$ has been used, with the correct subscript i) and we have returned to a configuration the same as the one we started with. The process can be iterated.

Assume now that we start by using a rule $X \to (Y_i, in)(E, in)$, for some $i \in lab_j, j = 1, 2$. As we have seen above, all objects should arrive in the same membrane, one of membranes 2, 5, 6. In membrane 2, we cannot use any other rule than $Y_i \to \#$ (we have rules $Y_i \to (Y_i, 1)$ only for $i \in lab_0$). Similarly, if we are in a membrane $4+j$ such that $i \notin lab_j$, then again the trap object is introduced. So, let us assume that we are in the right membrane $4 + j$. If any object $A_i, i \in lab_0$, exists there, then again the trap object is produced.

Assume that we are in membrane $4 + j$, with Y_i, for some $i \in lab_j$, and with all objects from N_2, unchanged, plus the object E. Because E is here, the set $R_{4+j,1}$ is enabled, Y_i sends out the object Y, while all other symbols go out, with the exception of $B^{(j)}$, which introduces the trap object. Therefore, the computation continues correctly (that is, it will eventually halt) if and only if the matrix $m_i : (X \to Y, B^{(j)} \to \#)$ was correctly simulated.

We return to a configuration similar to the starting one, so we can repeat this process.

If we reach a configuration where either only one object from N_1 or only objects from N_2 are still present, then the computation will never finish. If an object from N_1 arrives in membrane 2, because we cannot produce two copies of d here, we will produce the trap object in membrane 3, as described above. If an object Y_i arrives in the right membrane $4 + j$ (that is, with $i \in lab_j$), then it will exit unchanged, hence the process continues. The same happens with the objects from N_2 – except for $B^{(j)}$ which introduces the trap object in membrane $4 + j$. If only objects from N_2 are present, at most one of them

will be processed by a rule $A \to (A_j, in)(d, in)F$ from the skin membrane, so we will not be able to produce d, d' in membrane 2, hence the trap object is introduced in membrane 3 (or in membranes 4, 5, 6, if A_i arrives in one of them). If only objects from N_2 are present and all of them are processed in membranes 1 and 2 by rules $A \to (A, in)F$, then in membrane 3 the trap object will be introduced, irrespective of the region where the objects A will be sent. So, we have to erase the object from N_1 in the same step when the last object from N_2 is removed (hence we have to simulate a terminal matrix of G at the last step of the simulation). In this way, the computation stops and we know that in membrane 4 we have exactly the copies of a generated by G.

Consequently, $N(\Pi) = length(L(G))$, and this concludes the proof. □

At the cost of increasing the promoter diversity, we can decrease the number of membranes. We first give two intermediate universality results without proofs (the proofs are based on standard constructions, and use the techniques explained in the proof of Theorem 3.6.5), then we will give the extreme one: one membrane suffices (providing that we have "sufficient" promoter diversity).

Theorem 3.6.6. $NOP_4(ncoo, in, proS_3) = NOP_3(ncoo, in, proS_4) = NRE$.

Theorem 3.6.7. $NOP_1(ncoo, proS_6) = NRE$.

Proof. We start again from a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the strong binary normal form, with the terminal matrices $(X \to \lambda, A \to x)$ replaced with $(X \to f, A \to x)$, for a new symbol f, and we construct the system (of degree 1 and promoter diversity 6)

$$\begin{aligned}\Pi = (O, [_1\ ]_1, X_{init}A_{init}, (R_{1,1}, \lambda), (R_{1,2}, ee), (R_{1,3}, dd),\\ (R_{1,4}, g), (R_{1,5}, d_1), (R_{1,6}, d_2), 1),\end{aligned}$$

with the alphabet of objects

$$\begin{aligned}O = N_1 \cup N_2 \cup \{X_i, X_i', (X_i, j) \mid X \in N_1 \cup \{f\}, 1 \le i, j \le k\}\\ \cup \{X_i \mid X \in N_1, i \in lab_1 \cup lab_2\}\\ \cup \{A', A_i, A_i', (A_i, j) \mid A \in N_2, 1 \le i, j \le k\}\\ \cup \{a, d, d_1, d_2, e, f, g, \#\},\end{aligned}$$

and with the sets $R_{1,j}, 1 \le j \le 6$, containing the following rules:

$$\begin{aligned}R_{1,1} : X \to Y_i e, &\text{ for all } m_i : (X \to Y, A \to x), 1 \le i \le k,\\ X \to Y_i d_j, &\text{ for all } i \in lab_j, j = 1, 2,\\ A \to A_i ed, &\text{ for all } m_i : (X \to Y, A \to x), 1 \le i \le k,\\ A \to A',&\end{aligned}$$

$$
\begin{aligned}
& A' \to A', \text{ for all } A \in N_2, \\
& e \to \#, \\
& \# \to \#, \\
& Y_i \to (Y_i, 1), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& A_i \to (A_i, 1), \text{ for all } A \in N_2, 1 \le i \le k, \\
& (Y_i, j) \to (Y_i, j+1), \text{ for all } Y \in N_1 \cup \{f\}, 1 \le j < i \le k-1, \\
& (A_i, j) \to (A_i, j+1), \text{ for all } A \in N_2, 1 \le j < i \le k-1, \\
& (Y_i, i) \to Y_i' eg, \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& (A_i, i) \to A_i' e, \text{ for all } A \in N_2, 1 \le i \le k, \\
& Y_i' \to Y, \text{ for all } Y \in N_1 \cup \{f\}, 1 \le i \le k, \\
& A_i' \to x, \text{ for all } m_i : (X \to Y, A \to x), 1 \le i \le k, \\
& g \to \lambda, \\
& f \to \lambda; \\
R_{1,2} : & e \to \lambda; \\
R_{1,3} : & d \to \#; \\
R_{1,4} : & d \to \lambda, \\
& A' \to A, \text{ for all } A \in N_2; \\
R_{1,4+j} : & Y_i \to \#, \text{ for all } Y \in N_1 \cup \{f\}, i \notin lab_j, \\
& Y_i \to Y, \text{ for all } Y \in N_1, i \in lab_j, \\
& A' \to A, \text{ for all } A \in N_2 - \{B^{(j)}\}, \\
& {B^{(j)}}' \to \#, \\
& d_j \to \lambda, \text{ for } j = 1, 2.
\end{aligned}
$$

Starting from a multiset described by a string Xw with $w \in (N_2 \cup \{a\})^*$, we proceed as follows. The object X is replaced either by $Y_i e$, for some $i \in lab_0$, or by $Y_i d_j$, for some $i \in lab_j$, for $j = 1, 2$.

Assume that we are in the first case. If no rule $A \to A_j ed$ is used, then $R_{1,2}$ is not enabled, hence the rule $e \to \#$ from $R_{1,1}$ must be used. If two or more rules of the form $A \to A_j ed$ are used, then the two copies of d will enable the rule $d \to \#$. In both cases, the computation will never stop. If exactly one rule of the form $A \to A_j ed$ is used, then the unique copy of d will wait (until the object g is introduced), while the two copies of e will be erased. The two objects Y_i, A_j will follow a procedure (as in the proof of Theorem 3.6.5) of checking whether or not $i = j$. In the negative case again only one copy of e will be introduced, by one rule from the couple $(Y_i, i) \to Y_i' eg$, $(A_i, i) \to A_i' e$, and the trap object is produced. If $i = j$, then we can erase the two objects e; because we have also introduced the object g, we can also erase d, and return all non-terminals $A', A \in N_2$, to their unprimed variants. At the same time Y_i' introduces Y and A_i' introduces x, which means that the matrix $m_i : (X \to Y, A \to x)$ is correctly simulated.

If we have started by introducing the object $Y_i d_j$, for some $i \in lab_j, j = 1, 2$, then we simulate the matrix $m_i : (X \to Y, B^{(j)} \to \#)$. If an object $A \in N_2$ has been replaced by $A_i ed$, as above, then we get the trap object from the object e. Therefore, all non-terminals from N_2 should be primed. Then, d_j enables the set $R_{1,4+j}$ which checks whether or not $B^{(j)}$ is present. The matrix m_i is correctly simulated.

In both cases, we return to a configuration of the same form as that we started with (note that no object e, d, d_1, d_2, g remains in the system, all of them are removed by rules of the form $\alpha \to \lambda$). The process can be iterated. When no non-terminal is present, the computation halts after using the rule $f \to \lambda$. If we reach a configuration where only one object from N_1 and no object from N_2 is present, then the computation will continue forever, because the object e is produced. Assume that we have only objects from N_2. If we replace any object $A \in N_2$ by A', then the rule $A' \to A'$ will be used forever. If we use a rule $A \to A_i ed$, then again we will continue forever, because e will introduce the object $\#$. The halting computations in Π precisely correspond to terminal derivations in G, and $N(\Pi) = length(L(G))$. □

We close this section by mentioning, without a proof, a result about the case of inhibitors, where multisets $q_{i,j}$ of objects are associated with sets $R_{i,j}$ of rules. A set $R_{i,j}$ of rules can be used in membrane i only if the associated multiset $q_{i,j}$ is not included in the multiset of objects present in membrane i. The corresponding families of sets of numbers, computed by systems with the inhibitor diversity at most q, are denoted by $NOP_m(ncoo, in, inhS_q)$.

Theorem 3.6.8. $NOP_3(ncoo, in, inhS_6) = NRE$.

These results should be compared with those on the power of random context grammars with only permitting or only forbidding context conditions (where we do not get the universality – see Subsection 2.3.7), and with those on semi-conditional grammars (which are universal). Here, the promoting objects are used both to control the correct sequencing of rules and to inhibit the parallelism. Together with the compartmentalization and the maximal parallelism, both promoters and inhibitors prove to be rather powerful, leading to universality.

3.7 Systems with External Output

The way of defining the result of a computation by means of an elementary output membrane is mathematically natural, and may sometimes be even unavoidable – such a case will be that in the next chapter – but it is somewhat unsuitable from a "practical" point of view: in order to get the result of a computation we have to look inside the computer. In our case, the "computer" is a cell, hence reading the result can be rather difficult indeed. A more natural

possibility is to stay outside the system and to take into consideration the objects expelled through the skin membrane during the computation.

This way of defining the result of a computation has several mathematical advantages, too.

For instance, in some proofs from the previous sections we have used the output membrane only in order to squeeze out the result, separating the objects we want from some additional objects (for instance, catalysts) which we do not want. Such an output membrane can be avoided if we count the result outside the system: we just send out only the objects we want to observe. This easy modification works for Theorems 3.4.3, 3.4.6, 3.6.2, 3.6.4, hence in all these cases we get characterizations of NRE by systems with only one membrane, providing that the result of a halting computation is the number of objects sent out of the system during the computation.

Let us consider in some further detail the relation between the internal and the external mode of defining the result of a computation (in both cases, having as the result a number). Of course, for systems Π such that $N(\Pi)$ is defined in the external mode, no output membrane will be specified. Conversely, when we give a system Π without indicating an output membrane, this means that the result is defined in the external mode.

The following "bridge lemmas" show that there is a close relation between the two modes of defining the output of a computation.

Lemma 3.7.1. *Any membrane system Π, of any type (α, β, γ), with $\alpha \in \{ncoo, cat, 2cat\}$, $\beta \in \{tar, in\}$, and $\gamma \in \{\delta, \neg\delta\}$, with the output membrane being an elementary one in the initial configuration of Π, is equivalent to a system Π', of the same type and the same degree, with external output. When a priority relation is used in Π, the output membrane of Π may be arbitrary, not necessarily an elementary one.*

Proof. Let O and i_o be the alphabet and the output membrane of the system Π, and let $dom(R_{i_o}) = \{a \in O \mid$ there are rules of the forms $a \to v$, $ca \to cv$ in $R_{i_o}\}$. We change the system Π in the following way (the obtained system is denoted by Π').

Each symbol of the form (α, out) which appears in the rules associated with the skin membrane is replaced by d, where d is a new symbol ("dummy"), added to the alphabet of Π' (thus, no object sent out by Π will leave the skin membrane of Π').

If i_o is not the skin membrane, then for each $a \in O - dom(R_{i_o})$ we introduce the rule $a \to a'$ in R_{i_o}, the rule $a' \to (a', out)$ in all membranes excepting the skin membrane, and the rule $a' \to (a, out)$ in the skin membrane. If i_o is the skin membrane, then for each $a \in O - dom(R_{i_o})$ we introduce the rule $a \to (a, out)$ in R_{i_o}.

No other change is made. In particular, if Π has a priority relation, then it is the same for the system Π' (the new rules, involving the objects from $O - dom(R_{i_o})$, have no priority connection with the old rules). Thus, the

halting computations in Π are the same in Π', modulo sending out all objects which reach the output membrane and cannot evolve (these objects give the result of a computation with respect to Π). Consequently, the systems Π and Π' compute the same set of numbers.

We treat the case when we have a priority relation in more detail.

Consider a system $\Pi = (O, K, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m), i_o)$, with the output membrane i_o, which is not necessarily an elementary one. Let $O' = \{a' \mid a \in O\}$ and h be the morphism defined by $h(a) = a', a \in O$. We construct the system $\Pi' = (O'', K, \mu, fw_1, \ldots, f\#w_{i_o}, \ldots, fw_m, (R'_1, \rho_1), \ldots, (R'_{i_0}, \rho'_{i_0}), \ldots, (R'_m, \rho_m))$ (the object # is introduced only in membrane i_o) with

$$O'' = O' \cup (O - dom(R_{i_o})) \cup \{d, f, \#\},$$

and the sets $R'_i, 1 \leq i \leq m$, of rules obtained as follows:

- in all rules of Π we replace all symbols $a \in O$ with $a' \in O'$;
- if a symbol (a', out) appears in a rule from the skin membrane, then it is replaced with d;
- if $\beta = tar$, then to membrane i_o we add the rules $f \to (f, in_i)$, for all membranes i within i_o (this i is not necessarily the label of a membrane placed directly inside i_o, it can be separated from i_o by any number of intermediate membranes); if $\beta = in$, then to membrane i_o we add the rule $f \to (f, in)$;
- to all membranes i within membrane i_o, at any depth, we add the rule $f \to (f, out)$;
- to membrane i_o we add the rules $a' \to a$ for all $a \in O - dom(R_{i_o})$;
- to all membranes we add the rules $a \to (a, out)$, for all $a \in O - dom(R_{i_o})$;
- all membranes which contain membrane i_o have the rule $\# \to (\#, out)$, with the exception of the skin membrane, where we introduce the rule $\# \to \#$.

All rules from the set R'_{i_o} which are obtained by modifying as above the rules from R_{i_o} and all rules $f \to (f, in_i)$ or $f \to (f, in)$ from R'_{i_o} have priority over the rules $a' \to a$ from R'_{i_o}. All other priority relations, if any, remain unchanged for the corresponding rules from Π'.

A computation in Π is simulated in Π' by a computation which uses the primed version of objects in O. We can switch to non-primed objects only in membrane i_o, and only by using rules $a' \to a$, for $a \in O - dom(R_{i_o})$. Because all other rules from this membrane have priority over these rules, we can use such rules only when the computation in Π is finished, and moreover we have to wait until dissolving all membranes placed inside i_o. This is checked by the rules which use the object f. Note that in the initial configuration, a copy of this object is placed in any membrane of the system, including i_o and all membranes contained in i_o. At each step, both a rule $f \to (f, out)$ in a membrane inside i_o and a rule $f \to (f, in_i)$ or $f \to (f, in)$ in region i_o must be used. In this way, we always have at least one copy of f in the

output membrane; that is, we can use a rule $f \to (f, in_i)$ of $f \to (f, in)$. This prevents the use of the rules $a' \to a$. When there is no membrane inside i_o, then no rule of the form $f \to (f, in_i)$, $f \to (f, in)$ can be used. If the computation in Π is finished, then we can also finish the computation in Π'. Step by step, the symbols $a \in O - dom(R_{i_o})$ are sent out from all membranes, until they leave the system.

If the output membrane is dissolved (then no output is obtained in the system Π), then the trap object # is released to the upper membranes, and it will reach the skin region, where it will evolve forever by the rule $\# \to \#$. Thus, also the computation in Π' gives no output.

Consequently, we have $N(\Pi) = N(\Pi')$, which completes the proof. □

Lemma 3.7.1 says that when the output is defined by observing the system from outside we do not get anything more than by collecting the result in an output membrane. Also the converse of this result is true for all classes of systems, providing that one can use one more membrane.

Lemma 3.7.2. *Any membrane system Π, of any type (α, β, γ), with $\alpha \in \{ncoo, cat, 2cat\}, \beta \in \{tar, in\}$, and $\gamma \in \{\delta, \neg\delta\}$, of degree m, with external output, is equivalent to a system Π', of the same type, of degree $m+1$, with internal output. When the membrane dissolving action is not used in Π, the system Π' is of the same degree as Π.*

Proof. Let $\Pi = (O, K, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m))$ be a system with external output. As usual, the skin membrane is labelled with 1.

Consider first the case when the dissolving action is not used. Let $O' = \{a' \mid a \in O\}$ (a' is a new symbol associated with a), and let h be the morphism defined by $h(a) = a', a \in O$. We construct the system $\Pi' = (O' \cup O, K, \mu, h(w_1), \ldots, h(w_m), (R'_1, \rho_1), \ldots, (R'_m, \rho_m), i_o)$, where i_o is any elementary membrane of Π and the sets $R'_i, 1 \leq i \leq m$, of rules are obtained as follows starting from the rules of R_i:

- in all rules we replace all symbols $a \in O$ with $a' \in O'$;
- if a rule in the skin membrane contains a symbol (a', out), for $a \in O$, then we replace this symbol with a;
- to all membranes different from the skin membrane and from the output membrane i_o, we add the rules $a \to (a, out)$, for all $a \in O$;
- if $\beta = tar$, then to all membranes i different from i_o, we add the rules $a \to (a, in_j)$, for all $a \in O$ and each label j of a membrane inside membrane i; if $\beta = in$, then to all membranes different from i_o we add the rules $a \to (a, in)$, for all $a \in O$.

The priority relations refer to the modified version of the old rules and do not involve the new rules.

Using primed symbols, we simulate in Π' the computations in Π. Instead of sending symbols out, we direct them to membrane i_o. The computation

can stop correctly only if: (1) the simulated computation in Π stops (otherwise, the computation in Π' is not finished, because primed symbols can still evolve); (2) all symbols which are sent out by Π are collected in membrane i_o (otherwise rules for sending them up and down can still be used).

Clearly, the result of the corresponding computations in Π and Π' is the same.

When the dissolving action is allowed, we add to Π a new elementary membrane, with the label $m+1$, with no initial objects and no associated rules, and we call it the output membrane of Π'. The remaining part of the construction is the same as above. □

The reason why in the previous proof we need an additional membrane in the case when action δ is permitted is the fact that the membrane dissolving is a dynamic event: different membranes can be dissolved in different computations. For this reason, we cannot set one membrane in advance as the output membrane. In some sense, this is a further reason to read the result of a computation from outside the system (the skin membrane is never dissolved, hence the outer region is precisely defined and it is the same for all computations).

We will also see later (Section 8.1) that the question whether or not a given membrane is ever dissolved during the computations of a P system is not decidable for systems which are computationally universal (but it is decidable, for instance, for systems of type $(ncoo, \neg pri, \delta)$).

These bridge lemmas show that in what concerns the computational power, when considering a number as the result of a computation, the two modes of counting, internal and external, are quite similar, hence the external mode is not of much interest from this point of view. However, in the external mode we can have a qualitatively different type of output, namely, *we can get strings as the result of a computation.* This is a very important motivation for the external output, and it is implemented as follows. Consider the sequence of objects which leave the system during a (halting) computation, and arrange these objects in the order of leaving the system; when two or more objects exit the system at the same moment, then all permutations of them are considered (hence several strings are associated with the same computation, locally different, because of this kind of "local commutativity"). We will immediately illustrate these ideas. In this way, a system Π computes (generates) a language, the set of all strings associated as above with all halting computations in Π. We denote this language by $L(\Pi)$ and the family of languages generated by systems from a class corresponding to some family $NOP_m(\ldots)$ is denoted by $LOP_m(\ldots)$ (the initial L stands for "language"). All the ingredients considered for systems with internal output can also be considered for systems having the external output in the form of a string.

We examine several **examples**, both in order to illustrate the above definition and to get some hints about the power of systems generating languages in this manner.

Example 3.7.1. The system

$$\begin{aligned}
\Pi_1 &= (\{a,b\}, [_1[_2\ ]_2]_1, a, \lambda, R_1, R_2),\\
R_1 &= \{a \to a(a,out)(b,in),\ a \to (a,in),\ b \to (b,out)\},\\
R_2 &= \{a \to \delta\},
\end{aligned}$$

generates the language

$$L(\Pi_1) = \{a^n b^n \mid n \geq 1\}.$$

Therefore, this language is in the family $LOP_2(ncoo, in, \delta)$.

Example 3.7.2. The system

$$\begin{aligned}
\Pi_2 &= (\{a,b\}, [_1\ ]_1, a, R_1),\\
R_1 &= \{a \to a(a,out)b,\ a \to a,\ b \to b,\ a \to (a,out)b,\ b \to (b,out)\},
\end{aligned}$$

generates the Dyck language over $\{a,b\}$ (recall that the empty string is ignored when we compare two languages).

The use of the rule $a \to a$ changes nothing, so we have to start the computation by using one of the rules $a \to a(a,out)b$, $a \to (a,out)b$. The latter one can be used only once; let us assume that we start with a number of steps where the former rule is used. This means that the string starts with a. At any moment, any copy of objects a and b can evolve through the copying rules $a \to a$ and $b \to b$. Also, at any moment the number of objects b still present in the system or already sent out is equal to the number of objects a sent outside the skin membrane. The objects b can be sent out at any moment, but, because of the previous observation, the current string will always contain at most the same number of b occurrences as the number of a occurrences. Eventually, we have to use the rule $a \to (a,out)b$. The occurrence of b introduced in this way will be sent out after any occurrence of a. To summarise: the string starts with a and in any prefix of it the number of a occurrences is greater than or equal to the number of b occurrences. Hence it belongs to the Dyck language over $\{a,b\}$.

Conversely, each non-empty string in $D_{\{a,b\}}$ can be generated by Π_2. Indeed, take a string $w \in D_{\{a,b\}}$ and "mark the time" as follows. When an occurrence of a appears in the string, then use the rule $a \to a(a,out)b$, with the exception of the last occurrence of a, when the rule $a \to (a,out)b$ is used; at the moments corresponding to occurrences of b, we use the rule $a \to a$ and we also send out a copy of b by using the rule $b \to (b,out)$. In this way, we generate w.

Consequently, $D_{\{a,b\}} \in LOP_1(ncoo)$.

Example 3.7.3. Consider now a language similar to $L(\Pi_1)$, but with a more complex structure of symbols in the two "halves":

$$L = \{a^n x_1 x_2 \ldots x_n \mid n \geq 1, x_i \in \{de, ed\}, 1 \leq i \leq n\}.$$

It can be generated by the system (with catalysts)

$$\begin{aligned}
\Pi_3 &= (\{a,c,d,e\},\{c\},[_1[_2\ ]_2]_1, ca, \lambda, R_1, R_2),\\
R_1 &= \{a \to a(a,out)(d,in),\ a \to (a,in),\ cd \to c(d,out)(e,out)\},\\
R_2 &= \{a \to \delta\}.
\end{aligned}$$

For each copy of a which is sent out of the system, an occurrence of d is sent to the inner membrane, where it waits for the object a which determines the membrane dissolution. When the copies of d are released from the inner membrane, the catalyst c sends out one copy together with e. Thus, in our string we get consecutive substrings of the form de, ed, because both permutations of de should be considered.

The use of the catalyst is essential. If we were to use non-cooperative rules of the form $d \to d'(d,out)$, $d' \to (e,out)$, where d' is a new symbol, then they must be used at the same time for all occurrences of d, hence we would obtain a string of the form $a^n d^n e^n$. Consequently, $L \in LOP_2(cat,in,\delta)$ and $\{a^n d^n e^n \mid n \geq 1\} \in LOP_2(ncoo,in,\delta)$.

If we use only one rule, $d \to (d,out)(e,out)$, then the string will be of the form $a^n x$, with $x \in \{d,e\}^*$ and $|x|_d = |x|_e = n$: all objects d and e exit at the same time, hence all permutations of $d^n e^n$ should be considered.

Example 3.7.4. The system

$$\begin{aligned}
\Pi_4 &= (\{a,b\},[_1[_2\ ]_2]_1, \lambda, ab, R_1, R_2),\\
R_1 &= \{a \to (a,out),\ b \to (b,out)\},\\
R_2 &= \{a \to aa,\ b \to bbb,\ a \to aa\delta\},
\end{aligned}$$

generates the language

$$L(\Pi_4) = \{x \in \{a,b\}^* \mid |x|_a = 2^n, |x|_b = 3^n, n \geq 1\}.$$

In the inner membrane one produces 2^n copies of a and 3^n copies of b, for any $n \geq 1$. When membrane 2 is dissolved (this can be done at any time, by using the rule $a \to aa\delta$ instead of $a \to aa$), all these objects are left free in membrane 1 and sent out at the same time. Thus, any sequence of them is a valid output string. Consequently, $L(\Pi_4) \in LOP_2(ncoo,in,\delta)$. We mentioned in Subsection 2.3.5 (language L_{16} there) that this language is not in the $ET0L$ family.

Example 3.7.5. Let $p_1, p_2, \ldots, p_m$ be different prime numbers and consider the system (of degree $m+1$)

$$\begin{aligned}
\Pi_5 &= (\{a\},[_1[_2[_3 \cdots [_{m+1}\ ]_{m+1} \cdots]_3]_2]_1, \lambda, \ldots, \lambda, a, R_1, \ldots, R_{m+1}),\\
R_1 &= \{a \to (a,out)\},\\
R_{i+1} &= \{a \to a^{p_i},\ a \to a^{p_i}\delta\},\ 1 \leq i \leq m.
\end{aligned}$$

The reader can easily verify that

$$L(\Pi_5) = \{a^{p_1^{i_1} p_2^{i_2} \dots p_m^{i_m}} \mid i_j \geq 1, 1 \leq j \leq m\}$$

(each membrane "takes care" of one prime number, and because the communication to upper membranes is done by disolving membranes, outside the system we get the product of primes). Consequently, $L(\Pi_5) \in LOP_{m+1}(ncoo, in, \delta)$.

Up to now we have not used "strong" features, such as catalysts, priorities, or the conditional use of rules. Much more complex languages can be generated in such cases.

We call a language L *strictly bounded* if $L \subseteq a_1^* a_2^* \dots a_k^*$, for some symbols $a_1, a_2, \dots, a_k$ different from each other.

Theorem 3.7.1. *If h is a morphism and L is a strictly bounded semilinear language, then $h(L) \in LOP_1(cat, pri)$.*

Proof. Let $L \subseteq a_1^* a_2^* \dots a_k^*$ be a semilinear language and let $T = \{a_1, \dots, a_k\}$. Let $v_0^{(i)}, v_1^{(i)}, \dots, v_{m_i}^{(i)}$, for $1 \leq i \leq s$, be vectors in $\mathbf{N}^k$ such that

$$\Psi_T(L) = \bigcup_{i=1}^{s} \{v_0^{(i)} + \sum_{j=1}^{m_i} \alpha_j \cdot v_j^{(i)} \mid \alpha_1, \dots, \alpha_{m_i} \in \mathbf{N}\}.$$

Let $h : T^* \longrightarrow U^*$ be a morphism such that $h(a_i) = e_{i,1} e_{i,2} \dots e_{i,t_i}$, for $e_{i,j} \in U$, $1 \leq j \leq t_i$, $1 \leq i \leq k$. We assume that $t_i \geq 1$ for all i (otherwise we can remove the symbol a_i). We construct the following catalytic system (of degree 1):

$$\begin{aligned}
\Pi &= (O, \{c\}, [_1\ ]_1, bc, (R_1, \rho_1)),\\
O &= U \cup T \cup \{d_{i,j} \mid 1 \leq i \leq k, 1 \leq j \leq t_i\} \cup \{b, c\},\\
R_1 &= \{r_{0,i} : b \to b_i \mid 1 \leq i \leq s\}\\
&\quad \cup \{r_{i,j} : b_i \to b_i a_1^{v_j^{(i)}(1)} a_2^{v_j^{(i)}(2)} \dots a_k^{v_j^{(i)}(k)} \mid 1 \leq i \leq s, 1 \leq j \leq m_i\}\\
&\quad \cup \{r_{i,0} : b_i \to a_1^{v_0^{(i)}(1)} a_2^{v_0^{(i)}(2)} \dots a_k^{v_0^{(i)}(k)} \mid 1 \leq i \leq s\}\\
&\quad \cup \{p_{i,1} : ca_i \to cd_{i,1}(e_{i,1}, out) \mid 1 \leq i \leq k\}\\
&\quad \cup \{p_{i,j} : cd_{i,j} \to cd_{i,j+1}(e_{i,j+1}, out) \mid 1 \leq i \leq k, 2 \leq j \leq t_i - 2\}\\
&\quad \cup \{p_{i,t_i-1} : cd_{i,t_i-1} \to c(e_{i,t_i}, out) \mid 1 \leq i \leq k\},\\
\rho_1 &= \{r_{i,j} > p_{l,1} \mid 1 \leq i, l \leq s, 0 \leq j \leq m_i\}\\
&\quad \cup \{p_{i,l} > p_{i,j} \mid 1 \leq i \leq k, 1 \leq l < j \leq t_i\}\\
&\quad \cup \{p_{i,l} > p_{h,j} \mid 1 \leq i < h \leq k, 1 \leq l \leq t_i, 1 \leq j \leq t_h\}.
\end{aligned}$$

For $1 \leq i \leq s$, the rules $r_{0,i}$ choose the set i of vectors $v_0^{(i)}, v_1^{(i)}, \dots, v_{m_i}^{(i)}$ to be followed at the subsequent steps, the rules $r_{i,j}$ introduce the copies of

symbols $a_1, \ldots, a_k$ prescribed by the vectors $v_j^{(i)}$, for $1 \leq i \leq s$, $1 \leq j \leq m_i$, and the rules $r_{i,0}$ introduce the copies corresponding to vectors $v_0^{(i)}$, $1 \leq i \leq s$. These rules have priority over rules that can send symbols out. When no symbol b_i, $1 \leq i \leq k$, is present, we start to transform the symbols a_i into $h(a_i)$ and to send out the symbols of $h(a_i)$. Because of the priority relation among rules $p_{i,j}$, this is done starting with a_1 and finishing with a_k, in this order (note that $p_{1,1} > p_{2,1} > \ldots > p_{k,1}$). Because of the priority relation and because we use only one copy of the catalyst, after introducing the symbol $d_{i,1}$ we have to continue until finishing $h(a_i)$. Consequently, $L(\Pi) = h(L)$, and this completes the proof. □

If the morphism g is the identity, then we have $L(\Pi) = L$ and the catalyst is useless: instead of $p_{i,1}$ we consider the rules $p_i : a_i \to (a_i, out), 1 \leq i \leq k$, and the rules $p_{i,j}, j \geq 2$, are ignored. In this way, we obtain the following consequence of the previous theorem.

Corollary 3.7.1. *Every strictly bounded semilinear language belongs to the family* $LOP_1(ncoo, pri)$.

Let us now look for the possibility of generating languages from well-known families.

Theorem 3.7.2. $REG \subset LOP_1(ncoo)$.

Proof. Let $A = (Q, V, s_0, F, \delta)$ be a finite automaton. We construct the system

$$\begin{aligned} \Pi &= (Q \cup V \cup \{d\}, [_1\ ]_1, s_0, R_1), \\ R_1 &= \{s \to (a, out)s' \mid s' \in \delta(s, a)\} \\ &\quad \cup \{s_f \to d \mid s_f \in F\} \\ &\quad \cup \{s \to s \mid s \in Q - F\}. \end{aligned}$$

The equality $L(A) = L(\Pi)$ can be easily checked: due to the presence of rules $s \to s$, for all $s \in Q - F$, only complete parsings in A lead to correct computations in Π. Therefore, $REG \subseteq LOP_1(ncoo)$.

By Example 3.7.2, the inclusion is proper. □

Taking into account the fact that the strings of regular languages are generated/recognized "from left to right", one symbol at a time, this result is not at all surprising. Much more interesting is the following characterization of recursively enumerable languages, by means of systems with catalysts and promoters. In some sense, it corresponds to Theorem 3.6.4, but the important fact now is that in the system we work with multisets of symbol–objects and outside the system we get a language. Moreover, *any recursively enumerable language* can be generated in this way.

We *conjecture* that similar results hold also for other types of systems which are known to characterize NRE (Problem Q6). The difficulty lies in "programming" the passing from a number codifiying a string to the string itself (see also the proof of the theorem below).

Theorem 3.7.3. $LOP_1(cat, proR) = RE$.

Proof. Consider a language $L \in RE, L \subseteq T^*$, with $T = \{a_1, \ldots, a_k\}$. Interpret the strings in T^+ as numbers in base $k+1$, in the following sense: with the string $x = a_{i_1} a_{i_2} \ldots a_{i_r}, 1 \leq i_j \leq k, 1 \leq j \leq r$, we associate the numerical value

$$val(x) = i_1 + i_2(k+1) + i_3(k+1)^2 + \ldots + i_r(k+1)^{r-1}.$$

(Note that we do not use the digit 0 and that the "numbers" from T^+ are read in the reverse direction, the most significant digit being the righmost one.) Denote by $val(L)$ the one-letter language $\{a^n \mid n = val(x), x \in L\}$. Because $L \in RE$, we also have $val(L) \in RE$. Take a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the Z-normal form (in the standard notation), such that $L(G) = val(L)$. We construct the system

$$\Pi = (O, \{c\}, [_1\ ]_1, X_{init} A_{init} E r_0, R_1),$$

as in the proof of Theorem 3.6.4, but with only one membrane, with the rule $a \to (a, in)$ removed, and with the rule $E \to \lambda|_{d'_{n+1}}$ replaced with the rule $E \to H|_{d'_{n+1}}$. Because G was in the Z-normal form, the object H is introduced at the end of a successful computation, hence when in the unique membrane we also have a multiset $a^n = val(x)$ for some $x \in L$. We now modify the system Π as follows (the newly obtained system is denoted by Π').

What we want now is to recover x from $val(x)$, and to this end we will repeatedly divide n by $k+1$; at each step i, the remainder (always between 1 and k) will tell us the ith symbol of x, starting from the left end of x.

We add to the set O the following objects:

$$a_1, \ldots, a_k, a', b, b', c_1, c_2, D, F_0, F_1, \ldots, F_{k+1}, H, H', H'', H''', H^{iv}, H^v.$$

The objects c_1, c_2 are catalysts. To the initial configuration we add the multiset $c_1 c_2 D b^{k+1}$.

We also consider the following new rules:

1. $H \to H$,
   $c_1 a \to c_1|_H$,
   $c_2 b \to c_2 b'|_H$,
   $c_1 D \to c_1 \#|_H$,
   $c_2 D \to c_2 \#|_H$.
   (In the presence of the object H – and only in this case – we remove $k+1$ copies of a, at the same time marking the objects b with a prime. If the cutting of one a and priming one b is not synchronous, that is

either objects a or objects b are not sufficient, then the catalysts should be used by rules $c_1 D \to c_1 \#|_H$ and $c_2 D \to c_2 \#|_H$, which introduce the trap object $\#$. Because the rule $\# \to \#$ is already present in the system Π, the computation will never halt.)

2. $H \to H'a'$,
   $H' \to H''$,
   $b \to \#|_{H'}$.
   (At any moment, the object H can be replaced by H'; at the same step we have again to use the rules $c_1 a \to c_1|_H$ and $c_2 b \to c_2 b'|_H$, but if any copy of b still remains, at the next step, in parallel with the rule $H' \to H''$, we have to use the rule $b \to \#|_{H'}$, and the computation will continue forever. This ensures the fact that we have removed exactly $k+1$ copies of a. Note that a copy of a' was introduced in order to get the result of the division by $k+1$ of the initial number of copies of a.)
3. $H'' \to H$,
   $b' \to b|_{H''}$.
   (We return to the object H and at the same time all copies of b' are returned to b, hence the process can be repeated, and we can remove further $k+1$ copies of a from the current multiset.)
4. $H'' \to H''' F_0$,
   $H''' \to H'''$,
   $c_1 a \to c_1|_{H'''}$,
   $c_1 D \to c_1 \#|_{H'''}$,
   $F_i \to F_{i+1}|_a$, for $i = 0, 1, \ldots, k$,
   $F_{k+1} \to \#$.
   (At any moment, after a complete subtraction of $k+1$, the object H'' may choose not to return to H, but to introduce H'''. At the same step, all b' become b. Now, for a while we can use the rule $H''' \to H'''$ and in the presence of this object we start counting the number of copies of a still existing in the system. This is done by simultaneously removing a copy of a by the rule $c_1 a \to c_1|_{H'''}$ and increasing the subscript of objects F_i by rules of the form $F_i \to F_{i+1}|_a$. If in this way we go "too far", that is we reach F_{k+1}, then the trap object is introduced. In this way we check that we have completed the subtraction at the right time, when the remainder is less than or equal to k.)
5. $H''' \to H^{iv}$,
   $a \to \#|_{H^{iv}}$,
   $F_i \to (a_i, out)|_{H^{iv}}$.
   (At any moment, H''' can be changed with H^{iv} – in parallel we still count copies of a. Now, in the presence of H^{iv} any remaining copy of a is replaced by $\#$, and in this way we check that *all copies of a are counted.* In parallel, the available object F_i sends out the corresponding symbol a_i, contributing to the reconstruction of the desired string x.)

6. $H^{iv} \to H^v|_{a'}$,
   $H^v \to H$,
   $a' \to a|_{H^v}$.
   (If at least one copy of a' exists, hence the string x was not completely recovered from $val(x)$, then we return to the initial state, with the object H present, all existing a' unprimed, and $k+1$ copies of b already waiting in the system. If no copy of a' is available, then the computation stops, as no further rule can be applied.)
7. $H^v \to H'''F_0$,
   $E \to H'''F_0|_{r'_f}$.
   (Instead of returning to H, the object H^v may choose to introduce $H'''F_0$ directly, which means that we deal with the last symbol of the string x, that is less than $k+1$ copies of a are available in the system. No object a' will be introduced, hence we stop after introducing H^{iv} as mentioned above. The decision to introduce directly the objects $H'''F_0$ may be made also by the object E from the initial system Π, which means that in this way we can also generate the one-letter strings of the language L.)

From the previous explanations we can see that we stop if and only if we send outside Π' strings of L, and, conversely, each string of L can be generated in this way. That is, $L = L(\Pi)$, and the proof is complete. □

The external mode of defining the output of a computation will be systematically used in the case of systems with string–objects (Chapter 5), where the result of a computation will consist of all strings sent out of the system during the computation.

3.8 Bibliographical Notes

Membrane systems, both with symbol–objects and with string–objects, were introduced in a paper circulated as a TUCS Research Report (No. 208) in November 1998 and published in 2000 [186]. The term "P systems" was used for the first time by K. Krithivasan and her group in Madras.

Several related ideas have been investigated before. Among them, we mention the Chemical Abstract Machine, CHAM, introduced in [22], where one works with membranes (in a different sense, similar to what is here the contents of a region bounded from above by a membrane), but with completely different goals (process algebra), and with different tools. Multiset processing is also the central aim of the Gamma language, first considered in [16] (see the recent survey in [17]), but not in a distributed framework. Both CHAM and the Gamma language use multiset-rewriting-like rules of the general form $(u \to v, \pi)$, where π is a predicate which should be satisfied by the multiset where $u \to v$ is applied; such rules are rather far from the minimalistic style we have adopted here. The parallelism and the distributed organization of

a rewriting or recognizing device are old topics in formal language and automata theory – see L systems, grammar systems, regulated rewriting, etc. Membrane systems can be considered as a continuation of all these domains.

The contents of this chapter are based on several papers, but in most cases the proofs were simplified, taking advantage of two facts: (1) we work here with the number of objects from the output membrane, without distinguishing among these objects; in many papers, one deals with the multiplicity of these objects, hence instead of $N(\Pi)$ one considers $Ps(\Pi)$; (2) more efficient than this passing from $Ps(\Pi)$ to $N(\Pi)$ was the existence of the strong binary normal form, proved in [67], and of the Z-binary normal form, proved in [150]. Another difference with the literature, this time making some arguments slightly more difficult, is the fact that in many papers one also uses a terminal alphabet T of objects, and only objects from T are counted in the output membrane or outside the system; here we have chosen to work with the "pure" (non-extended) model.

There are already available several surveys of membrane computing, such as the "guide" [194], the early presentation from [185], as well as [153, 190, 151, 155] – the last two mainly devoted to the case of string–objects, which will be considered in Chapter 5.

Section 3.3 is based on [186, 48] – but the second paper works with $Ps(\Pi)$ instead of $N(\Pi)$. Theorems 3.3.1, 3.3.2 are from [48], Theorem 3.3.3 is new (in [48] priorities are also used), Theorem 3.4.2 is also from [48]. Priorities and action δ were already considered in [186], where Theorem 3.4.3 is given with a more complex proof. Theorem 3.4.4 is from [48]. Non-synchronized systems are considered in [203], where the contents of Subsection 3.4.5 can be found (Theorem 3.4.7 has a new proof).

Section 3.5 closely follows the paper [206]. A similar formalization can be found in [10], while [170] gives a formalization (of a different class of systems) with the membrane structure defined in terms of balls in the n-dimensional Euclidean space – that formalization can be extended to the case of systems with symbol–objects, priorities, and dissolving action, too. The formalization from [206] is used as a basis for the formal verification of specific P systems in [209] and [207]. For instance, in [207] it is rigorously verified that the system Π_1 from the first example from Subsection 3.4.3 really generates the squares of natural numbers.

The control of membrane permeability by means of actions δ, τ was introduced in [187], where the universality is given without a bound on the number of membranes. Theorem 3.6.1 is from [63]. Concentration-controlled P systems were considered in [49], where Theorem 3.6.2 can be found. It is also proved in [49] that if one uses only one bi-stable catalyst, then one can generate at least the Parikh sets of matrix languages, that is the family $PsMAT^{\lambda}$, without having a characterization of the power of systems of the type $(2cat, con)$ with a bounded number of bi-stable catalysts. In view of the fact that the control of communication by means of the concentration

is biologically important and of the fact that only a few results are known about this class of systems, we recall the problems from [49], about the power of systems of type $(2cat, con)$ with a given number of catalysts, as an *open problem* here (Problem Q7).

The idea of considering multisets of rules and to consume/create rules during a computation was introduced in [12], where Theorem 3.6.3 appears. This idea is also investigated in a series of papers by R. Freund, in a more general set-up, where rules and *operators* are counted, moved from a membrane to another membrane, etc. – see [58, 60].

Promoters and inhibitors at the level of sets of rules were considered in [30], where all Theorems 3.6.5–3.6.8 can be found.

The external mode to define the result of a computation, both as a number and as a string/language, was considered in [196] and after that followed in many papers (even some results recalled here for systems with internal output were originally given for the external output). The contents of Section 3.7 are based on [196]. Theorem 3.7.3 is new, but it imitates the main result from [99]. A similar characterization of recursively enumerable languages is obtained also in [217], but using cooperative rules and priorities, as well.

Further results about systems with symbol–objects can be found also in other papers. For instance, [119] considers *time-varying P systems* as a class with a decreased non-determinism: the membranes are active periodically, one at a time. Such systems with two membranes and using catalysts and priorities are shown to be universal – compare this result with Theorem 3.4.3, which is trivially time-varying, because membrane 2 is never working (this is not true for the proof given in [186] for Theorem 3.4.3). Systems with two membranes, using priorities but not catalysts, or using catalysts and action δ but not priorities, are proven in [119] to generate sets of vectors of natural numbers which are not in $PsET0L$. Also, partially deterministic and fully deterministic systems are considered in [119], but we do not recall the details here.

The same authors consider in [118] systems with symbol–objects where the same set of rules is used in all membranes; such a system is called *simple* in [118] (and *with global rules* in [176], where string–objects are dealt with). The main result from [118] says that simple P systems with bi-stable catalysts, without priorities or actions δ, τ, but with arbitrarily many membranes, are universal. Bridge lemmas for simple P systems with internal and external output are also given, in the style of Lemmas 3.7.1 and 3.7.2.

The PhD thesis of S.N. Krishna [108] – the first one devoted to membrane systems – contains all the results mentioned above, as well as several other related results.

A large number of papers are devoted to systems with symbol–objects, not of the type considered in this chapter, but with *active membranes*; this class will be investigated in Chapter 7, hence we will mention the bibliographical sources there.

4. Trading Evolution for Communication

In this chapter we still deal with membrane systems which have multisets of symbol–objects in their regions, but with the evolution rules of a type much different from the rewriting-like rules considered in the previous chapter, and closer to ways of transferring chemicals through membranes as encountered in biology. Specifically, *we will compute by communication only*, by changing the places of objects with respect to the membranes of a system, and not by changing the objects themselves. We will find that also in this framework we can get computational universality, and this is a rather interesting result, from various points of view. First, as we will see, the rules we use correspond directly to well-known biochemical processes. Second, because we do not change the objects, we do not create and we do not destroy them, and this means that the systems which follow observe the conservation law (which is not necessarily the case with the systems considered in the previous chapter, where rules of the form $a \to aa$ were used without taking care of the way of producing two copies of an object from only one existing copy). Third, the universality results show that in our framework communication is rather powerful. Communication makes the difference between a collection of separate computing agents and *a system* of cooperating agents; it is related to the so-called "system effect" (to synergy, emergent behaviour), so it is somewhat a common sense statement that communication is powerful. Still, our results are surprising in the fact that *communication alone is capable of universal computations* (in a precise set-up, as we will immediately define). Fourth, a technical detail: because no object can be created inside a system and because we start with finite multisets of objects present in the regions of a system, in order to compute arbitrary numbers we need to bring new objects from the environment; that is, not only do we send objects outside the system, but we also bring in objects from the environment. Thus, the environment is an active participant in a computation, a feature which is again rather similar to what happens in nature. The cells do not live alone, and are always in a continuous interaction with their environment. (Because the environment already contains arbitrarily many copies of all objects we need in the system, we have to return to the internal mode of defining the result of a computation, and to use again an output membrane specified in advance.)

All these observations are good reasons for investigating in some detail the purely communicative membrane systems – and to devote a separate chapter to them.

4.1 Systems with Symport/Antiport

We recall the biological observation that there are many cases where two chemicals pass through a membrane at the same time, with the help of each other, either in the same direction, or in opposite directions; in the former case we say that we have a *symport*, in the latter case we have an *antiport*.

Mathematically, we can capture the idea of symport by considering rules of the form (ab, in) or (ab, out), associated with a membrane, and stating that the objects a and b can enter, respectively, exit the membrane together. For antiport we consider rules of the form $(a, out; b, in)$, stating that a exits and at the same time as b enters the membrane.

Generalizing such kinds of rules, we can consider rules of the unrestricted forms (x, in) or (y, out) (generalized symport rules), and $(x, out; y, in)$ (generalized antiport rules), where x, y are strings representing multisets of objects, without any restriction on the length (of course, an antiport rule with one of x, y empty is nothing more than a symport rule).

Based on rules of this type, we consider *membrane systems with symport/antiport* in the form of constructs

$$\Pi = (O, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m, i_o),$$

where:

(i) O is the alphabet of *objects*;
(ii) μ is a membrane structure consisting of m membranes, with the membranes (and hence the regions) injectively labelled with $1, 2, \ldots, m$;
(iii) $w_i, 1 \leq i \leq m$, are strings which represent multisets over O associated with the regions $1, 2, \ldots, m$ of μ;
(iv) $E \subseteq O$ is the set of objects which are supposed to appear in the environment in arbitrarily many copies;
(v) $R_1, \ldots, R_m$ are finite sets of symport and antiport rules over O associated with the membranes $1, 2, \ldots, m$ of μ;
(vi) i_o is the label of an elementary membrane of μ (the *output membrane*).

For a symport rule (x, in) or (x, out), we say that $|x|$ is the *weight* of the rule. The *weight* of an antiport rule $(x, out; y, in)$ is $\max(|x|, |y|)$.

The rules from a set R_i are used with respect to the membrane i as suggested above. In the case of (x, in), the multiset of objects x enters the region defined by the membrane from the immediately upper region; this is the environment in the case when the rule is associated with the skin membrane. In the case of (x, out), the objects specified by x are sent out of

membrane i into the region immediately outside; this can be the environment in the case of the skin membrane. The use of a rule $(x, out; y, in)$ means expelling the objects specified by x from membrane i at the same time as bringing the objects specified by y into membrane i. The rules are used in the non-deterministic maximally parallel manner specific to membrane systems with symbol–objects. In this way, we obtain transitions between the configurations of the system.

A configuration is described by the m-tuple of the multisets of objects present in the m regions of the system, as well as the multiset of objects which were sent out of the system during the computation, other than the objects appearing in the set E; it is important to keep track of such objects because they appear in a finite number of copies (as is the case in the initial configuration) and can again enter the system. We do not need to take care of the objects from E which leave the system because they appear in arbitrarily many copies in the environment (the environment is supposed to be inexhaustible, irrespective of how many copies of an object from E are introduced into the system, still arbitrarily many remain in the environment). The initial configuration is $(w_1, \ldots, w_m, \lambda)$.

As usual, a sequence of transitions is called a computation, and with any halting computation we associate an output, in the form of the number of objects present in membrane i_o in the halting configuration. The set of these numbers computed by a system Π is denoted by $N(\Pi)$. The family of all sets $N(\Pi)$, computed in this way by systems Π of degree at most $m \geq 1$, using symport rules of weight at most p and antiport rules of weight at most q, is denoted by $NOP_m(sym_p, anti_q)$; when any of the parameters m, p, q is not bounded, we replace it with $*$.

The fact that the environment is inexhaustible implies that a system having a symport rule (x, in), for some $x \in E^*$, will have a strange behaviour: in each step *infinitely many* objects are brought into the system (and infinitely many remain outside, ready for the next step). That is why we forbid such "black-hole systems", and consider only systems without rules of this form. The reader should distinguish this kind of system from systems able to bring inside arbitrarily many objects (but not infinitely many!), for instance by means of an antiport rule $(a, out; aa, in)$, for $a \in E$. Such a rule increases indefinitely the number of copies of a present in the system, but at each moment this number is finite, double the number from the previous step. Such behaviour of the system is "normal" – although computationally non-productive, because no computation will ever stop.

Before investigating the power of these systems, we discuss a non-trivial **example**. Consider the system

$$\Pi = (O, \mu, w_1, w_2, E, R_1, R_2, 2), \text{ with}$$
$$O = \{a, b, c, c', d, e, e', f, g, \#\},$$
$$\mu = [_1[_2\ ]_2]_1,$$

$$
\begin{aligned}
w_1 &= ac,\\
w_2 &= df,\\
E &= \{a, b, c, c', e, e', f, g, \#\},\\
R_1 &= \{(c, out; \#, in),\ (ca, out; cbb, in),\\
&\quad (ca, out; c'bb, in),\ (da, out; \#, in),\\
&\quad (c'd, out; e, in),\ (eb, out; ea, in),\\
&\quad (eb, out; e'a, in),\ (fb, out; \#, in),\\
&\quad (e'f, out; cdf, in),\ (e'f, out; g, in)\},\\
R_2 &= \{(d, out; c', in),\ (c', out),\\
&\quad (f, out; e', in),\ (e', out),\\
&\quad (df, in),\ (\#, in),\\
&\quad (\#, out),\ (da, in),\ (g, out)\}.
\end{aligned}
$$

Assume that at some moment we have in the skin region n copies of the object a (initially, $n = 1$) and one copy of c, and in region 2 we have one copy of d and one copy of f. If we use the rule $(c, out; \#, in) \in R_1$, then $\#$ gets into region 1 from the environment. In this case the computation will never stop, because of the rules $(\#, in)$ and $(\#, out)$ from R_2 which cause $\#$ to oscillate between regions 1 and 2. Thus we have to begin by using the rule $(ca, out; cbb, in) \in R_1$ a number of times. When we replace all copies of a by b, then we have to use the rule $(c, out; \#, in) \in R_1$, and so the computation will not halt, hence we have to save one a and at some time to use the rule $(ca, out; c'bb, in) \in R_1$. The object c' must now go to membrane 2 and send from there object d, by the rule $(d, out; c', in) \in R_2$. In the next step, if any copy of a is still present in region 1, then the rule $(da, out; \#, in) \in R_1$ must be used, and thus again the computation will never stop. If no copy of a is present in region 1, that is the doubling of n was complete (the n copies of a were replaced by $2n$ copies of b), then d will wait one step in the skin region, as c' exits region 2, and then by the rule $(c'd, out; e, in) \in R_1$ the two objects exit the system together, while e is brought in. In the same way as c has changed all a to b, e will now change all b into a. When this operation is completed (and only then, since otherwise $\#$ will get into the system), e is replaced by e', and e' will send f out of membrane 2. The object f checks whether or not all copies of b were replaced by a, and only in the affirmative case will it exit the system together with e' and reintroduce the objects c, d, and f; d and f enter membrane 2, and the whole process can be iterated. In this way, we can double the number of copies of a as many times as we want, but at least once. At any moment, instead of the rule $(e'f, out; cdf, in) \in R_1$ we can use the rule $(e'f, out; g, in) \in R_1$. The object g will carry each copy of a into membrane 2 and when this operation is completed g gets stuck in the skin region. Therefore, $N(\Pi) = \{2^n \mid n \geq 1\}$.

The previous system Π uses symport rules of weight 2, but also uses antiport rules whose weight is not 1, as is the case in biology; we will see from the results which follow that one can generate this set $N(\Pi)$ using only symport rules of weight 2 and antiport rules of weight 1 (but then with five membranes).

4.2 Computational Universality

The **conventions** from the previous chapter are carried, *mutatis mutandis*, to membrane systems with symport/antiport (we do not state the obvious inclusions which follow from the definitions, but always state the results in the stronger form, etc.).

Obviously, systems with only one membrane can generate the length sets of context-free languages: for each rule $A \to x$ of a context-free grammar G, consider the antiport rule $(A, out; x, in)$, as well as all rules $(A, out; A, in)$ for all non-terminals A of G (to make sure that any halting computation corresponds to a terminal derivation in G). If we want to bound the weight of rules, then G should be considered in the Chomsky normal form, and the obtained system will be of type $(sym_0, anti_2)$.

On the other hand, from the example considered in the previous section we know that systems with two membranes (and rules of a large weight) can generate sets of numbers which are not semilinear; actually, two membranes are universal (providing that both symport and antiport rules are used).

A symport rule $(x, in) \in R_i$ can be considered as a cooperating rule $x \to (x, in)$ placed in the region immediately outside the membrane i, while a rule (x, out) is equivalent to $x \to (x, out)$ placed in region i (clearly, (x, tar) is the string obtained by attaching the target to each object from x). These rules are of a rather restrictive form; moreover, in the previous chapter we have not considered rules $u \to v$ in the environment. Still more "exotic" are the antiport rules, which do not have a counterpart in the framework of the previous chapter. Thus, intuitively, we can expect that membrane systems with symport and antiport rules will be powerful. This expectation is confirmed, as we get the computational universality, but it is rather interesting that such results hold even for systems with a small number of membranes and/or with rules of very reduced weights.

In the first result we allow both symport and antiport rules.

Theorem 4.2.1. $NOP_2(sym_2, anti_2) = NRE$.

Proof. We only have to prove the inclusion $NRE \subseteq NOP_2(sym_2, anti_2)$. Let $G = (N, \{a\}, S, M, F)$ be a matrix grammar with appearance checking in the Z-binary normal form, with the standard notation. We construct the system

$$\Pi = (O, [_1[_2\ ]_2]_1, w_1, w_2, E, R_1, R_2, 2),$$

with

$$
\begin{aligned}
O &= N_1 \cup N_2 \cup \{a, c, g, h, Z, \#\} \cup \{c_i, c_i', d_i \mid 1 \le i \le n\}, \\
w_1 &= cX_{init}A_{init}, \\
w_2 &= \lambda, \\
E &= N_1 \cup N_2 \cup \{a, g, h, Z, \#\} \cup \{c_i, c_i', d_i \mid 1 \le i \le n\}, \\
R_1 &= \{(cX, out; c_iY, in),\ (c_iA, out; cc_i', in), \\
&\quad (c_i', out; x, in),\ (c_i, out; \#, in) \mid m_i : (X \to Y, A \to x), 1 \le i \le k\} \\
&\quad \cup \{(c, out; \#, in),\ (cZ, out)\} \\
&\quad \cup \{(cX, out; c_id_i, in),\ (d_i, out; Yh, in),\ (c_iA, out; \#, in), \\
&\quad (h, out; cg, in),\ (c_ig, out) \mid m_i : (X \to Y, A \to \#), k+1 \le i \le n\}, \\
R_2 &= \{(a, in),\ (\#, in),\ (\#, out)\}.
\end{aligned}
$$

The objects c, c_i, c_i' control the simulation of matrices $m_i, 1 \le i \le k$, in the following way. After using the rule $(cX, out; c_iY, in) \in R_1$, we have c_i in the system. If the rule $(c_iA, out; c_i'c, in) \in R_1$ cannot be used, then the object c_i will bring the trap symbol # into the system, and this object will go forever back and forth through membrane 2, hence the computation will never finish. If the rule $(c_iA, out; c_i'c, in) \in R_1$ is used, then at the next step c_i' will exit, bringing into the system the multiset x, which completes the simulation of the matrix $m_i : (X \to Y, A \to x)$.

If we start with a rule of the form $(cX, out; c_id_i, in) \in R_1$, for some $m_i : (X \to Y, A \to \#)$, $k+1 \le i \le n$, then at the next step we have to use the rule $(d_i, out; Yh, in) \in R_1$. If in the skin membrane there is any copy of A, then the rule $(c_iA, out; \#, in)$ has to be used, and the computation will never finish, because of the rules $(\#, in)$ and $(\#, out)$ from R_2. If no copy of A is present, then c_i waits in the skin membrane. At the next step, h exits and brings cg inside. Now, together with g, the object c_i can leave the system. In this way, the application of matrix m_i was simulated.

In all cases the object c is again available, hence the process can be iterated. When the object Z is introduced, hence the derivation in G is terminal, we send it out together with the object c and the computation halts. Consequently, $N(\Pi) = length(L(G))$. (It is worth noting that the main part of the work of our system is done by the interaction between the skin membrane and the environment; membrane 2 is used only for storing the result of the computation and in order to prevent the "wrong" computations, by forever using the trap object after introducing it into the system.) □

The antiport rules used in the previous proof have the weight 2, hence they are not very "bio-realistic". We can eliminate this drawback (actually, we eliminate completely the antiport rules, which is a rather unexpected result), at the cost of using more membranes.

Theorem 4.2.2. $NOP_5(sym_2, anti_0) = NRE$.

Proof. Consider again a matrix grammar $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, with the standard notation, and construct the following system Π with symport rules only:

$$
\begin{aligned}
\Pi &= (O, \mu, w_1, w_2, w_3, w_4, w_5, E, R_1, R_2, R_3, R_4, R_5, 5),\\
O &= N_1 \cup N_2 \cup \{c_i, d_i, e_i, f_i \mid 1 \le i \le n\}\\
&\quad \cup \{a, b, c_{n+1}, g, Z, \#\},\\
\mu &= [_1[_2[_3[_4\ ]_4]_3]_2[_5\ ]_5]_1,\\
w_1 &= X_{init}A_{init}b,\\
w_2 &= e_1e_2\ldots e_nf_1f_2\ldots f_n,\\
w_3 &= d_1d_2\ldots d_n,\\
w_4 &= c_1c_2\ldots c_nc_{n+1}g,\\
w_5 &= \lambda,\\
E &= N_1 \cup N_2 \cup \{a, Z, \#\},\\
R_1 &= \{(c_iX, out),\ (d_iA, out),\ (e_if_i, out),\\
&\qquad (c_iY, in),\ (d_i, in),\ (e_i\alpha, in),\ (f_i\beta, in),\\
&\qquad (c_ib, out),\ (d_ib, out),\ (e_ib, out),\ (f_ib, out) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \alpha\beta), 1 \le i \le k,\\
&\qquad \text{with } X, Y \in N_1, A \in N_2, \text{ and } \alpha, \beta \in N_2 \cup \{a, \lambda\}\}\\
&\quad \cup \{(b\#, in),\ (c_{n+1}Z, out),\ (c_{n+1}b, out)\}\\
&\quad \cup \{(c_iX, out),\ (d_iA, out),\ (e_if_i, out),\\
&\qquad (c_iY, in),\ (d_i\#, in),\ (e_if_i, in),\\
&\qquad (c_ib, out),\ (e_ib, out),\ (f_ib, out) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \#), k+1 \le i \le n,\\
&\qquad \text{with } X \in N_1, Y \in N_1 \cup \{Z\}, \text{ and } A \in N_2\},\\
R_2 &= \{(c_ie_i, out),\ (d_if_i, out),\ (c_if_i, in),\ (d_ie_i, in) \mid 1 \le i \le n\}\\
&\quad \cup \{(c_{n+1}, out),\ (\#, in),\ (\#, out)\},\\
R_3 &= \{(c_id_i, out),\ (c_id_i, in) \mid 1 \le i \le n\}\\
&\quad \cup \{(c_{n+1}, out)\},\\
R_4 &= \{(gc_i, out),\ (gc_i, in) \mid 1 \le i \le n\}\\
&\quad \cup \{(gc_{n+1}, out)\},\\
R_5 &= \{(a, in)\}.
\end{aligned}
$$

Let us examine the functioning of Π beginning with a configuration which consists of the multisets w_2, w_3, w_4 present in membranes 2, 3, 4, respectively, any number of occurrences of a in membrane 5, and a multiset of the form Xw in region 1, where $X \in N_1$ and w contains objects from $N_2 \cup \{a, b\}$; initially, we have in the skin region only $X_{init}A_{init}b$, for $(S \to X_{init}A_{init})$ the start matrix of G. At any moment, each copy of the object a is sent to

membrane 5, hence from now on we will ignore this object. Note that # is a trap object; once # is introduced into the system, the computation will continue forever by oscillation caused by the rules $(\#, in)$ and $(\#, out)$ from R_2. Moreover, # can be introduced in the system either by the object b (this is the only role of b), or by the objects d_i, for $k+1 \leq i \leq n$, if any of these objects gets into the environment.

The computations in Π correspond to derivations in G, in the following sense. Each matrix $m_i, 1 \leq i \leq n$, can be simulated in Π, and this simulation is controlled by the objects c_i, d_i, e_i, f_i. If the simulation is correct and (in the case of matrices used in the appearance checking mode) the symbol # is not introduced in the derivation of G, then the computation in Π does not introduce the trap object # into the system, and it halts (only) when terminal derivations in G are simulated (hence object Z occurs).

In any configuration of the form $(Xw, w_1, w_2, w_3, a^r, w_e)$ – at this moment we ignore the objects from the environment – as described above (including the initial one), only a rule of the form (gc_i, out) from R_4 can be used, and so one object $c_i, 1 \leq i \leq n+1$, is sent out of membrane 4. Assume that $1 \leq i \leq n$. By using the rule (gc_i, in) we can return to the previous configuration, hence the computation continues. From region 3 we can send out the pair c_i, d_i, and then we can send out of region 2 the objects c_i, d_i, e_i, f_i (note that the initial value of i is preserved during all these steps). During these steps, g remains in region 3. These steps can be reversed, but this does not change anything, and so the computation continues.

Assume now that in the skin region we have the objects c_i, d_i, e_i, f_i, for some $1 \leq i \leq k$ (hence corresponding to a matrix of type 2, $m_i : (X \to Y, A \to \alpha\beta)$, with $\alpha, \beta \in N_2 \cup \{a, \lambda\}$). If we simultaneously use the rules (c_iX, out), (d_iA, out), (e_if_i, out) from R_1, then all these objects are sent out, and we have to use simultaneously the rules (c_iY, in), (d_i, in), $(e_i\alpha, in)$, $(f_i\beta, in)$, which complete the simulation of matrix m_i, and bring back the objects c_i, d_i, e_i, f_i into the skin region. Note that any or both of α, β can be empty. If any of the objects c_i, d_i, e_i, f_i is used by a rule (c_ib, out), (d_ib, out), (e_ib, out), (f_ib, out) from R_1, then the computation will never stop. These rules are useful in order to ensure that the simulation of the matrix m_i is correct (because otherwise # is introduced into the system). For instance, assume that we do not have the right objects X and A in the skin region. Then either (c_iX, out) or (d_iA, out) cannot be used. In order to avoid using (c_ib, out) we can use (c_if_i, in) from R_2, which means that the rule (e_if_i, out) cannot be used. Then either e_i exits together with b and the computation will never stop, or e_i enters region 2 using the rule $(d_ie_i, in) \in R_2$, which takes us back to a previous configuration, without simulating any matrix. Similar reasoning holds when we use the rule $(d_ie_i, in) \in R_2$ instead of (c_if_i, in).

Thus, in order not to introduce the trap object #, either the matrix is correctly simulated, or the computation just repeats the configuration, going through a useless loop. If $X = Y$, then we can correctly simulate the

same matrix again. Otherwise, the only way to continue without cycling and without introducing # into the system is to return the objects c_i, d_i, e_i, f_i to the regions where they were placed at the very beginning of the simulation. When c_i returns to region 4, it also brings back g, and the process can be iterated.

Assume now that we have started with some $i \in \{k+1, \ldots, n\}$, hence corresponding to a matrix $m_i : (X \to Y, A \to \#)$ with the second rule to be used in the appearance checking manner. Again, by the rule $(c_i X, out) \in R_1$, we send the object c_i to the environment. In the next step this c_i will bring back the object Y, but if any occurrence of A is present in the skin region, then the rule $(d_i A, out)$ has to be used, followed by $(d_i \#, in)$, and so the computation will never halt. If A is not present, then d_i will wait in the skin region (note that now $(d_i b, out)$ is not in R_1). Simultaneously with c_i leaving the system, e_i and f_i also exit into the environment and return at the next step. If e_i is not sent out, and rule $(d_i e_i, in)$ from R_2 is used (avoiding in this way the use of rule $(d_i A, out) \in R_1$), then f_i cannot exit the system (the rule $(e_i f_i, out) \in R_1$ is not applicable). If f_i uses the rule $(f_i b, out) \in R_1$, then the computation will never stop. On the other hand, if f_i uses the rule $(c_i f_i, in) \in R_2$, then the matrix m_i was not simulated, and we simply return to the previous configuration. If all the objects c_i, d_i, e_i, f_i are returned to the system, then matrix m_i is correctly simulated, and these objects can go to their original regions 2, 3, 4, and the process can continue (if $X = Y$, then m_i can be simulated again, returning to the same configuration).

Consequently, we can correctly simulate all non-terminal matrices of G, and the computation will halt only if it corresponds to a correct derivation in G. Note that the objects c_i, d_i, e_i, f_i can go forever back and forth through membranes 2, 3, 4 (they can even introduce the object # into the system) – so in order to halt the computation the object c_{n+1} has to be sent out of region 4. However, c_{n+1} leaves region 4 together with g; since c_{n+1} does not return to region 4, g remains forever in region 3. If Z is present in the skin region, then this means that c_{n+1} was introduced at the right moment. We can then use the rule $(c_{n+1} Z, out)$ and the computation stops. Otherwise we have to use the rule $(c_{n+1} b, out)$, and b will bring # into the system. Since the presence of Z indicates that the considered derivation in G is terminal, we stop at the completion of a correct simulation.

Therefore, $N(\Pi) = length(L(G))$, which completes the proof. □

The number of membranes can be reduced at the expense of using symport rules of a larger weight (and no antiport rule). In fact, a trade-off relation seems to exist between the number of membranes and the weight of symport rules necessary for obtaining the computational universality.

Theorem 4.2.3. $NOP_2(sym_5, anti_0) = NRE$.

Proof. Consider a matrix grammar with appearance checking $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, with the standard notation, and con-

struct the following system (of degree 2, using only symport rules):

$$
\begin{aligned}
\Pi &= (O, [_1[_2\ ]_2]_1, w_1, \lambda, E, R_1, R_2, 2),\\
O &= N_1 \cup N_2 \cup \{d_i, e_i \mid 1 \le i \le n+1\}\\
&\quad \cup \{a, b, c, e, f, g, h, Z, \#\},\\
w_1 &= X_{init} A_{init} bced_1 d_2 \ldots d_{n+1},\\
E &= N_1 \cup N_2 \cup \{a, f, g, h, Z, \#\},\\
R_1 &= \{(cd_i XA, out),\ (cd_i Y\alpha_1\alpha_2, in) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \alpha_1\alpha_2), 1 \le i \le k,\\
&\qquad \text{with } X, Y \in N_1, A \in N_2, \text{ and } \alpha_1, \alpha_2 \in N_2 \cup \{a, \lambda\}\}\\
&\quad \cup \{(bc, out),\ (b\#, in),\ (cd_{n+1}Z, out)\}\\
&\quad \cup \{(cd_i X, out),\ (d_i Y e_i h, in),\ (e_i Ab, out),\ (e_i g, out) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \#), k+1 \le i \le n,\\
&\qquad \text{with } X, Y \in N_1, \text{ and } A \in N_2\}\\
&\quad \cup \{(eh, out),\ (ceg, in)\},\\
R_2 &= \{(a, in),\ (\#, in),\ (\#, out)\}.
\end{aligned}
$$

Assume that in membrane 1 we have a multiset corresponding to a sentential form Xw of G (initially, we have $X_{init}A_{init}$), as well as copies of the objects b, c, e, and $d_1, \ldots, d_n, d_{n+1}$. If c does not exit together with an object $d_i, 1 \le i \le n+1$, then it will be used by the rule (bc, out) and will introduce a copy of the trap object $\#$ into the system, and this object will pass forever through membrane 2, preventing the halting of the computation. Thus, as long as c is present in the system, we have to continue the computation.

Assume that we use a rule $(cd_i XA, out)$, for some $1 \le i \le k$, such that $m_i : (X \to Y, A \to \alpha_1\alpha_2)$. At the following step we have to use the rule $(cd_i Y\alpha_1\alpha_2, in)$, and the matrix is correctly simulated. Because we return to a configuration with the objects c, d_i again present in membrane 1, we can continue.

Assume now that we start by using a rule $(cd_i X, out)$ for some $k+1 \le i \le n$, hence associated with a matrix $m_i : (X \to Y, A \to \#)$. The object d_i returns to the system by the rule $(d_i Y e_i h, in)$, which simulates the use of the first rule of the matrix, and, at the same time, the auxiliary objects e_i, h are introduced in the system. Because c is not present, we cannot start the simulation of another matrix. In the presence of e_i, if any copy of A is present, then we can send it out, together with the object b (by using the rule $(e_i Ab, out)$), and then b brings back the trap object $\#$, hence the computation never stops. If no copy of A is present in membrane 1, then the rule $(e_i Ab, out)$ cannot be used, and e_i waits in the system. In parallel, the rule (eh, out) is used, followed by (ceg, in), which brings back the objects c, e, as well as a copy of g. The multiset again contains the auxiliary objects b, c, e,

and $d_1, \ldots, d_{n+1}$, hence the simulation of matrices from M can be iterated. (The objects e_i, g exit immediately, by the rule $(e_i g, out)$.)

When Z is introduced, it will exit together with c and d_{n+1}, hence the computation can stop. All copies of the object a are introduced in the output membrane, hence $N(\Pi) = length(L(G))$, which completes the proof. □

Theorem 4.2.4. $NOP_3(sym_4, anti_0) = NRE$.

Proof. We start again from a matrix grammar $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, with the standard notation, and we construct the system (of degree 3)

$$
\begin{aligned}
\Pi &= (O, [_1[_2\ ]_2[_3\ ]_3]_1, w_1, \lambda, w_3, E, R_1, R_2, R_3, 2),\\
O &= N_1 \cup N_2 \cup \{d_i, e_i \mid 1 \le i \le n+1\}\\
&\quad \cup \{a, b, c, c', c'', e, g, h, q, Z, \#\},\\
w_1 &= A_{init} b e d_1 d_2 \ldots d_n,\\
w_3 &= X_{init} q c c' c'',\\
E &= N_1 \cup N_2 \cup \{a, g, h, Z, \#\},\\
R_1 &= \{(c d_i X A, out),\ (d_i Y \alpha_1 \alpha_2, in) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \alpha_1\alpha_2), 1 \le i \le k,\\
&\qquad \text{with } X, Y \in N_1, A \in N_2, \alpha_1, \alpha_2 \in N_2 \cup \{a, \lambda\}\}\\
&\quad \cup \{(c, in),\ (bc, out),\ (b\#, in)\},\ (c'' d_{n+1} Z, out)\}\\
&\quad \cup \{(c' d_i X, out),\ (d_i Y e_i h, in),\ (e_i A b, out),\ (e_i g, out) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \#), k+1 \le i \le n,\\
&\qquad \text{with } X, Y \in N_1, \text{ and } A \in N_2\}\\
&\quad \cup \{(eh, out),\ (c' eg, in),\ (q, out)\},\\
R_2 &= \{(a, in),\ (\#, in),\ (\#, out)\}\\
R_3 &= \{(qc, out),\ (qc, in),\ (qc', out),\ (qc', in),\ (qc'', out)\}\\
&\quad \cup \{(X, in),\ (X, out) \mid X \in N_1\}.
\end{aligned}
$$

The difference from the system constructed in the previous proof is that the simulation of non-terminal matrices $m_i, 1 \le i \le k$, is done in the presence of the object c, the simulation of the terminal matrix $(Z \to \lambda)$ is done in the presence of the object c'', while the simulation of matrices $m_i, k+1 \le i \le n$, is done in the presence of the object c'. These objects are released from membrane 3 (at the same time with the unique object from N_1) by means of the object q, which non-deterministically chooses one of them. When c is in the skin membrane, several matrices without rules to be used in the appearance checking manner can be simulated, and then the object c returns to membrane 3. As long as any object $X \in N_2$ is present in the system, the computation should continue, as we can use the rules $(X, in), (X, out)$ from R_3. After simulating a matrix having a rule used in the appearance

checking mode, the object c' has to return to membrane 3 (we can simulate two matrices $m_i, k+1 \leq i \leq n$, in a row only if they have the first rule of the form $X \to X$). We get $N(\Pi) = length(L(G))$. □

It is not known whether or not the previous results are optimal (in the number of membranes for the given weight of rules, and in the weight of rules for the given number of membranes). We *conjecture* that some of these results can be improved (Problem Q8).

At first sight, the antiport rules are a generalization of symport rules, because each rule (u, out) or (u, in) can be replaced by $(u, out; d, in)$, respectively $(d, out; u, in)$, where d is a dummy object. Actually, this is not true, as no system Π using only antiport rules can compute both the number 0 and a non-null number, because if $0 \in N(\Pi)$, then $N(\Pi) = \{0\}$: if the output membrane is initially empty, then it will remain empty forever; if it is not initially empty, then it will never become empty.

However, if we consider two sets of numbers to be equal if they differ at most in the element 0 – or we ignore the number 0 – then all recursively enumerable sets of numbers can be computed by systems using only antiport rules. The proof of this assertion is not trivial, because we have to start with finite multisets of objects present in the system and we have to ensure that arbitrarily large multisets can be produced. (Remember that in Section 3.6 we denoted by $N'RE$ the family of recursively enumerable sets of non-null natural numbers.)

Theorem 4.2.5. $NOP_3(sym_0, anti_2) = N'RE$.

Proof. The proof is based on the construction from the proof of Theorem 4.2.1, with additional care paid to the fact that each symport rule should be replaced by an antiport rule, hence "sufficient" objects should be available when such rules are used.

Let $G = (N, \{a\}, S, M, F)$ be a matrix grammar with appearance checking in the Z-binary normal form. We construct the system with antiport rules

$$
\begin{aligned}
\Pi &= (O, [_1[_2\ ]_2[_3\ ]_3]_1, w_1, h', g, E, R_1, R_2, R_3, 2),\\
O &= N_1 \cup N_2 \cup \{a, c, f, g, h, h', h'', F, H, Z, \#\}\\
&\quad \cup \{c_i, c_i', d_i \mid 1 \leq i \leq n\},\\
w_1 &= X_{init} A_{init} ch,\\
E &= N_1 \cup N_2 \cup \{a, f, h, h', h'', F, H, Z, \#\} \cup \{c_i, c_i', d_i \mid 1 \leq i \leq n\},\\
R_1 &= \{(cX, out; c_iY, in),\ (c_iA, out; cc_i', in),\ (c_i', out; u, in) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to u), 1 \leq i \leq k, \text{ with } |u| \neq 0\}\\
&\quad \cup \{(cX, out; c_iY, in),\ (c_iA, out; c, in) \mid\\
&\qquad \text{for } m_i : (X \to Y, A \to \lambda), 1 \leq i \leq k\}\\
&\quad \cup \{(cZ, out; f, in)\}
\end{aligned}
$$

$$\cup \{(cX, out; c_i d_i, in),\ (d_i, out; YH, in),\ (c_i A, out; \#, in),$$
$$(H, out; cF, in),\ (c_i F, out; h'', in) \mid$$
$$\text{for } m_i : (X \to Y, A \to \#), k+1 \leq i \leq n\}$$
$$\cup \{(h, out; hh, in),\ (h, out; h', in)\},$$
$$R_2 = \{(h', out; h'h', in),\ (h', out; a, in),\ (h', out; h', in)\},$$
$$R_3 = \{(g, out; \alpha, in),\ (\alpha, out; g, in) \mid \alpha \in \{\#\} \cup \{c_i, c_i' \mid 1 \leq i \leq k\}\}.$$

The way this system works in simulating the grammar G is already known to the reader; what is new here is the way of providing sufficiently many objects in membrane 2, so that each copy of a can be introduced here. Initially, we have one copy of h in membrane 1 and one copy of h' in membrane 2. By using the rules $(h, out; hh, in), (h, out; h', in)$ any number of copies of h' can be introduced in membrane 1. From here, any number of copies of h' can be introduced into membrane 2. If all these symbols h' are exhanged for copies of a, then no rule $(h', out; x, in)$ can be used for membrane 2. Thus, if we introduce exactly m copies of h' when we want to generate the number m, then the computation will stop correctly. □

We do not know what the power of systems is with only one membrane (Problem Q9), in particular whether or not the family $NOP_1(sym_2, anti_1)$ contains infinite sets of numbers.

The last question has a positive answer for the family $NOP_1(sym_3, anti_0)$: consider the system

$$\Pi = (\{a, b, c, d\}, [_1\ ]_1, bcd, \{a\}, R_1, 1),$$
$$R_1 = \{(bcd, out),\ (ba, in),\ (cd, in),\ (c, out)\}.$$

Starting from a multiset $a^n bcd$ (initially we have $n = 0$), we can either send c out, by the rule (c, out), and the computation stops, or use the rule (bcd, out); at the next step, b returns together with one copy of a, and also the objects cd come back. The operation can be iterated, hence the number of copies of the object a can increase as much as we want. Consequently, $N(\Pi) = \{n \mid n \geq 2\}$.

4.3 Controls on the Use of Rules

The number of membranes used in Theorem 4.2.2 – five – for characterizing NRE can de reduced if the symport rules have permitting or forbidding context conditions (promoting or inhibiting objects, in the sense already used in Subsection 3.6.5 for rules of the form $u \to v$).

A symport rule with a permitting condition is given in the form $(x, in)|_a$ or $(x, out)|_a$, where a is an object; this object should be present in membrane i when a rule $(x, in)|_a$, $(x, out)|_a$ is applied (in the second case, a should not be an element of x, for that multiset x of objects for which the rule is

used – but this copy of a can evolve at the same step by means of a rule not promoted by it). A symport rule with a forbidding condition is given in the form $(x, in)|_{\neg a}$ or $(x, out)|_{\neg a}$, and such a rule can be used only if a is not present in the membrane where the rule is used.

Similarly, an antiport rule with a permitting (forbidding) condition is of the form $(x, out; y, in)|_a$ (resp., $(x, out; y, in)|_{\neg a}$).

Rules without conditions are used freely, considered by default as having an empty permitting/forbidding condition, which is always satisfied.

The use of permitting (forbidding) conditions is indicated by replacing *sym* by *psym* (*fsym*, respectively) and *anti* by *panti* (*fanti*, respectively) in the notation $NOP_m(sym_p, anti_q)$. As expected, the use of context conditions is useful, as the number of membranes sufficient for characterizing NRE can be (slightly) decreased.

Theorem 4.3.1. $NOP_3(psym_2, anti_0) = NOP_3(fsym_2, anti_0) = NRE$.

Proof. We give the proof only for the forbidding case (remember that in Subsection 3.6.5 we have given a result only for the case of promoters – that time, for systems with multiset rewriting rules).

We start from a matrix grammar $G = (N, \{a\}, S, M, F)$ with appearance checking in the Z-binary normal form, and we construct the following system with symport rules having forbidding contexts:

$$
\begin{aligned}
\Pi &= (O, \mu, w_1, w_2, w_3, E, R_1, R_2, R_3, 3),\\
O &= N \cup \{a, b, f, f', g, h, c_{n+1}, d_{n+1}, Z, \#\}\\
&\quad \cup \{c_i, d_i, d_i', d_i'', d_i''', d_i^{iv} \mid 1 \le i \le k\}\\
&\quad \cup \{c_i, d_i, d_i', d_i'', d_i''' \mid k+1 \le i \le n\},\\
\mu &= [_1[_2]_2[_3]_3]_1,\\
w_1 &= X_{init} A_{init} b f f',\\
w_2 &= d_1 d_1' d_1'' d_1''' d_1^{iv} c_1 \ldots d_k d_k' d_k'' d_k''' d_k^{iv} c_k d_{k+1} d_{k+1}' d_{k+1}'' d_{k+1}''' c_{k+1}\\
&\quad \ldots d_n d_n' d_n'' d_n''' c_n d_{n+1} c_{n+1},\\
w_3 &= \lambda,\\
E &= N_1 \cup N_2 \cup \{a, g, h, Z, \#\},\\
R_1 &= \{(d_i X, out)|_{\neg g},\ (d_i' A, out)|_{\neg g},\ (d_i'' d_i''', out)|_{\neg g},\\
&\quad (d_i^{iv}, out)|_{\neg g},\ (d_i b, out)|_{\neg g},\ (d_i' b, out)|_{\neg g},\ (d_i g, in),\\
&\quad (d_i' h, in),\ (d_i'' Y, in),\ (d_i''' \alpha, in),\ (d_i^{iv} \beta, in) \mid\\
&\quad \text{for } m_i : (X \to Y, A \to \alpha\beta), 1 \le i \le k,\\
&\quad \text{with } X, Y \in N_1, A \in N_2, \text{ and } \alpha, \beta \in N_2 \cup \{a, \lambda\}\}\\
&\quad \cup \{(d_i X, out)|_{\neg g},\ (d_i' A, out)|_{\neg g},\ (d_i'' d_i''', out)|_{\neg g},\\
&\quad (d_i g, in),\ (d_i' b, out)|_{\neg g},\ (d_i' \#, in),\\
&\quad (d_i'' Y, in),\ (d_i''' h, in) \mid \text{for } m_i : (X \to Y, A \to \#),
\end{aligned}
$$

$$
\begin{aligned}
& k+1 \leq i \leq n, \text{ with } X, Y \in N_1, \text{ and } A \in N_2\} \\
\cup\ & \{(d_{n+1}Z, out),\ (d_{n+1}b, out),\ (c_{n+1}f, out), \\
& (gh, out)|_{\neg f},\ (b\#, in),\ (f'h, out)|_{\neg f}\}, \\
R_2 = & \{(fc_i, out),\ (c_i, in),\ (d_i d_i', out)|_{\neg c_i}, \\
& (d_i'' d_i''', out)|_{\neg c_i},\ (d_i d_i', in)|_{\neg f'},\ (d_i'' d_i''', in)|_{\neg f'} \mid 1 \leq i \leq n\} \\
\cup\ & \{(d_i^{iv}, out)|_{\neg c_i},\ (d_i^{iv}, in)|_{\neg f'} \mid 1 \leq i \leq k\} \\
\cup\ & \{(ff', in),\ (h, in),\ (f'h, out),\ (\#, in),\ (\#, out), \\
& (fc_{n+1}, out),\ (d_{n+1}, out)|_{\neg c_{n+1}}\}, \\
R_3 = & \{(a, in)\}.
\end{aligned}
$$

We will prove that Π correctly simulates any terminal derivation in G, and that a computation stops only if it corresponds to a terminal derivation in G.

Suppose that we have a configuration where the objects f and f' are present in region 1 (initially this is the case). By using the rule $(ff', in) \in R_2$ we send f and f' into region 2. Now, in region 2 we can use the rule (fc_i, out), which sends the object c_i to region 1, meaning that we can start the simulation of matrix m_i.

Assume that we have $1 \leq i \leq k$, corresponding to a matrix $m_i : (X \to Y,\ A \to \alpha\beta)$ of type 2.

Because c_i is no longer present in region 2, we can simultaneously apply the rules $(d_i d_i', out)|_{\neg c_i}$, $(d_i'' d_i''', out)|_{\neg c_i}$, $(d_i^{iv}, out)|_{\neg c_i}$. Similar to the simulation from the proof of Theorem 4.2.2, we continue by sending d_i together with X, and d_i' together with A, out of the system (also d_i'', d_i''', d_i^{iv} are sent out). We can do this because the object g is not yet present in region 1. Now, all of the d objects just expelled from the system come back together with various "accompanying objects": d_i with g, d_i' with h, d_i'' with Y, d_i''' with α, and d_i^{iv} with β. It is easy to see that in region 1 X is replaced by Y and A by $\alpha\beta$, so we have correctly simulated the matrix m_i.

At the next step we have the object g in region 1, meaning that the objects $d_i, d_i', d_i'', d_i''', d_i^{iv}$ cannot exit the system, and the object h enters region 2. Then, f' and h exit region 2, meaning that at the next step the objects $d_i, d_i', d_i'', d_i''', d_i^{iv}$ present in the skin region can return to region 2 (where they belong after a correct simulation). Therefore, f and f' can now initiate a new simulation. Also, g and h can now exit the system since f is not present in region 1 (it has just entered region 2 to initiate a new simulation).

If either X or A (or both of them) is not present in region 1 at the beginning of the simulation of matrix m_i, then one of the objects d_i or d_i' has to exit the system together with b – this will bring the trap object $\#$ into the system and the computation will continue forever using the rules $(\#, in)$ and $(\#, out)$ from R_2.

Assume now that we have started with $k+1 \leq i \leq n$, corresponding to a matrix $m_i : (X \to Y,\ A \to \#)$ of type 3.

The simulation starts in the same way: f and f' get into region 2, then f leaves region 2 together with the object c_i. At the next step all objects d_i, d_i', d_i'', d_i''' exit into region 1 (because c_i is no longer present in membrane 2). Then d_i exits the system together with X, and d_i'' exits the system together with d_i'''. At this point we perform the appearance checking in the following way: d_i' can get out only together with A, but it gets back into the system together with # (and in this way the computation will continue forever). If there is no occurrence of A in the system, then d_i' cannot exit. At the next step all objects d_i, d_i'', d_i''' get back into region 1 bringing with them the objects Y, g, h. From now on we perform the same steps as in the case of matrices of type 2: h enters region 2, then exits together with f', the objects d_i, d_i', d_i'', d_i''' from region 1 get back into region 2, and the coupled objects ff' initiate another simulation.

At any moment in the simulation we can also send the objects c_{n+1}, d_{n+1} out of region 2. If Z is present in the skin region, then the rule $(d_{n+1}Z, out)$ can be used (and we know that the derivation in G was terminal); otherwise we have to use $(d_{n+1}b, out)$, which will lead to a non-halting computation. Thus, objects d_{n+1} and Z can be sent out of region 2 at "the right moment". In this case, c_{n+1} will leave the system together with the object f, and consequently the computation halts (f and f' cannot return to region 2).

Since each copy of a ends in the output membrane, we have the equality $N(\Pi) = length(L(G))$, and this completes the proof for the forbidding case.

The proof for the permitting case is principially similar, only the technical details are different – and left as a task to the reader. □

The power of systems with permitting or forbidding conditions imposed on antiport rules have not been investigated – Problem Q10.

Moreover, also for systems with symport/antiport rules one can introduce the control of membrane thickness by means of actions δ, τ, by considering rules of the forms $(x, in)\alpha$, $(x, out)\alpha$, $(x, out; y, in)\alpha$, for $\alpha \in \{\delta, \tau\}$. The effect of such an extension on the power of our devices has still to be investigated (Problem Q11).

4.4 Following the Traces of Objects

As we have also mentioned in Section 3.7, in the case of systems with symport/antiport rules it is not natural to consider an external output, because we already have objects in the environment, even in an "infinite" number ("infinite" here means "as many as necessary in any computation", without any bound on the multiplicities). On the other hand, working with multisets of objects and trying to generate strings is a challenging task. In the case of purely communicative systems, as those with symport/antiport are, there is a nice possibility of answering such a question, namely, by considering the itineraries of objects through membranes as strings. More specifically, we

consider a sequence of events, for instance the moment when an object enters a membrane, and we write the string which describes all such events with respect to a given object or a given set of objects, from the initial configuration of the system until the halting configuration of a computation. In this way, a membrane system again generates a language.

We introduce this idea here for the general case when the rules have promoters/inhibitors. Specifically, we consider systems of the form

$$\Pi = (O, t, T, h, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m),$$

where O is the alphabet of objects, $t \in O$ is a distinguished object ("the traveller"), T is an alphabet (no relation with O is assumed), $h : \{1, 2, \ldots, m\} \longrightarrow T \cup \{\lambda\}$ is a weak coding, $w_1, \ldots, w_m$ are strings representing the multisets of objects present in the m regions of μ, E is the set of objects present in the environment in an arbitrary number of copies, and $R_1, \ldots, R_m$ are the sets of symport and antiport rules (with promoters/inhibitors) associated with the m membranes. The traveller is present in exactly one copy in the system, that is $|w_1 \ldots w_m|_t = 1$ and $t \notin E$.

Let $\sigma = C_1 C_2 \ldots C_k, k \geq 1$, be a halting computation with respect to Π, with $C_1 = (w_1, \ldots, w_m)$ the initial (internal) configuration, and $C_i = (z_1^{(i)}, \ldots, z_m^{(i)})$ the configuration at step $i, 1 \leq i \leq k$ (we ignore the objects from the environment, as we do not take into account the environment when defining the trace of the traveller). If $|z_j^{(i)}|_t = 1$ for some $1 \leq j \leq m$, then we write $C_i(t) = j$ (therefore, $C_i(t)$ is the label of the membrane where t is placed in configuration C_i). If $|z_j^{(i)}|_t = 0$ for all $j = 1, 2, \ldots, m$, then we put $C_i(t) = \lambda$. Then, *the trace* of t in the computation σ is

$$trace(t, \sigma) = C_1(t) C_2(t) \ldots C_k(t).$$

Note that the trace starts with the label of the membrane where t is placed in the initial configuration.

The computation σ is said to generate the string $h(trace(t, \sigma))$, hence the language generated by Π is $L(\Pi) = \{h(trace(t, \sigma)) \mid \sigma$ is a halting computation in $\Pi\}$. (By means of h we ignore some labels and we can rename the others.)

We denote by $LOP_m(psym_p, panti_q)$ the family of languages $L(\Pi)$ generated by P systems with at most m membranes, with symport rules of weight at most p and antiport rules of weight at most q, using promoters; when the rules have associated forbidding conditions we write *fsym, fanti* instead of *psym, panti*; when the rules are used in the free mode (they have no associated promoter/inhibitor symbols), we remove the initial "p" and "f" from *psym, panti*, and *fsym, fanti*. As usual, the subscripts m, p, q are replaced by $*$ when no bound on the corresponding parameter is considered.

Before starting to investigate the power of this way of generating a language, let us examine an **example**. Consider the system

$$\Pi = (\{d,t\}, t, \{a,b,c\}, h, \mu, t, \emptyset, \emptyset, \emptyset, \emptyset, \{d\}, R_1, R_2, R_3, R_4, R_5),$$

with the weak coding $h(1) = a, h(3) = b, h(5) = c, h(2) = h(4) = \lambda$, the membrane structure $\mu = [_1[_2[_3[_4[_5\ ]_5]_4]_3]_2]_1$, and the following rules:

$$R_1 = R_3 = R_5 = \{(t, out),\ (td, in)\},$$
$$R_2 = R_4 = \{(t, in),\ (d, in)\}.$$

First, the traveller brings $n \geq 1$ copies of d from the environment (each of them immediately enters membrane 2), then t goes to membrane 2. From here, the traveller brings $m \leq n$ copies of d into membrane 3 (each of them immediately enters membrane 4), then t goes to membrane 4 (at that moment, it is possible that some copies of d remain in membrane 2). From membrane 4, all copies of d are carried into membrane 5; the computation stops with the traveller in membrane 4. Thus, for any computation σ of this type, we have

$$trace(t, \sigma) = 1^{n+1}(23)^m(45)^m 4, \text{ for some } n \geq 1, m \leq n.$$

The traveller can also end up in membrane 2, after introducing all copies of d in membrane 3, and returning to membrane 2. In the case of such a computation σ we have

$$trace(t, \sigma) = 1^{n+1}(23)^n 2, \text{ for some } n \geq 1.$$

Finally, we can also have the trivial computation where t just enters membrane 2, without any copy of d present in the system, and this leads to $trace(t, \sigma) = 12$.

Consequently,

$$L(\Pi) = \{a^{n+1}b^m c^m \mid n \geq 1, m \leq n\} \cup \{a^{n+1}b^n \mid n \geq 1\} \cup \{a\}.$$

Clearly, this language is not context-free. Note that the system Π has only symport rules, and that the rules are freely applied (we use no promoter or inhibitor).

This example proves the relation $LOP_5(sym_2, anti_0) - CF \neq \emptyset$. Clearly, by removing membranes 2 and 3 (as well as all their associated elements), we get a system of degree 3 which generates a non-regular language, hence we also have $LOP_3(sym_2, anti_0) - REG \neq \emptyset$. The result for the context-free case can be slightly improved.

Theorem 4.4.1. $LOP_4(sym_2, anti_0) - CF \neq \emptyset$.

Proof. Consider the system

$$\Pi = (\{t,d\}, t, \{a,b,c\}, h, [_1[_2[_3[_4\ ]_4]_3]_2]_1, t, \emptyset, \emptyset, \emptyset, \{d\}, R_1, R_2, R_3, R_4),$$

with the morphism $h(1) = a, h(2) = b, h(3) = c, h(4) = \lambda$, and the rules

$$R_1 = R_2 = R_3 = \{(t, out),\ (td, in)\},$$
$$R_4 = \{(t, in)\}.$$

It is easy to see that

$$L(\Pi) \cap a^*(ba)^*(bc)^* = \{a^{n+1}(ba)^m(bc)^r \mid n \geq 1, n \geq m \geq r\},$$

and this is not a context-free language (the traveller ends up in membrane 4, after carrying copies of d through membranes; some of these copies can be left behind, hence the relations among n, m, r follow). □

We do not know whether or not all recursively enumerable languages can be generated as traces of objects in membrane systems as above, without promoters and inhibitors, but the power of such machinery is very large even in the non-controlled case: all one-letter recursively enumerable languages can be obtained in this way. This assertion is a consequence of the results from Sections 4.2 and 4.3 and of the proofs of these results. In all these proofs one starts from a matrix grammar $G = (N, \{a\}, S, M, F)$ and one constructs systems Π where the only rules associated with the output membrane and involving the unique symbol a are of the form (a, in). Consider such a system Π, add a symbol t to its alphabet, introduce this symbol into the initial configuration in the region immediately outside the output membrane, then replace each rule $(a, in) \in R_{i_o}$ with $(ta, in) \in R_{i_o}$; also, add the rule (t, out) to the set R_{i_o}. It is obvious that the obtained system, Π', works exactly as Π, the only difference being that t accompanies a when entering the output membrane (and then t immediately exits this membrane). Consequently, if the terminal alphabet of Π' is $T = \{b\}$ (while $h(i_o) = b$ and $h(i) = \lambda$, for $i \neq i_o$) then $L(\Pi') = \{b^n \mid n \in N(\Pi)\}$. In this way, if we denote by $1RE$ the family of one-letter recursively enumerable languages, then from the theorems given in Sections 4.2 and 4.3 we obtain:

Corollary 4.4.1. $LOP_2(sym_2, anti_2) = LOP_5(sym_2, anti_0) = LOP_3(sym_4, anti_0) = LOP_2(sym_5, anti_0) = LOP_3(psym_2, anti_0) = LOP_3(fsym_2, anti_0) = 1RE$.

Note that these results do not improve the assertion from Theorem 4.4.1, but imply that all families mentioned in Corollary 4.4.1 contain languages outside the family CS.

The passing from languages over the one-letter alphabet to languages over an arbitrary alphabet is not trivial, but it can be done (at least in certain cases) by following the idea from the proof of Theorem 3.7.3 – see Theorem 4.4.4 below. Actually, the number of symbols appearing in the strings of a language obviously induces an infinite hierarchy with respect to the number of membranes used by a system able to generate

that language: each symbol should correspond to a membrane, hence $L \in LOP_*(psym_*, panti_*) \cup LOP_*(fsym_*, fanti_*)$ implies $card(alph(L)) \leq m$. This observation deserves to be formulated as a theorem, because it is the first one up to now of this type.

Theorem 4.4.2. *The hierarchies $LOP_m(\alpha sym_p, \beta anti_q), m \geq 1$, on the number of membranes are infinite for all combinations of $\alpha, \beta \in \{p, f\}$ or missing, and for all p, q.*

As we have already mentioned, it is not known at this moment whether or not any of the families $LOP_*(sym_p, anti_q)$ equals RE. A partial result in this respect is the following one: the regular languages can be generated even in the case when no antiport rule is used (and no promoter or inhibitor).

Theorem 4.4.3. $REG \subset LOP_*(sym_2, anti_0)$.

Proof. We only have to prove the inclusion, since its properness is known from Theorem 4.4.1.

Consider a (non-deterministic) finite automaton $A = (Q, T, s_0, F, R)$, with the rules from R given in the form $r : sa \to s'$, for $s, s' \in Q, a \in V$. Assume that we have k rules in R, labelled with $r_1, \ldots, r_k$.

We construct the system (of degree $k + 4$)

$$
\begin{aligned}
\Pi &= (O, t, T, h, \mu, w_1, \ldots, w_{k+4}, E, R_1, \ldots, R_{k+4}),\\
O &= Q \cup \{c, d, t\},\\
h &: \{1, 2, \ldots, k+4\} \longrightarrow T \cup \{\lambda\}, \text{ defined by } h(4+i) = a\\
&\quad \text{for } r_i : sa \to s', 1 \leq i \leq k,\\
&\quad \text{and } h(1) = h(2) = h(3) = h(4) = \lambda,\\
\mu &= [_1[_2\ ]_2[_3\ ]_3[_4\ ]_4[_5\ ]_5 \cdots [_{k+4}\ ]_{k+4}]_1,\\
w_1 &= c,\\
w_2 &= t,\\
w_3 &= w_4 = \lambda,\\
w_{4+i} &= d, \text{ for all } i = 1, 2, \ldots, k,\\
E &= Q,\\
R_1 &= \{(c, out)\} \cup \{(cs, in) \mid s \in Q\},\\
R_2 &= \{(cs_0, in),\ (ts_0, out)\},\\
R_3 &= \{(ts, in) \mid s \in F\},\\
R_4 &= \{(s, in) \mid s \in Q\} \cup \{(t, in),\ (t, out),\ (d, in),\ (d, out)\},\\
R_{4+i} &= \{(cs', in),\ (ts, in),\ (ts', out) \mid r_i : sa \to s'\}\\
&\quad \cup \{(c, out),\ (td, out)\}, \text{ for all } i = 1, 2, \ldots, k.
\end{aligned}
$$

We start with the traveller object in membrane 2 and the "carrier" object c in membrane 1. The carrier will bring states of A from the environment

and place them in membranes $5, 6, \ldots, k+4$. Arbitrarily many states are brought into these membranes, but only copies of state s' are introduced in membrane $4+i$ for $r_i : sa \to s'$. If a state s remains in the skin membrane and the carrier c again exits the system, then s is immediately moved into the "garbage collector" membrane 4.

At any moment when the state s_0 is brought into the system, it can be introduced into membrane 2, by the rule $(cs_0, in) \in R_2$; in this way, c ends its work (it remains forever in membrane 2), and the traveller can leave this membrane together with s_0. Note that only s_0 is present in the skin membrane – and from now on, only one state will be available in the skin membrane.

Assume that in the skin membrane we have, together with t, a state s such that $r_i : sa \to s'$ appears in R. If s enters membrane 4, then t will pass forever through membrane 4 by using the rules $(t, in), (t, out) \in R_4$, hence the computation will never end. If we use a rule $(ts, in) \in R_{4+i}$, then at the next step, the rule $(ts', out) \in R_{4+i}$ can be applied, and this simulates the use of the rule r_i in the automaton A: the traveller has passed through membrane $4+i$, which, by the weak coding h, provides the object a parsed by rule r_i, while the state s has been replaced by s'. The work of A is simulated in this manner until reaching a final state, $s \in F$. At that moment, the rule $(ts, in) \in R_3$ can be used, and the computation stops.

If we do not have sufficient copies of states s' in membranes $4+i$ associated with rules $r_i : sa \to s'$, then, instead of the rule $(ts', out) \in R_{4+i}$, we have to use the rule $(td, out) \in R_{4+i}$ and the computation will never stop: the object d will pass forever back and forth through membrane 4, by using the rules $(d, in), (d, out) \in R_4$. Similarly, if in the skin membrane we have a state s such that there is no rule $sa \to s'$ in R, then the object t has to use forever the rules (t, in) and $(t, out) \in R_4$, and the computation will never stop.

Thus, we can simulate in Π all correct parsings of A and, conversely, each halting computation in Π corresponds to a correct parsing in A. That is, $L(A) = L(\Pi)$, which concludes the proof. □

At the cost of using antiport rules of an arbitrary weight (depending on the number of symbols appearing in the strings of our language), as well as promoters or inhibitors, we can generate all recursively enumerable languages.

Theorem 4.4.4. $LOP_*(psym_2, panti_*) = LOP_*(fsym_2, fanti_*) = RE$.

Proof. Let us consider a language $L \in RE, L \subseteq T^*$, with $T = \{a_1, \ldots, a_k\}$. We interpret the strings in T^+ as numbers in base $k+1$, in the same way as used in the proof of Theorem 3.7.3, hence with $x \in T^+$ we associate the numerical value $val(x)$. Take a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the Z-normal form, such that $L(G) = val(L)$. With the standard notations for elements of G already used many times, we construct the system (of degree $k+4$)

$$\Pi = (O, t, T, h, \mu, w_1, w_2, \ldots, w_{k+4}, E, R_1, \ldots, R_{k+4}),$$

$$
\begin{aligned}
O &= N_1 \cup \{X' \mid X \in N_1\} \cup N_2 \cup \{A' \mid A \in N_2\} \\
&\quad \cup \{e_i \mid 1 \le i \le k\} \cup \{a, b, c, d, e, g, t, Z, Z', \#\}, \\
h &: \{1, 2, \ldots, k+4\} \longrightarrow T \cup \{\lambda\} \text{ defined by } h(4+i) = a_i, \\
&\quad \text{for } 1 \le i \le k, \\
&\quad \text{and } h(1) = h(2) = h(3) = h(4) = \lambda, \\
\mu &= [_1[_2[_3[_4\ ]_4]_3[_5\ ]_5[_6\ ]_6 \cdots [_{k+4}\ ]_{k+4}]_2]_1, \\
w_1 &= X_{init} A_{init}, \\
w_2 &= gt, \\
w_{2+i} &= \lambda, \text{ for } 1 \le i \le k+2, \\
E &= V - \{t\}, \\
R_1 &= \{(XA, out; Yx, in) \mid (X \to Y, A \to x) \in M\} \\
&\quad \cup \{(X, out; \#, in) \mid X \in N_1\} \\
&\quad \cup \{(X, out; A'X', in),\ (X', out; Ye, in), \\
&\qquad (eA', out),\ (A'A, out; \#, in) \mid (X \to Y, A \to \#) \in M\} \\
&\quad \cup \{(\#, out; \#, in),\ (Z, out; Zb, in),\ (Z, out; cd, in)\} \\
&\quad \cup \{(Z, out; Ze_i, in) \mid 1 \le i \le k\}, \\
R_2 &= \{(a, in)|_g,\ (c, in),\ (g, out; d, in), \\
&\qquad (b, out; a, in),\ (a^{k+1}, out; b, in)|_c\} \\
&\quad \cup \{(ca^i, out; e_i, in) \mid 1 \le i \le k\}, \\
R_3 &= \{(da, in)\}, \\
R_4 &= \{(da, in),\ (da, out)\}, \\
R_{4+i} &= \{(te_i, in),\ (t, out)\}, \text{ for } i = 1, 2, \ldots, k.
\end{aligned}
$$

Note that there are only two conditionally used rules, the symport rule $(a, in)|_g$ and the antiport rule $(a^{k+1}, out; b, in)|_c$, both in R_2; the former rule cannot be used after removing the object g from membrane 2, and the latter rule cannot be used before introducing the object c in membrane 2.

Roughly speaking, the system Π works as follows. Membranes 1 and 2 are used in order to generate the language $val(L)$. When the generation of a string $a^n = val(x)$ is finished, hence the symbol Z is produced, we start to "translate" the number n in such a way to recover the string x ("codified" by the labels of the membranes $5, 6, \ldots, k+4$ associated with its symbols). This phase is triggered by the appearance of the object c, the promoter of the rule $(a^{k+1}, out; b, in)|_c$. The recovering of the string x from $val(x)$ is mainly done by the rules from R_2, which repeatedly divide by $k+1$ the current number of copies of the object a, provide the remainder in the form of an object e_i, and the quotient in the form of the number of copies of the object b (at the next step, each b is replaced by a, making possible the iteration of the process). The object e_i indicates to the traveller the right membrane to enter, thus adding the corresponding symbol a_i to the generated string. The

process of division continues until exhausting the objects a, hence completely reconstructing the string x.

Here are the details of this process. We start with $X_{init}A_{init}$ in the skin membrane (and gt in membrane 2). Assume that at any moment we have here a multiset (described by a string) of the form Xw, for some $X \in N_1, x \in N_2^*$ (each copy of a is immediately sent to membrane 2, by the rule $(a, in)|_g \in R_2$, hence we ignore the occurrences of a in this phase).

A matrix $(X \to Y, A \to x) \in M$ is simulated by the antiport rule $(XA, out; Yx, in) \in R_1$.

A matrix $(X \to Y, A \to \#) \in M$, with the second rule used in the appearance checking manner, is simulated in a slightly more complex manner: first, X gets out and brings into the system the objects A' and X' (by the rule $(X, out; A'X', in) \in R_1$); at the next step, if A is present in the skin membrane, then the rule $(A'A, out; \#, in) \in R_1$ brings into the system the trap object $\#$, which will pass forever through the skin membrane by the rule $(\#, out; \#, in) \in R_1$, hence the computation will never end. If A is not present, then A' waits one step; at the same time, the rule $(X', out; Ye, in) \in R_1$ brings the object e into the system (and completes the simulation of the rule $X \to Y$). At the next step, the objects A' and e leave the system. In this way, the matrix is correctly simulated.

If, at any time, we cannot simulate a matrix as above, although an object $X \in N_1$ is present in the skin membrane, then the rule $(X, out; \#, in) \in R_1$ will bring the trap object $\#$ into the system and the computation will never finish.

In this way, any derivation in G can be correctly simulated, and, conversely, all correct (that is, not introducing the trap object $\#$) computations in Π correspond to correct derivations in G. When the derivation is terminal, that is the object Z is introduced, we pass to the second phase of the work of Π.

First, the object Z will bring into the system arbitrarily many copies of the objects b and $e_i, 1 \leq i \leq k$. The work of Z ends by introducing the objects c and d into the system (by the rule $(Z, out; cd, in) \in R_1$; note that Z is left in the environment). The objects c and d immediately enter membrane 2, d in exchange with g, hence from now on the rule $(a, in)|_g$ can no longer be used.

In the presence of c, the rule $(a^{k+1}, out; b, in)|_c$ can be used. Because of the maximal parallel manner of using the rules, the application of this rule means dividing the number of occurrences of a present in membrane 2 by $k+1$. The result is obtained as the number of occurrences of the object b. At the same time, the remainder is obtained in the form of an object e_i. Specifically, if we had n copies of a, such that $n = m(k+1) + i$, for some $m \geq 0$ and $1 \leq i \leq k$ (the remainder is never zero), then in membrane 2 we get m copies of b and one copy of e_i. The correctness of this operation is ensured by the fact that if any copy of a remains unused after the application of the rules

$(a^{k+1}, out; b, in)|_c$ (maximal parallelism) and $(ca^i, out; e_i, in)$ (the choice of the maximal i), then the rule $(da, in) \in R_3$ can be used, and after that the pair da will pass forever through membrane 4, by the rules $(da, in), (da, out) \in R_4$, and the computation will never end. The same result is obtained if we do not have sufficient copies of b or e_i in the skin membrane: any remaining copy of a will be used by the rule $(da, in) \in R_3$ and the computation will never halt.

At the next step, in the presence of e_i, the traveller enters the corresponding membrane $4 + i$. Because the remainder i corresponds to the symbol a_i from the string of L which is "parsed" at the present step, we get the same object via the weak coding h. Simultaneously, by means of the rule $(b, out; a, in) \in R_2$, we exchange all objects b from membrane 2 with objects a from the skin membrane (note that we have enough copies of a in the skin membrane, hence the exchange of objects b is complete, and that we can bring into membrane 2 exactly as many copies of a as we have copies of b, because the rule $(a, in)|_g$ is no longer applicable); in this way the process can be iterated, and the division of the current number (m in the above notation) by $k + 1$ can be repeated. At the moment of division, t exits membrane $4 + i$, hence it is available for the next step.

We continue in this way until exhausting the number n we started with, in the sense that we reach a stage where we have at most k copies of a in membrane 2; this means that only a rule $(ca^i, out; e_i, in) \in R_2$ is used. The object c will return to membrane 2, but no further rule can involve it, the object e_i will enter membrane $4 + i$ together with t, the traveller exits again, and the computation stops. The trace of t, with all occurrences of label 2 ignored (removed by h), is exactly the string $x \in L$ whose value was initially codified by the number of a's produced by the derivation in G. In conclusion, $L(\Pi) = L$, and this proves the inclusion $RE \subseteq LOP_*(psym_2, panti_*)$.

The previous construction can be easily modified in order to use inhibitors. Specifically, we use the object g, placed initially in membrane 2, in the forbidding mode: we replace the rule $(a, in)|_g \in R_2$ with $(a, in)|_{\neg d}$, and we also replace the permitting context rule $(a^{k+1}, out; b, in)|_c \in R_2$ with the forbidding context rule $(a^{k+1}, out; b, in)|_{\neg g}$. Thus, as long as g is present in membrane 2, the rule $(a^{k+1}, out; b, in)|_{\neg g}$ cannot be used. After finishing the simulation of a derivation in G, the object Z introduces the objects c and d into the system, and d will entail the removal of g from membrane 2 (also, it will prevent the entrance of any further copy of a). From now on we work (in the forbidding mode) in the same way as we have proceeded in the previous proof (in the permitting mode). If Π' is the system obtained by modifying the system Π as above, then we get $L(\Pi) = L(\Pi') = L$. Consequently, we have also obtained the inclusion $RE \subseteq LOP_*(fsym_2, fanti_*)$, which concludes the proof. □

The weight of symport rules used in the previous proof is minimal, but we use antiport rules of an arbitrary weight (actually, depending on the number of symbols that appear in the considered language); also, there are one sym-

port rule and one antiport rule which are used conditionally. Can the results from Theorem 4.4.4 be improved from either of these two points of view? (Problem Q12.)

4.5 Systems with Carriers

We now consider another class of membrane systems where the computation is done by communication only, by carrying objects from one region to another one. To implement this idea we separate the objects into two classes, *carriers* and *passengers* (carrying objects and carried objects). The biological motivation is twofold: it abstracts the work of carrier proteins assisting molecules to pass through membranes, as well as the work of *vectors* used in gene cloning (in nature, two frequently used vehicles that transport genes into a host cell are plasmids – circular sequences of DNA nucleotides – and bacteriophages).

In our framework, we will consider rules which attach passengers to carriers, move the conglomerates so obtained through membranes, and disassemble them, thus releasing the carriers and the carried objects, making them available for further attachments and moves.

Technically, a membrane system with carriers is a construct

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, R_e, i_o),$$

where:

1. O is the alphabet of *objects*.
2. C is the alphabet of *carriers*.
3. μ is a membrane structure with m membranes (injectively labelled by positive integers $1, 2, \ldots, m$).
4. $w_1, \ldots, w_m$ are strings over $O \cup C$, representing the multisets of objects and carriers initially present in the m regions of the system.
5. $R_1, \ldots, R_m$ are finite sets of rules, associated with the m regions of μ. Each rule has one of the following forms:
   - $ca_1 \ldots a_k \to \langle ca_1 \ldots a_k \rangle$, for $c \in C, a_1, \ldots, a_k \in O, k \geq 1$ (*attaching rules*);
   - $\langle ca_1 \ldots a_k \rangle \to ca_1 \ldots a_k$, for $c \in C, a_1, \ldots, a_k \in O, k \geq 1$ (*detaching rules*);
   - $\langle ca_1 \ldots a_k \rangle \to in$, for $c \in C, a_1, \ldots, a_k \in O, k \geq 0$ (*move-in rules*);
   - $\langle ca_1 \ldots a_k \rangle \to out$, for $c \in C, a_1, \ldots, a_k \in O, k \geq 0$ (*move-out rules*);

   in each of these rules, c is the carrier and $a_1, \ldots, a_k$ are the passengers.
6. R_e is a finite set of *rules*, of the first three forms above, placed in the environment of the system.

7. $i_o \in \{1, \ldots, m\}$ is an elementary membrane of μ (the output membrane).

The meaning of the rules in $R_i, 1 \leq i \leq m$, and in R_e is as follows. By an attaching rule, the objects $a_1, \ldots, a_k$ get attached to the carrier c (a multiset of objects plus one copy of a carrier, all of them from region i – this is the environment in the case of R_e), yielding a *conglomerate* which behaves as a single body; the whole multiset is represented by a string and the fact that we have a single body is indicated by enclosing this string in angular brackets. A detaching rule performs the opposite operation, separating all objects and the carrier from a conglomerate $\langle ca_1 \ldots a_k \rangle$. By moving-in and moving-out rules we can move conglomerates through membranes: *in* indicates that we have to go to one of the adjacent inner membranes, non-deterministically choosing it (if there is no lower level membrane, then the rule cannot be applied); *out* indicates the move outside the current membrane. Note that carriers alone can pass through membranes, but an object can never pass through a membrane alone (without being attached to a carrier). Moreover, an object is never modified by these rules. By carrier rules we can also send objects out of the system and, by using the rules from R_e, we can carry objects into the system, from the environment. At the beginning of a computation, we assume that the environment contains arbitrarily many copies of each object from O (but no carrier; they are present only inside the system, in a finite number of copies each, as specified by the initial multisets $w_1, \ldots, w_m$).

It is important to notice the fact that the rules from the set R_i are associated *with region i*, and *not with membrane i* (as is the case for systems with symport/antiport rules); the attaching–detaching rules make no sense with respect to a membrane, but only with respect to a (multiset of objects from a) region.

The maximum number of objects which can be attached to a carrier (the maximum k in the rules from $R_1, \ldots, R_m, R_e$ as written above) is called the *carrying index* of the system.

The transitions among configurations and the halting computations are defined in the standard manner. The result of a halting computation is the number of objects (carriers are ignored) present in the output membrane in the halting configuration. The set of all such numbers is denoted by $N(\Pi)$ and the family of all sets $N(\Pi)$, computed by systems Π of degree at most $m \geq 1$, using at most $p \geq 1$ carriers, and with the carrying index not exceeding $k \geq 1$, is denoted by $NOP_m(carr_{p,k})$; if any one of the parameters m, p, k is not limited, then we replace it with $*$.

Similar to "computing by communication" by means of symport/antiport rules, "computing by carrying" (which is also a form of computing by communication) proves to be very powerful; the universality can be reached again, even with a small number of membranes, a small number of carriers, and a small carrying index.

Theorem 4.5.1. $NOP_2(carr_{3,3}) = NRE$.

Proof. Let $G = (N, \{a\}, S, M, F)$ be a matrix grammar with appearance checking in the binary normal form, in the standard notation. Consider two new symbols, f, g. We replace the matrices $(X \to \lambda, A \to x)$ from M with $(X \to f, A \to x)$; then, we replace all matrices $(X \to \alpha, A \to x)$, $\alpha \in N_1 \cup \{f\}$, $x \in (N_2 \cup \{a\})^*$, by $(X \to \alpha, A \to xg^i), i = 0, 1, 2$, such that $|xg^i| = 2$. We will still denote by $(X \to \alpha, A \to x)$ the matrices so obtained and by G the grammar so obtained, also preserving the labels of matrices.

We construct the system (of degree 2)

$$\Pi = (O, C, [_1[_2\ ]_2]_1, w_1, w_2, R_1, R_2, R_e, 2),$$

with the following components:

$$\begin{aligned}
O &= N \cup \{a, d, e, f, g\} \cup \{b_i, d_i \mid 1 \le i \le n\},\\
C &= \{c, c', c''\},\\
w_1 &= c'c''d,\\
w_2 &= cb_1 \ldots b_n d_1 \ldots d_n X_{init} A_{init},
\end{aligned}$$

and the sets of rules constructed in the following way.

1. For each matrix $m_i : (X \to \alpha, A \to \alpha_1\alpha_2), 1 \le i \le k$, the following rules are in R_2:
   $cb_id_iX \to \langle cb_id_iX\rangle$,
   $\langle cb_id_iX\rangle \to out$,
   $\langle c'd_id\rangle \to c'd_id$,
   $c'd_idA \to \langle c'd_idA\rangle$,
   $\langle c'd_idA\rangle \to out$,
   $\langle cb_i\alpha_1\alpha_2\rangle \to cb_i\alpha_1\alpha_2$,
   $\langle c'd_i\alpha e\rangle \to c'd_i\alpha e$;
   the following rules are in R_1:
   $\langle cb_id_iX\rangle \to cb_id_iX$,
   $cb_iX \to \langle cb_iX\rangle$,
   $\langle cb_iX\rangle \to out$,
   $c'd_id \to \langle c'd_id\rangle$,
   $\langle c'd_id\rangle \to in$,
   $\langle c'd_idA\rangle \to c'd_idA$,
   $c'd_iA \to \langle c'd_iA\rangle$,
   $\langle c'd_iA\rangle \to out$,
   $\langle cb_i\alpha_1\alpha_2\rangle \to in$,
   $\langle c'd_i\alpha e\rangle \to in$;
   and the following rules are in R_e:
   $\langle cb_iX\rangle \to cb_iX$,
   $cb_i\alpha_1\alpha_2 \to \langle cb_i\alpha_1\alpha_2\rangle$,
   $\langle cb_i\alpha_1\alpha_2\rangle \to in$,
   $\langle c'd_iA\rangle \to c'd_iA$,

$c'd_i\alpha e \to \langle c'd_i\alpha e\rangle$,
$\langle c'd_i\alpha e\rangle \to in$.

2. For each matrix $m_i : (X \to Y, A \to \#), k+1 \leq i \leq n$, the following rules are in R_2:
   $cb_id_iX \to \langle cb_id_iX\rangle$,
   $\langle cb_id_iX\rangle \to out$,
   $\langle c'd_i\rangle \to c'd_i$,
   $c'd_iA \to \langle c'd_iA\rangle$,
   $\langle c'd_iA\rangle \to out$,
   $\langle cb_iYe\rangle \to cb_iYe$;
   the following rules are in R_1:
   $\langle cb_id_iX\rangle \to cb_id_iX$,
   $cb_iX \to \langle cb_iX\rangle$,
   $\langle cb_iX\rangle \to out$,
   $c'd_i \to \langle c'd_i\rangle$,
   $\langle c'd_i\rangle \to in$,
   $\langle c'd_iA\rangle \to in$,
   $\langle cb_iYe\rangle \to in$;
   and the following rules are in R_e:
   $\langle cb_iX\rangle \to cb_iX$,
   $cb_iYe \to \langle cb_iYe\rangle$,
   $\langle cb_iYe\rangle \to in$.
3. For all $i = 1, 2, \ldots, n$ and for each $\alpha \in N_2$, the following rules are in R_2:
   $cfb_i \to \langle cfb_i\rangle$,
   $\langle cfb_i\rangle \to out$,
   $cfd_i \to \langle cfd_i\rangle$,
   $\langle cfd_i\rangle \to out$,
   $cf\alpha \to \langle cf\alpha\rangle$,
   $\langle cf\alpha\rangle \to out$,
   $c''b_i \to \langle c''b_i\rangle$,
   $\langle c''b_i\rangle \to out$,
   $c''d_i \to \langle c''d_i\rangle$,
   $\langle c''d_i\rangle \to out$,
   $c''\alpha \to \langle c''\alpha\rangle$,
   $\langle c''\alpha\rangle \to out$;
   and the following rules are in R_1:
   $\langle cfb_i\rangle \to cfb_i$,
   $\langle cfd_i\rangle \to cfd_i$,
   $\langle cf\alpha\rangle \to in$,
   $\langle c''b_i\rangle \to in$,
   $\langle c''d_i\rangle \to in$,
   $\langle c''\alpha\rangle \to in$.
4. Also, the following rules are in R_2:

$c'e \to \langle c'e \rangle$,
$\langle c'e \rangle \to out$,
$\langle cf \rangle \to cf$,
$cfg \to \langle cfg \rangle$,
$\langle cfg \rangle \to out$,
$c''f \to \langle c''f \rangle$,
$\langle c''f \rangle \to out$,
$c''g \to \langle c''g \rangle$,
$\langle c''g \rangle \to out$;

the following rules are in R_1:

$\langle c'e \rangle \to c'e$,
$cf \to \langle cf \rangle$,
$\langle cf \rangle \to in$,
$\langle cfg \rangle \to cfg$,
$\langle c''g \rangle \to in$,
$\langle c''f \rangle \to out$,
$\langle c'' \rangle \to out$,
$\langle c'' \rangle \to in$;

and the following rules are in R_e:

$\langle c''f \rangle \to c''f$,
$\langle c'' \rangle \to in$.

5. For all $\alpha \in N_1 \cup \{f\}$ we also introduce in R_2 the rules:

$c'\alpha \to \langle c'\alpha \rangle$,
$\langle c'\alpha \rangle \to out$,

and the following rule in R_1:

$\langle c'\alpha \rangle \to in$.

This system works as follows.

In the initial configuration we have objects and carriers both in region 2 and in region 1, but the computation starts in region 2 (the carrier c'' from region 1 just goes to the environment and comes back, by using the rules $\langle c'' \rangle \to out$ from R_1 and $\langle c'' \rangle \to in$ from M_e). Assume that at some moment in region 2 we have a multiset consisting of one copy of an object $X \in N_1$, some copies of objects from N_2 (initially, we have here the multiset $X_{init}A_{init}$), possibly copies of the object a, as well as the objects $b_1, \ldots, b_n, d_1, \ldots, d_n$.

Assume that in region 2 we use the attaching rule $cb_id_iX \to \langle cb_id_iX \rangle$ for some $1 \leq i \leq k$. This will start the simulation of the matrix $m_i : (X \to \alpha, A \to \alpha_1\alpha_2)$, which is done in the following way. The conglomerate $\langle cb_id_iX \rangle$ exits membrane 2 and gets detached in membrane 1. The object b_i together with c and X will exit the system, the object d_i will go inside membrane 2 together with the carrier c' and the object d, which was present in region 1 from the beginning of the computation; c' and d_i will also bring out a copy of A. The conglomerate $\langle c'd_idA \rangle$ gets detached in membrane 1, while c', d_i, A will go together into the environment, and d remains in region 1 for further use.

From the environment, c (together with b_i) will bring into the central membrane the objects α_1, α_2, while c' (and d_i) will also bring into membrane 2 the object α together with the auxiliary object e. Note that α is brought in later than α_1, α_2, hence the simulation of using another matrix cannot be started before completing the simulation of the matrix m_i. The carrier c remains in region 2, and the carrier c' returns to region 1, by making use of the object e (by using rules introduced in group 4).

Here is an important detail: if there is no copy of A in region 2, then the carrier c' waits here until c returns from the environment. In the presence of any symbol α from $N_1 \cup \{f\}$, the carrier c' uses a rule $c'\alpha \to \langle c'\alpha \rangle$, and the conglomerate $\langle c'\alpha \rangle$ passes forever back and forth through membrane 2, in this way preventing the halting of the computation. These operations are performed by using the rules from group 5. Consequently, the simulation of the matrix should be complete: both of its rules must be simulated.

The simulation of a matrix $m_i : (X \to Y, A \to \#)$, $k+1 \leq i \leq n$, is performed in the following way. Again, the carrier c brings into region 1 the objects b_i, d_i, together with X. The carrier and the objects b_i, X exit the system, and d_i goes back to the inner membrane, together with the carrier c'. If A is present in region 2, then the rules $c'd_iA \to \langle c'd_iA \rangle$, $\langle c'd_iA \rangle \to out$ from region 2, and $\langle c'd_iA \rangle \to in$ from region 1 will be used forever. If A is not present, then c' waits in region 2 until c comes back, together with the objects Y and e; then c' returns to region 1 together with e.

Note that when waiting in region 2, the carrier c' cannot be attached to an object from $N_1 \cup \{f\}$, because no such object is present; when a symbol from N_1 is brought back by the carrier c, and e is present, we can continue the computation without entering a cycle.

Therefore, in both cases, the simulation of matrices is correct and we return to a configuration where all objects b_i, d_i, and the carriers c, c' are in the same regions as they were at the beginning of the computation. The fact that we have new copies of the object e in region 1 is of no importance, because no rule can be applied to this object here and, moreover, the output of a computation is read in membrane 2.

When the object f is brought into the inner membrane, this means that no symbol from N_1 is present and hence the derivation in G should be terminal. In the presence of f, the carrier c carries all objects b_i, d_i, g to region 2 (the computation is not terminated as long as at least one symbol b_i, d_i is present in region 2) and it also checks whether or not the derivation is terminal. If the check yields a negative result, then the rules $\langle cf\alpha \rangle \to out$ from membrane 2 and $\langle cf\alpha \rangle \to in$ from region 1 will be used forever.

In order to remove also the object f from region 2 we use the carrier c''. During the whole computation, c'' has just passed across membrane 1 by using the rules $\langle c'' \rangle \to out$ and $\langle c'' \rangle \to in$ from region 1 and the environment, respectively. At any moment, this carrier can also go to membrane 2, by using the rule $\langle c'' \rangle \to in$ from region 1. Here c'' can be attached to f and brings

it to the environment; then it eventually returns to region 2. If any symbol $b_i, d_i, 1 \leq i \leq n$, or $\beta \in N_2 \cup \{g\}$ is present here, then the computation never stops, because of the rules in group 3 of the form $\langle c''\beta\rangle \to out, \langle c''\beta\rangle \to in$ which are present in regions 2 and 1, respectively. If no such symbol is present in region 2, then the computation stops. Thus, we cannot finish the computation before bringing the carrier c'' into region 2; if this is done prematurely (while non-terminals from N_2 or symbols b_i, d_i are still present), then the computation will never halt.

Consequently, $N(\Pi) = length(L(G))$. Because the largest conglomerates from our system have four elements ($\langle cb_id_iX\rangle, \langle cb_i\alpha_1\alpha_2\rangle$, $\langle cb_iYe\rangle$, etc.), we get $length(L(G)) \in NOP_2(carr_{3,3})$, and this concludes the proof. □

At the cost of the unbounded increase in the number of carriers, we can reduce the carrying index to two. The idea is to use as many carriers as matrices in the starting grammar: instead of controlling the work of carriers by objects b_i, d_i as in the previous proof, we attach the subscript i to the carriers, and in this way we save one carried passenger, hence we decrease the carrying index by one. We skip the details and only mention the result.

Corollary 4.5.1. $NOP_2(carr_{*,2}) = NRE$.

4.6 Bibliographical Notes

Membrane systems with the computation done only by transferring objects from one region to another region were first investigated in [160], where Theorem 4.5.1 and its corollary can be found.

Symport/antiport rules were introduced in [179], where Theorem 4.2.1, as well as the equality $NOP_5(sym_2, anti_1)$, are also given. This second result was improved in [150] to the form given in Theorem 4.2.2 (no antiport rule is used). Theorems 4.2.3, 4.2.4, and 4.2.5 are from [149] – the proofs are slightly simplified here. In [149] systems with permitting contexts are also introduced, and it is proved that $NOP_4(psym_2, anti_0) = NRE$. The improvement to three membranes is given in [150], where also the forbidding context case is considered and Theorem 4.3.1 is given.

Systems with control on the membrane permeability are considered in [180], where it is proved that $NOP_5(sym_2, anti_0, \delta, \tau) = NRE$, but the result is now covered by Theorem 4.2.2, hence the influence of actions δ, τ should be further investigated (that is why we have formulated this as an open problem, Problem Q10). A generalization which we have not recalled here is also considered in [180]: besides the usual symport/antiport rules, to allow rules for changing the names of objects, of the forms (a, in, b) and (a, out, b), with the meaning that when a enters/exits the membrane, it becomes b. Making use of such rules one can prove that $NOP_3(sym_2, anti_0, rename, \delta, \tau) = NRE$.

Section 4.4 is based on [99].

5. Structuring the Objects

In a living cell, many objects can be considered as being *atomic*, with no internal structure, but many others have a structure. In many cases, this structure can be described by a string (as happens for DNA, RNA, proteins, etc.), and this motivates the introduction of string–objects in membrane systems.

Working with strings imposes some specific features, or makes possible the relaxation of some features used in membrane systems. For instance, string–objects have to evolve by string processing rules, as known from formal language theory, DNA computing, etc. The first candidates are the operations of rewriting, splicing, insertion–deletion, and context adjoining. We will consider all these operations in the subsequent sections. Then, a string can be rewritten in a sequential manner (as in a Chomsky grammar) or in a parallel manner (as in a Lindenmayer system); an intermediate case is to rewrite as many symbols as possible, but not necessarily all of them. In the parallel rewriting we can obtain systems with a triple parallelism: at the level of symbols from each string–object, at the level of string–objects from each region, at the level of the regions of a system.

A problem appears with the way of defining the output of a computation. This can be *the set* of strings sent out of the system, the set of strings collected in a given output membrane (in both cases, a language), or it can be *the number* of strings sent out or stored in the output membrane during the computation. If we consider the set of strings sent out, then we do not need to work with halting computations, since a string sent out remains unchanged even if the system continues the computation, processing other strings.

This is not necessarily true when we consider strings collected in an output membrane; in this case either we can use halting computations, or we can use a squeezing criterion for selecting desired strings, for example by means of a terminal alphabet, as in Chomsky grammars. When dealing with an output in the form of a language, it is not necessary to consider multiplicities of strings present in the regions of the system – but it is possible. On the contrary, when the result of a computation is a number we have to consider both halting computations and multisets of strings, otherwise the respective numbers are not defined. Moreover, when working with multisets of strings we need operations which are able to modify the number of strings present in a region. Rewriting, for instance, takes a string and delivers another string;

splicing takes two strings and delivers two strings. The total number of strings is not changed, hence further operations should be used, such as splitting a string into two parts, replicating a string, and so on. There are a lot of possibilities, a small "jungle", to explore. (And a lot of questions to address, as we have several language hierarchies at hand for comparing the power of our devices with classic devices in formal language theory: Chomsky families, Lindenmayer families, Marcus contextual families, etc.) Membrane systems with string–objects can be the subject of a whole monograph, so we will recall here only the basic models and we will give only specific properties/results about these models.

5.1 Rewriting Membrane Systems

We start by considering the most natural case from a mathematical point of view, that of rewriting string–objects, in the sequential manner (one rule only is applied to a string at a time), with the result of a computation in the form of the strings sent out of the system at any time, without restricting ourselves to halting computations (the "life of the cell" may last forever, we just collect the strings expelled from the system). From the beginning we introduce both priorities among rules, the membrane thickness actions, as well as the possibility to squeeze the strings we want by means of a terminal alphabet.

Therefore, we deal with systems of the following form:

$$\Pi = (V, T, \mu, M_1, \ldots, M_m, (R_1, \rho_1), \ldots, (R_m, \rho_m)),$$

where V is an alphabet, $T \subseteq V$ (terminal alphabet), μ is a membrane structure with m membranes labelled with $1, 2, \ldots, m$ (1 is always the label of the skin membrane), $M_1, \ldots, M_m$ are finite languages over V (initial string–objects placed in the regions of μ), $R_1, \ldots, R_m$ are finite sets of context-free evolution rules associated with the m regions of μ, and $\rho_1, \ldots, \rho_m$ are partial order relations over $R_1, \ldots, R_m$.

The rules from the sets $R_1, \ldots, R_m$ are of the forms $a \to (v, tar)$ or $a \to (v, tar)\alpha$, where $a \to v$ is a context-free rule over V, that is $a \in V$ and $x \in V^*$, $tar \in \{here, out, in\}$, and $\alpha \in \{\delta, \tau\}$, with the usual meaning: the string produced by using this rule will go to the membrane indicated by *tar* (*here* means that the string remains in the same region, *out* means to send the string out of the region where the rule is applied, and *in* means to go to any directly lower membrane, non-deterministically chosen); sometimes one will also use indications of the form in_j, where j is the label of the membrane where the string should be sent. To any rule we can append the symbols δ, τ, indicating that after using the rule the respective membrane changes its thickness (permeability) in the way discussed in Subsection 3.6.2.

Note that each string which can be rewritten is rewritten by only one rule, in a sequential manner, as in a Chomsky grammar, and not by rewriting all

its symbols, as in a Lindenmayer system (although this is also possible – and we will discuss it later).

A system with $T = V$ is said to be *non-extended*, hence a system of the general form is said to be *extended* (we follow the terminology used in the L systems area).

The language generated by Π is denoted by $L(\Pi)$ and it is defined as follows: we start from the initial configuration of the system and proceed iteratively, by transition steps performed by applying the rules in parallel, to all strings which can be rewritten, obeying the priority relations; all strings over the alphabet T sent out of the system during any step of any computation form the language $L(\Pi)$.

There is a subtle point here: the priority relation is defined at the level of the sets R_i associated with each membrane; in membranes we can have several strings at the same time, considered in the set sense. That is, we start with a set of strings in each region and after rewriting any string, the string disppears and the result of the rewriting is present. Thus, on the one hand, the number of strings present in the system cannot increase (it can decrease by sending strings outside the system); on the other hand it is possible that a rule r_1 is applicable to a string x and thus the rewriting of another string, y, by a rule r_2 with a lower priority than r_1, is not allowed. In this way, the rewriting of a string can be controlled by the form of other strings from the same region. In most of the proofs below we always have only one string in the system, hence this dependence of strings does not occur.

We stress the facts that we do not consider halting computations, and that in each transition each string is processed by one rule only; the parallelism refers here to processing simultaneously all available strings by all applicable rules. Of course, a string which cannot be rewritten remains unchanged and it passes as it is to the next configuration.

We denote by $LSP_m(rw, in, pri, \delta, \tau)$ the family of languages generated by non-extended string–object P systems of degree at most $m \geq 1$, using rewriting rules, the target indications *here, out, in*, priorities, and actions δ, τ. When we use target indications in_j instead of in, we write $LSP_m(rw, tar, pri, \delta, \tau)$; as usual, m is replaced with $*$ if the degree of the systems is not bounded, and pri, δ, τ are removed if the corresponding feature is not used. When using extended systems, the letter E is added in front of $LSP_m(\ldots)$, thus giving families $ELSP_m(\ldots)$. Often, we will write $[E]LSP_m(\ldots)$, with the meaning that the assertion holds both when E is present and when it is not present. As usual, when $m = 1$, the target parameter is omitted.

In order to illustrate the way of working of a rewriting membrane system, we consider an **example** (which also proves that the family $LSP_2(rw, in)$ contains non-context-free languages). Consider the (non-extended) system

$$\Pi = (V, V, \mu, M_1, M_2, R_1, R_2),$$
$$V = \{a, b, c, d, d', e, e'\},$$

$$\begin{aligned}
\mu &= [_1[_2\ ]_2]_1,\\
M_1 &= \{de\},\\
M_2 &= \emptyset,\\
R_1 &= \{d \to (ad'b, here),\ e \to (ce', in),\\
&\qquad d \to (ab, here),\ e \to (c, out)\},\\
R_2 &= \{d' \to (d, out),\ e' \to (e, here)\}.
\end{aligned}$$

Assume that we have a string of the form $a^n db^n c^m e$ in the skin region, with $n, m \geq 0$ (initially we have $n = m = 0$). If we apply the rule $e \to (ce', in)$, then the string $a^n db^n c^{m+1} e'$ is sent to membrane 2, where we apply the rule $e' \to (e, here)$, and the computation halts without sending out any string. Thus, in the skin membrane we have to use both rules $d \to (ad'b, here)$ and $e \to (ce', in)$. The string $a^{n+1} d' b^{n+1} c^{m+1} e'$ is sent to membrane 2. If we use here only the rule $d' \to (d, out)$, then the string $a^{n+1} db^{n+1} c^{m+1} e'$ is sent to region 1, where we can use one of the rules $d \to (ad'b, here)$ or $d \to (ab, here)$, but the computation halts again without producing any output. We have to use both rules from region 2, hence we return the string $a^{n+1} db^{n+1} c^{m+1} e$ to the skin membrane. The process can be iterated, resulting in the simultaneous increase of the number of occurrences of symbols a, b, c, hence producing strings of the form $a^n db^n c^n e, n \geq 0$. If in the skin region we use the rules $d \to (ab, here)$ and $e \to (ce', in)$, then again we cannot exit from the inner membrane. If we use the rule $e \to (c, out)$, then we send out the string $a^n db^n c^{n+1}$ If we first use the rule $c \to (ab, here)$ and after that the rule $e \to (c, out)$, then we get the string $a^{n+1} b^{n+1} c^{n+1}$. Consequently $L(\Pi) = \{a^n b^n c^n \mid n \geq 1\} \cup \{a^n db^n c^{n+1} \mid n \geq 0\}$, which is a non-context-free language. Note that for the extended system $\Pi' = (V, \{a, b, c\}, \mu, M_1, M_2, R_1, R_2)$ we obtain $L(\Pi') = \{a^n b^n c^n \mid n \geq 1\}$.

We now start to investigate the families $[E]LSP_m(rw, in, \ldots)$, mainly in comparison with families in the Chomsky hierarchy.

Theorem 5.1.1. *$LIN \subset LSP_1(rw) \subset CF$, and $ELSP_1(rw) \subseteq CF$.*

Proof. If $G = (N, T, S, R)$ is a linear grammar, then $L(G) = L(\Pi)$, for the system $\Pi = (N \cup T, N \cup T, [_1\ ]_1, \{S\}, \{A \to (uBv, here) \mid A \to uBv \in R, A, B \in N$, and $u, v \in T^*\} \cup \{A \to (u, out) \mid A \to u \in R, A \in N$, and $u \in T^*\})$, hence $LIN \subseteq LSP_1(rw)$.

This inclusion is proper: the system $\Pi = (\{a, b, c\}, \{a, b, c\}, [_1\ ]_1, \{cc\}, \{c \to (acb, here), c \to (acb, out\})$ generates the non-linear language $L(\Pi) = \{a^n cb^n a^m cb^m \mid n, m \geq 0$, and $n + m \geq 1\}$ (this is similar to language L_2 from Section 2.3).

In Theorem 5.1.2 below we will prove that there are context-free languages not in $LSP_*(rw, tar)$; thus, it suffices to prove that each language from $ELSP_1(rw)$ is context-free. Let $\Pi = (V, T, [_1\ ]_1, M_1, R_1)$ be a rewriting P system. We consider the pure context-free grammar $G = (V', S, R)$ with

$V' = V \cup \{a' \mid a \in V\} \cup \{S, q\}$ and the following rules (h is the morphism defined by $h(a) = a', a \in V$):

$$\begin{aligned} R = &\{S \to x \mid x \in M_1\} \\ &\cup \{a \to x \mid a \to (x, here) \in R_1\} \\ &\cup \{a \to h(x)q \mid a \to (x, out) \in R_1\}. \end{aligned}$$

Consider the morphism g defined by $g(a') = g(a) = a$, and $g(q) = \lambda$. Then we have

$$L(\Pi) = g(L(G) \cap \{w_1 q w_2 \mid w_1, w_2 \in (V \cup \{a' \mid a \in V\})^*\}) \cap T^*.$$

The intersection with the regular language $\{w_1 q w_2 \mid w_1, w_2 \in (V \cup \{a' \mid a \in V\})^*\}$ selects from $L(G)$ those strings which were sent outside Π – and only those strings for which exactly only one rule of the form $a \to (x, out)$ was used; the priming of x ensures the fact that its symbols were not further rewritten in G, which is not correct with respect to Π. Then, the intersection with T^* selects the terminal strings with respect to Π. Using the closure properties of the family CF, we conclude that $L(\Pi)$ is a context-free language. □

Although systems with two membranes can generate non-context-free languages (see the example above), the non-extended rewriting P systems without priorities and membrane thickness control cannot generate all context-free languages.

Theorem 5.1.2. $CF - LSP_*(rw, tar) \neq \emptyset$.

Proof. Consider the context-free language $L = \{a^n b^n \mid n \geq 1\}^*$ and assume that there is a system Π of some degree m such that $L = L(\Pi)$. Let us examine the possible ways of generating blocks $a^n b^n$ (a *block* is a maximal string of this form) with an arbitrarily large n in the system Π. Assume that we have rules of the form $a \to (u, tar), u \neq a$, and consider a string w which eventually leads to a string $z = a^{n_1} b^{n_1} \ldots a^{n_k} b^{n_k}$ with an arbitrarily large k. There are such strings w in the computation of z containing arbitrarily many occurrences of a, contributing to arbitrarily many blocks $a^{n_i} b^{n_i}$ of z. We change the computation which leads from w to z by using the rule $a \to (u, tar)$ in such a way as to produce strings with blocks $a^s b^t$ with $s \neq t$, hence strings outside the language L (this is clearly possible, as occurrences of a in w contribute to arbitrarily many blocks $a^{n_i} b^{n_i}$ of z, so we may change the place of using the rule $a \to (v, tar)$ from a block to another block; in this way, both these blocks are affected). The same conclusion holds if we have rules of the form $b \to (u, tar), u \neq b$. It follows that the possible rules are $a \to (a, tar)$, $b \to (b, tar)$, and rules $c \to (u, tar)$ with $c \notin \{a, b\}$. The rules of the first two forms do not increase the string, hence we need at least one rule of the last form. It is also clear that such a rule must be used in the derivation of each block $a^{n_i} b^{n_i}$ with arbitrarily large n_i. Consequently,

there are strings z as above such that during their generation we use strings which contain arbitrarily many occurrences of c. Let us examine now the way a string can leave the system. If the membrane structure μ has the maximal depth, m, then we need m rules of the form $d \to (v, out)$ in order to send a string from the deepest membrane to the environment. Each rule of this type can remove at most one occurrence of c (the rewriting is sequential). This means that a string w which contains at least $m+1$ occurrences of c can be sent out of the system and the string which is obtained outside contains at least one occurrence of c. (The ordering of rules can be changed, unless if one rule introduces a symbol rewritten by another rule and which is not yet present in the string.) In this way we produce a string outside language L, a contradiction which concludes the proof. □

Corollary 5.1.1. *CF is incomparable with all families $LSP_m(rw, \alpha), m \geq 2$, and $LSP_*(rw, \alpha)$, where $\alpha \in \{tar, in\}$.*

Proof. Combine Theorem 5.1.2 with the example discussed above. □

Such results do not hold for the extended case, and this means that using a terminal alphabet for defining the language of a rewriting P system is of real help. The assertion is obvious: for a context-free grammar $G = (N, T, S, R)$ we consider the system

$$\Pi = (N \cup T, T, [_1\ ]_1, \{S\}, \{A \to (x, here),\ A \to (x, out) \mid A \to x \in R\})$$

and we have $L(G) = L(\Pi)$. Consequently, $CF \subseteq ELSP_1(rw)$. The converse inclusion was given in Theorem 5.1.1. We formulate this observation separately because of its importance.

Theorem 5.1.3. *$CF = ELSP_1(rw)$.*

The following result provides an upper limitation/approximation for the families $[E]LSP_*(rw, tar)$.

Theorem 5.1.4. *$LSP_*(rw, tar) \subset MAT^\lambda$, $ELSP_*(rw, tar) \subseteq MAT^\lambda$.*

Proof. We only have to prove the inclusions, as the strictness of the first one follows from Theorem 5.1.2. Let $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, M_m)$ be a rewriting P system of degree m with the skin membrane labelled by 1. For each membrane i different from the skin membrane we denote by $up(i)$ the membrane immediately above it. If any set R_i contains a rule of the form $a \to (u, in_j)$ such that $up(j) \neq i$, then such a rule is never used, hence it can be removed without changing the language generated by Π. We construct the matrix grammar $G = (N, T, S, M)$ with the non-terminal alphabet

$$N = \{a' \mid a \in V\} \cup \{X_i \mid 0 \leq i \leq m\} \cup \{S\},$$

and with the following matrices (h is the morphism defined by $h(a) = a'$ for all $a \in V$):

$(S \to X_i\ h(w))$, for all $w \in M_i, 1 \le i \le m$.
$(X_i \to X_i, a' \to h(u))$, for all $a \to (u, here) \in R_i, 1 \le i \le m$,
$(X_i \to X_j, a' \to h(u))$, for all $a \to (u, in_j) \in R_i, 1 \le i \le m$,
$(X_i \to X_{up(i)}, a' \to h(u))$, for all $a \to (u, out) \in R_i, 2 \le i \le m$.
$(X_1 \to X_0, a' \to u)$, for all $a \to (u, out) \in R_1$,
$(X_0 \to X_0, a' \to a)$, for all $a \in V$,
$(X_0 \to \lambda)$.

The rules from R_i are simulated in the presence of the non-terminal X_i, which controls the communication of strings through the membranes of Π in a correct manner. After simulating a rule which sends the string out of the system, the primed symbols are replaced by their originals. The equality $L(G) = L(\Pi)$ completes the proof. □

Also the converse of the second inclusion from the previous result holds true, even in a stronger form: for only four membranes.

Theorem 5.1.5. $MAT^\lambda \subseteq ELSP_4(rw, in)$.

Proof. Consider a matrix grammar without appearance checking $G = (N, T, S, M)$, in the binary normal form. We replace each matrix $(X \to \lambda,\ A \to x)$ with $(X \to f, A \to x)$, where f is a new symbol. With the standard notations, we construct the system

$$\begin{aligned}
\Pi &= (V, T, [_1[_2[_3[_4\ ]_4]_3]_2]_1, X_{init}A_{init}, \emptyset, \emptyset, \emptyset, R_1, R_2, R_3, R_4),\\
V &= N_1 \cup N_2 \cup T \cup \{f\}\\
&\quad \cup \{X_{i,j} \mid X \in N_1, 1 \le i \le k, 1 \le j \le i\}\\
&\quad \cup \{A_{i,j} \mid A \in N_2, 1 \le i \le k, 1 \le j \le i\},\\
R_1 &= \{A \to (A_{i,1}, in) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}\\
&\quad \cup \{f \to (\lambda, out)\},\\
R_2 &= \{X \to (X_{i,1}, in) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}\\
&\quad \cup \{X_{i,j} \to (X_{i,j+1}, in) \mid X \in N_1, 1 \le j < i \le k\}\\
&\quad \cup \{\alpha \to (\alpha, out) \mid \alpha \in N_1 \cup \{f\}\},\\
R_3 &= \{A_{i,j} \to (A_{i,j+1}, out) \mid A \in N_2, 1 \le j < i \le k\}\\
&\quad \cup \{A_{i,i} \to (x, in) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}\\
&\quad \cup \{\alpha \to (\alpha, out) \mid \alpha \in N_1 \cup \{f\}\},\\
R_4 &= \{X_{i,i} \to (\alpha, out) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}.
\end{aligned}$$

Let us examine the work of Π. Assume that we have a string of the form Xw in membrane 1 (initially, we have the string $X_{init}A_{init}$). The only rules which can be used are those of the form $A \to (A_{i,1}, in)$, for some $1 \le i \le k$, and the string is sent to membrane 2. The only applicable rules from membrane 2 are those of the form $X \to (X_{j,1}, in)$, for some $1 \le j \le k$, and the string is sent to membrane 3. Here the second component of the

subscript of A is increased by one and the string is sent back to membrane 2, where we do the same with the second component of the subscript of the symbol X, and the string is sent to membrane 3 again. The string circulates between membranes 2 and 3 until one of the symbols gets a subscript of the form i,i or j,j. If $i < j$, then this happens for $A_{i,i}$; we use the rule $A_{i,i} \to (x, in) \in R_3$, and the string remains forever in membrane 4. If we have $j < i$, then the string gets stuck in membrane 2. In both cases, we have no output. If $i = j$, then, after using the rule $A_{i,i} \to (x, in) \in R_3$, the rule $X_{i,i} \to (\alpha, out)$ from membrane 4 can be used. This completes the simulation of the matrix $m_i : (X \to \alpha, A \to x)$. The string is sent to membrane 3, then to membrane 2. If we use the rule $\alpha \to (\alpha, out) \in R_2$, then the string is sent to the skin membrane and the process can be iterated. If $\alpha = Y \in N_1$ and in membrane 2 we use the rule $Y \to (Y_{s,1}, in)$, then the string gets stuck in membrane 3, hence we have to return the string to the skin membrane. We continue until introducing the symbol f, when the string can be sent out. It is accepted in the language $L(\Pi)$ only if it is terminal. Consequently, $L(\Pi) = L(G)$ and the proof is complete. □

By combining the relations from Theorems 5.1.4 and 5.1.5, we obtain the following equalities.

Corollary 5.1.2. $MAT = ELSP_4(rw, \alpha) = ELSP_*(rw, \alpha),\ \alpha \in \{tar, in\}$.

It is an *open problem* whether or not this result can be improved; that is, whether or not less than four membranes suffice in order to generate all matrix languages (Problem Q13). In the case of a positive answer, several universality results could probably be improved, because their proofs also include the simulation of matrices without appearance checking, and in several cases one needs four membranes to this aim, as in the previous result.

Besides the fact that we obtain the proof that without priorities we cannot get the universality, the previous result also shows that the hierarchy on the number of membranes collapses at level 4 for extended rewriting P systems (without priority).

Adding a priority relation is sufficient for obtaining computational university, even for systems with only two membranes (hence again the hierarchy on the number of membranes collapses), and without using a terminal alphabet. This result deserves some comments, in relation with the known strict inclusion $ORD^\lambda \subset RE$ pointed out in Subsection 2.3.7. A priority relation on the set of rules increases the power of context-free grammars, but not to the level of Turing computability. However, in our framework this is achieved, a fact that emphasizes the power of distributed computing (of cooperation among computing agents). In some sense, what was needed to get universality was minimal, because two compartments are sufficient.

Theorem 5.1.6. $ORD^\lambda = LSP_1(rw, pri) \subset LSP_2(rw, in, pri) = RE$.

Proof. Let $G = (N, T, S, R, >)$ be an ordered grammar, and construct the system

$$\begin{aligned}
\Pi &= (N \cup T \cup \{X\}, N \cup T \cup \{X\}, [_1\ ]_1, \{XS\}, (R', \rho)),\\
R' &= \{A \to (u, here) \mid A \to u \in R\}\\
&\quad \cup \{A \to (A, here) \mid A \in N\} \cup \{X \to (\lambda, out)\},
\end{aligned}$$

where the priority relation is the extension of $>$ to the corresponding rules from Π, as well as the priority of all rules $A \to (A, here), A \in N$, over the rule $X \to (\lambda, out)$. Thus, a string can leave the system only if it is terminal with respect to G; therefore, $L(G) = L(\Pi)$ and we have the inclusion $ORD^\lambda \subseteq LSP_1(rw, pri)$.

The inclusion $LSP_1(rw, pri) \subseteq ORD^\lambda$ can be proved in the same way as the inclusion $ELSP_1(rw) \subseteq CF$ from Theorem 5.1.1, using the fact that the family ORD^λ is closed under intersection with regular languages and under morphisms.

For the remaining relation $RE \subseteq LSP_2(rw, in, pri)$, we start from a matrix grammar $G = (N, T, S, M, F)$ with appearance checking, in the Z-binary normal form, given in the standard notation. We construct the rewriting P system

$$\Pi = (V, V, [_1[_2\ ]_2]_1, \{X_{init}A_{init}\}, \emptyset, (R_1, \emptyset), (R_2, \rho_2)),$$

with:

$$\begin{aligned}
V &= N \cup T \cup \{X_i \mid X \in N_1, 1 \le i \le n\} \cup \{Z\},\\
R_1 &= \{X \to (Y_i, in),\\
&\qquad Y_i \to (Y, here) \mid \text{for all } m_i : (X \to Y, A \to \alpha), 1 \le i \le n\}\\
&\quad \cup \{Z \to (\lambda, out)\},\\
R_2 &= \{r_i : Y_i \to (Y_i, here),\\
&\qquad r'_i : A \to (x, out) \mid \text{for all } m_i : (X \to Y, A \to x), 1 \le i \le k\}\\
&\quad \cup \{r_A : A \to (A, here) \mid \text{for all } A \in N_2\}\\
&\quad \cup \{p_i : Y_i \to (Y_i, out) \mid \text{for all } m_i : (X \to Y, A \to \#), k+1 \le i \le n\},\\
\rho_2 &= \{(r_i, r'_j) \mid \text{for all } 1 \le i, j \le k, i \ne j\}\\
&\quad \cup \{(r_A, p_i) \mid \text{for all } m_i : (X \to Y, A \to \#), k+1 \le i \le n\}\\
&\quad \cup \{(p_i, r'_j) \mid \text{for all } k+1 \le i \le n, 1 \le j \le k\}.
\end{aligned}$$

The equality $L(G) = L(\Pi)$ holds. Assume that we have a string Xw in the skin membrane, $X \ne Z$. The only rules which can be applied are of the form $X \to (Y_i, in), 1 \le i \le n$, and the string $Y_i w$ is sent to the inner membrane. If $1 \le i \le k$, then the only way to get out of this membrane is by the rule $r'_i : A \to (u, out)$ (all rules $r_B, B \in N_2$, keep the string in membrane 2, no rule p_j with $k+1 \le j \le n$ can be used, the rule $Y_i \to (Y_i, here)$ keeps the string

inside, and all rules r'_j with $1 \leq j \leq k, i \neq j$, have a lower priority than r_i, which is applicable). The string $Y_i w'uw''$, for $w = w'Aw''$, is sent to the skin membrane, where Y_i is replaced by Y, and this completes the simulation of the matrix $m_i : (X \to Y, A \to u)$. If $k+1 \leq i \leq n$, then we can exit membrane 2 only by the rule p_i, but this is possible only if $m_i : (X \to Y, A \to \#)$ and A is not present in the string, otherwise the rule r_A can be used. Note that the rule p_i has priority over all rules $r'_j, 1 \leq j \leq k$, hence we cannot exit by using a rule $r'_j : B \to (v, out)$. In the skin membrane we replace Y_i with Y and in this way we complete the simulation of the matrix m_i with the second rule used in the appearance checking manner. Thus, the only way to introduce the symbol Z is by correctly simulating a derivation with respect to G. When Z appears, this means that we have a string of the form Zy, with $y \in T^*$. We send out the string y, and the computation stops. □

The previous construction can be slightly modified in order to obtain a nice normal form for rewriting P systems with priorities. We say that a system $\Pi = (V, T, \mu, w_1, \ldots, w_m, (R_1, \rho_1), \ldots, (R_m, \rho_m))$ has *global rules* if $R_i = R_j$ for all $1 \leq i, j \leq m$ (that is, all membranes are associated with the same set of rules).

Theorem 5.1.7. *Each language $L \in RE$ can be generated by a (non-extended) rewriting P system of degree 2 with global rules.*

Proof. Consider the system $\Pi = (V, V, \mu, M_1, M_2, (R_1, \rho_1), (R_2, \rho_2))$ constructed in the proof of Theorem 5.1.6 and modify it in order to obtain the following system with global rules:

$$\begin{aligned}
\Pi' &= (V, V, \mu, M_1, M_2, (R, \rho)), \text{ where} \\
R &= R'_1 \cup R'_2, \\
R'_1 &\text{ is } R_1 \text{ with the rule } Y_i \to (Y, here) \text{ replaced with} \\
&\qquad q_i : Y_i \to (Y, in), 1 \leq i \leq n, \\
R'_2 &= R_2 \cup \{r_Y : Y \to (Y, out) \mid Y \in N_1\}, \\
\rho &= \rho_1 \cup \rho_2 \cup \{(r, r') \mid r \in R'_1, r' \in R'_2\} \\
&\quad \cup \{(r_Y, r'_i) \mid Y \in N_2, 1 \leq i \leq k\}.
\end{aligned}$$

There are two ideas here. First, all rules from R'_1 have the target indication in, hence they cannot be applied in membrane 2. Then, all rules from R'_1 have priority over all rules from R'_2 (hence also on the new rules r_Y). Thus, if a string is in membrane 1, then only rules from R'_1 can be applied. If a string is in membrane 2, then only rules from R'_2 can be applied – and they can indeed be applied, because no rule from R'_1 can be applied in membrane 2. Consequently, $L(\Pi) = L(\Pi')$. As Π' is a system with global rules, the proof is complete. □

Here is another normal form for rewriting P systems with priorities. Such a system Π is said to be in the *2-normal form* if each membrane from Π

has associated exactly two different rewriting rules. (Of course, bounding the number of rules in each membrane makes it not necessary to bound the number of membranes in advance.)

Theorem 5.1.8. *Each recursively enumerable language can be generated by a rewriting P system of type (rw, in, pri) in the 2-normal form.*

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the Z-binary normal form and construct the system (of degree $n+1$, where n is the number of two-rule matrices from M)

$$\Pi = (V, V, \mu, M_1, \ldots, M_{n+1}, (R_1, \rho_1), \ldots, (R_{n+1}, \rho_{n+1})),$$

with:

$$\begin{aligned}
V &= N_1 \cup N_2 \cup T \cup \{Z, H, \#\},\\
\mu &= [_1[_2\ ]_2[_3\ ]_3 \cdots [_{n+1}\ ]_{n+1}]_1,\\
M_1 &= \{X_{init} A_{init} H\},\\
M_i &= \emptyset,\ 2 \le i \le n+1,\\
R_1 &= \{H \to (\lambda, in),\ Z \to (\lambda, out)\},\\
\rho_1 &= \emptyset,\\
R_{1+i} &= \{X \to (HY, here),\\
&\qquad A \to (x, out) \mid \text{for } m_i : (X \to Y, A \to x)\},\\
\rho_{1+i} &= \emptyset,\ \text{ for } 1 \le i \le k,\\
R_{1+i} &= \{r_{i,1} : A \to (\#, out),\\
&\qquad r_{i,2} : X \to (HY, out) \mid \text{for } m_i : (X \to Y, A \to \#)\},\\
\rho_{1+i} &= \{(r_{i,1}, r_{i,2})\},\ \text{for } k+1 \le i \le n.
\end{aligned}$$

The simulation of matrices m_i for $1 \le i \le k$ is obvious (if we do not use the rule $X \to (HY, here)$ in membranes $2, \ldots, k+1$, then the string gets stuck in the skin membrane; if the rule $A \to (x, out)$ cannot be applied, then the string gets stuck in the inner membranes); in the case of matrices $m_i, k+1 \le i \le n$, the priority ensures the correct simulation of the appearance checking. A string is sent out only by removing the symbol Z, which ensures that the derivation in G is terminal. □

Let us now also consider the actions δ, τ. As is the case also for symbol–object membrane systems, these actions provide powerful computing techniques, which again lead to universality.

Theorem 5.1.9. $LSP_4(rw, in, \delta, \tau) = RE$.

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the strong binary normal form, in the standard notation. If

$m_i : (X \to \lambda, A \to x)$, then we replace it with $m_i : (X \to f, A \to x)$, where f is a new symbol (we preserve the label of the matrix).

We construct the (non-extended) rewriting P system

$$\Pi = (V, V, \mu, M_1, M_2, M_3, M_4, R_1, R_2, R_3, R_4),$$

with the following components:

$$\begin{aligned}
V &= N_1 \cup N_2 \cup \{X', X_i, X_i' \mid X \in N_1, 1 \le i \le k\} \\
&\quad \cup \{X_i, X_i', X_i'', X_i''', X_i^{iv}, X_i^{v} \mid X \in N_1, k+1 \le i \le n\} \\
&\quad \cup \{A', A_i, A_i' \mid A \in N_2, 1 \le i \le k\} \\
&\quad \cup \{f, f', E, H, H', H'', H''', H^{iv}, \bar{H}, \#\}, \\
\mu &= [_1[_2\ ]_2[_3\ ]_3[_4\ ]_4]_1, \\
M_1 &= \{X_{init},\ A_{init}H\}, \\
M_2 &= M_3 = M_4 = \emptyset,
\end{aligned}$$

and with the sets R_1, R_2, R_3, R_4 containing the following rules:

R_1: $X \to (X', in)$, for all $X \in N_1$,
$X \to (X_i, in)$, for all $m_i : (X \to Y, B^{(j)} \to \#), k+1 \le i \le n, j = 1, 2$,
$A \to (A', in)$,
$A \to (\#, here)$, for all $A \in N_2$,
$A_i' \to (\#, here)$, for all $A \in N_1$ and $1 \le i \le k$,
$H \to (H', in)$,
$H \to (\bar{H}, in)$,
$H'' \to (H''', in)$,
$H^{iv} \to (\#, here)$,
$E \to (\lambda, out)$,
$f \to (f', here)$,
$f' \to (f', in)$,
$\# \to (\#, here)\tau$;

R_2: $X' \to (X_1, here)$,
$X_i \to (X_{i+1}, here)$, for all $1 \le i \le k-1$,
$X_k \to (\#, out)$,
$A' \to (A_1, here)$,
$A_i \to (A_{i+1}, here)$, for all $1 \le i \le k-1$,
$A_k \to \#$,
$X_i \to (X_i', here)\tau$,
$X_i' \to (\#, out)$,
$X_i' \to (\alpha, out)$,
$A_i \to (A_i', here)\delta$,
$A_i' \to (x, out)$, for all $m_i : (X \to \alpha, A \to x), 1 \le i \le k$,
$A \to (\#, out)$, for all $A \in N_2$,
$X_t \to (\#, out)$, for all $X \in N_1, k+1 \le t \le n$,
$H \to (\#, out)$,

$H' \to (\#, out)$,
$H'' \to (\#, out)$,
$\bar{H} \to (E, here)\tau$,
$E \to (E, out)$,
$f' \to (\lambda, here)\delta$;

R_3, R_4: for each $j = 1, 2$ such that $m_t : (X \to Y, B^{(j)} \to \#)$ for some $k+1 \leq t \leq n$, we introduce in R_{2+j} the following rules:
$X_t \to (X'_t, here)\tau$,
$X'_t \to (X''_t, here)\delta$,
$X''_t \to (X'''_t, here)$,
$X'''_t \to (X^{iv}_t; here)\tau$,
$X^{iv}_t \to (X^v_t, here)$,
$X^v_t \to (Y, out)$,
$X^v_t \to (\#, out)$,
$B^{(j)} \to (\#, out)$,
together with the rules:
$H' \to (H'', here)$,
$H'' \to (H'', out)$,
$H''' \to (H^{iv}, here)\delta$,
$H^{iv} \to (H, out)$,
$\bar{H} \to (\#, out)$,
$H \to (\#, out)$,
$f' \to (\#, out)$,
as well as all rules:
$X' \to (\#, out)$, for all $X \in N_1$.
Moreover, we introduce in R_{2+j} the rules $X_t \to (\#, out)$ for those $t \in \{k+1, \ldots, n\}$ such that the matrix m_t does not contain the rule $B^{(j)} \to \#$ in the second position, $j = 1, 2$.

The matrices of G without rules used in the appearance checking mode are simulated by rules in the skin membrane and by membrane 2, while membranes 3, 4 are used for simulating matrices which contain rules of the forms $B^{(1)} \to \#, B^{(2)} \to \#$, respectively.

Before examining the way these simulations are performed, let us state some general observations about the system Π in order to make clearer its work. We always have exactly two strings in the system; initially X_{init} and $A_{init}H$ in the skin membrane, later the strings derived from these two axioms. Always, the descendants of X_{init} are strings of length 1. The strings generated from $A_{init}H$ are the strings we are interested in.

The symbol # is a trap symbol. It is always introduced directly in the skin region, and it immediately makes the skin membrane thicker. In this way, no string can leave the system, hence nothing is generated, irrespective what the system will continue to do (there is no rule in R_1 which introduces the action δ).

Assume that we have some strings X and wH in membrane 1, $w \in (N_2 \cup \{a\})^*$, and that we apply the rule $X \to (X', in)$. If the symbol X' is sent to one of membranes 3, 4, then the rule $X' \to (\#, out)$ is used and the computation will produce nothing. Thus, the symbol X' should arrive in membrane 2. Assume that this symbol is alone in membrane 2, that is the string wH was not sent to the same membrane with one symbol A replaced by A' (at the same time with the use of the rule $X \to (X', in)$, in the skin membrane we have to use either a rule $A \to (A', in)$, for some $A \in N_2$, or the rule $H \to (H', in)$, for rewriting the second string; using any rule of the form $A \to (\#, here)$ will lead to a non-productive computation; if the string has arrived in membrane 2 because of using the rule $H \to (H', in)$, then only the rule $H' \to (\#, out)$ can be applied, hence this case also leads to a non-productive computation). In membrane 2, the symbol X' will introduce the symbol X_1, then the subscript will be increased, step by step, until either reaching X_k, and then the trap symbol is introduced, or a rule $X_i \to (X_i', here)\tau$ is used. If at the next step we cannot use the rule $X_i \to (\alpha, out)$, because the membrane has thickness 2, then the rule $X_i' \to (\#, out)$ must be used and the computation will produce nothing.

Assume now that we use a rule $X \to (X_t, in)$, for some $k+1 \leq t \leq n$. If the symbol X_t arrives in membrane 2 or in a membrane $2+j$ such that the matrix m_t does not have the rule $B^{(j)} \to \#$ in its second position, then the trap symbol is immediately introduced. Thus, X_t should arrive in one of the membranes 3, 4 such that the matrix m_t is of the form $(X \to Y, B^{(j)} \to \#)$.

Assume again that it arrives here alone. By using the rules of R_j we will change the priming of X, changing the thickness of the membrane from one to two, then back to one, and then again to two. Therefore, the rule $X^{vi} \to (Y, out)$ cannot be used and the trap symbol is introduced. As above, *if the symbol X_t arrives alone in a membrane j, then the computation will produce nothing.*

We conclude that the strings obtained from X and from wH must be sent to the same membrane (more exactly, X' should be in membrane 2 and X_t in one of membranes 3, 4).

Assume now that we are in membrane 2 with X' and $w_1A'w_2H$. Both X' and A' will introduce variants with subscript 1, then the subscripts will be increased, synchronously, for both symbols. If we reach X_k or A_k without using rules $X_i \to (X_i', here)\tau$ and $A_j \to (A_j', here)\delta$, respectively, for some $1 \leq i, j \leq k$, then the trap symbol is introduced by rules $X_k \to (\#, out)$, $A_k \to (\#, out)$.

If we use the rule $X_i \to (X_i', here)\tau$ without using at the same time the rule $A_j \to (A_j', here)\delta$, then the membrane gets thickness 2, hence at the next step we cannot use the rule $X_i' \to (\alpha, out)$, corresponding to the matrix m_i, and therefore the trap symbol is introduced by the rule $X_i' \to (\#, out)$. Similarly, if we first use the rule $A_j \to (A_j', here)\delta$, then the membrane is dissolved, the symbol A_j' reaches the skin membrane, and hence the rule

$A'_j \to (\#, here)$ should be used here. In conclusion, we have to use the rules $X_i \to (X'_i, here)\tau$ and $A_j \to (A'_j, here)\delta$ at the same time, that is for $i = j$. In such a case, the membrane remains of thickness 1 and the symbols X'_i, A'_i will be rewritten at the next step by the rules $X'_i \to (\alpha, out), A'_i \to (x, out)$, which means that we have simulated the matrix $m_i : (X \to \alpha,\ A \to x)$, $\alpha \in N_1 \cup \{f\}$.

Assume now that we send to a membrane 3, 4 the strings X_t, wH', for some $k+1 \leq t \leq n$ (if we have a string containing the symbol H, then the trap symbol is introduced either by the rule $H \to (\#, out)$ or by a rule $B^{(j)} \to (\#, out)$). If the matrix m_t contains the corresponding rule $B^{(j)} \to \#$, then we simulate the use of this matrix in the following way. In the first step, the membrane gets thicker by the rule $X_t \to (X'_t, here)\tau$; at the same time, one of the rules $B^{(j)} \to (\#, out)$ (if the symbol $B^{(j)}$ is present in the string) and $H' \to (H'', here)$ is used. Assume that we are in the good situation when the latter rule has been used. At the next step we cannot use the rule $H'' \to (H'', out)$, because the membrane is thick, hence, if $B^{(j)}$ is present in the string, then we must use the rule $B^{(j)} \to (\#, out)$. If $B^{(j)}$ is not present, then the string remains unchanged, while using the rule $X'_t \to (X''_t, here)\delta$. The membrane returns to thickness 1, hence at the next step we can use the rule $H'' \to (H'', out)$ in parallel with the rule $X''_t \to (X'''_t, here)$. At the next step, the string wH''' is again sent to a lower membrane. If this is not our membrane $2+j$, then the rules in R_j will rewrite X'''_t until reaching X^v_t, the rule $X^v_t \to (Y, out)$ will not be allowed to be used, because the membrane will be of thickness 2, hence the trap symbol is introduced. If the string returns to membrane $2+j$ (in a step when using $X'''_t \to (X^{iv}_t, here)\tau$ and $H'' \to (H''', in)$), then at the next step we use both $X^{iv}_t \to (X^v_t, here)$ and $H''' \to (H^{iv}, here)\delta$, and the membrane returns to thickness 1. At the next step, both strings can be sent out, with X^v_t replaced with Y, as indicated by the first rule of the matrix.

Consequently, the matrix $m_t : (X \to Y, B^{(j)} \to \#)$ with $k+1 \leq t \leq n$, $j \in \{1, 2\}$, is correctly simulated.

In this way, correct derivations in the matrix grammar G are simulated, and, conversely, each computation in Π which does not introduce the trap symbol corresponds to a correct derivation in G.

Let us now see how the simulation of a derivation is completed. Eventually, in membrane 2 a terminal matrix $m_i : (X \to f, A \to x)$ is simulated; that is, rules $X'_i \to (f, out), A'_i \to (x, out)$ are used and the strings f and zH are sent to the skin membrane. No symbol from N_1 is present in the system. If the string zH is rewritten by a rule $A \to (A', in)$, then the trap symbol will be introduced irrespective of where the string is sent: in membrane 2 we will use a rule $A_k \to (\#, out)$ or $A_i \to (A'_i, here)\delta$ (in the latter case, the trap symbol is introduced by a rule from the skin membrane), while in membranes 3, 4 we will use a rule $B^{(j)} \to (\#, out)$ or $H \to (\#, out)$. If the string wH is rewritten by the rule $H \to (H', in)$, then the trap symbol is again produced.

In membrane 2 we directly use the rule $H' \to (\#, out)$. From membranes 3, 4 we will send H'' out, then we send H''' into one of membranes 2, 3, 4; in membrane 2 we directly introduce the trap symbol, in membranes 3, 4 we will use the rule $H''' \to (H^{iv}, out)$, and the symbol H^{iv} will introduce the trap symbol in the skin membrane.

Therefore, we have to use the rule $H \to (\bar{H}, in)$ in parallel with the rule $f \to (f', here)$. In both membranes 3, 4 we will again introduce the trap symbol; in membrane 2 we can replace the symbol $\bar{H}$ by E, by using the rule $\bar{H} \to (E, here)\tau$. At the same time, the rule $f' \to (f', in)$ is used in the skin membrane. If f' arrives in any of membranes 3, 4, then the trap symbol is introduced. Thus, f' should arrive in membrane 2 at the same time when this membrane gets thicker. At the next step we cannot use the rule $E \to (E, out)$. If any non-terminal symbol $A \in N_2$ is present, then the rule $A \to (\#, out)$ from R_2 must be used. At the same step, the rule $f' \to (\lambda, here)\delta$ is used and membrane 2 returns to thickness 1. Thus, at the next step the string can be sent to the skin membrane (if the symbol # was not produced, then the string is terminal). In the skin membrane we can use the rule $E \to (\lambda, out)$ and the string is sent out of the system.

Summing up, the following assertions hold:

1. The strings $X', w_1A'w_2H$, and X_t, wH' should go together, otherwise the trap symbol is introduced.
2. The strings $X', w_1A'w_2H$ can be correctly processed only in membrane 2, while the strings X_t, wH' can be correctly processed only in one of membranes 3, 4, corresponding to the symbol $B^{(j)}$ which is used in the appearance checking manner in the matrix m_t ("correctly processed" means here "in such a way as not to introduce the trap symbol").
3. Membrane 2 correctly simulates the matrices of G which do not contain rules used in the appearance checking manner.
4. Membranes $2+j$ correctly simulate the use of matrices $m_t, k+1 \leq t \leq n$, which contain rules $B^{(j)} \to \#$, $j = 1, 2$.
5. The computation of Π stops only if the simulated derivation of G is terminal (and the result is sent out of the system).

We conclude that $L(\Pi) = L(G)$. □

From the proof of Theorem 5.1.9 we see that if the matrix grammar we start with does not use the appearance checking feature, then a system with two membranes suffices in order to simulate it; that is, the next inclusion holds.

Corollary 5.1.3. $MAT^{\lambda} \subseteq LSP_2(rw, in, \delta, \tau)$.

We do not know if this inclusion is proper. Because MAT^{λ} is strictly included in RE, from Theorem 5.1.9 we obtain the fact that $LSP_4(rw, in, \delta, \tau)$ contains languages which are not in the family MAT^{λ}. This result can be improved.

Theorem 5.1.10. $LSP_3(rw, in, \delta, \tau) - MAT^\lambda \neq \emptyset$.

Proof. Let us consider the following rewriting P system (of degree 3):

$$
\begin{aligned}
\Pi &= (V, V, [_1[_2\ ]_2[_3\ ]_3]_1, \{X, aH\}, \emptyset, \emptyset, R_1, R_2, R_3),\\
V &= \{a, b, X, X', X'', X''', X^{iv}, X^v, X^{vi}, Y, Y', Y'', Y''', Y^{iv}, Y^v, Y^{vi},\\
&\quad E, H, H', H_1, H_2, H'', \#\},\\
R_1 &= \{X \to (X, here),\ (X \to X, in),\ a \to (bb, here),\\
&\quad H \to (H, in),\ Y \to (Y, in),\ H_1 \to (H_1, in),\\
&\quad H_2 \to (H_2, in),\ H_2 \to (\#, here),\ H'' \to (\#, here),\\
&\quad X^{vi} \to (\#, here),\ Y'' \to (\#, here),\ Y''' \to (\#, here),\\
&\quad Y^{vi} \to (\#, here),\ X''' \to (\#, here),\ H \to (\#, here),\\
&\quad E \to (\lambda, out),\ \# \to (\#, here)\tau\},\\
R_2 &= \{X \to (X', here)\tau,\ X' \to (X'', here),\ X'' \to (X''', here)\delta,\\
&\quad X''' \to (X^{iv}, here),\ X^{iv} \to (X^v, here)\tau,\ X^v \to (X^{vi}, here),\\
&\quad X^{vi} \to (Y, out),\ X^{vi} \to (\#, here),\ a \to (\#, out),\\
&\quad H \to (H', here),\ H' \to (H_1, out),\ H_1 \to (H'', here)\delta,\\
&\quad H'' \to (H, out),\ H'' \to (\#, out),\ H_2 \to (\#, out),\\
&\quad Y \to (\#, out)\},\\
R_3 &= \{Y \to (Y, here)\tau,\ Y \to (Y', here)\tau,\ Y' \to (Y'', here),\\
&\quad Y'' \to (Y''', here)\delta,\ Y''' \to (Y^{iv}, here),\ Y^{iv} \to (Y^v, here)\tau,\\
&\quad Y^v \to (Y^{vi}, here),\ Y^{vi} \to (X, out),\ Y^{vi} \to (\#, out),\\
&\quad b \to (a, here)\delta,\ H \to (H', here),\ b \to (\#, out),\\
&\quad H' \to (H_2, out),\ H_2 \to (H'', here)\delta,\ H'' \to (H, out),\\
&\quad H'' \to (\lambda, here),\ H_1 \to (\#, out),\ X \to (\#, out),\\
&\quad Y^{vi} \to (\lambda, out),\ H'' \to (E, out)\}.
\end{aligned}
$$

We start with the strings X, aH in the skin membrane; assume that we have here some strings X, a^nH, for $n \geq 1$. By iteratively using the rules $X \to (X, here)$, $a \to (bb, here)$ we can replace the n occurrences of a by $2n$ occurrences of b. At any moment, we can use $X \to (X, in)$ or $H \to (H, in)$. If X arrives in membrane 3, then the trap symbol $\#$ is introduced. As in the previous proof, it will make the skin membrane impermeable, hence no string will be produced. If X arrives alone in membrane 2, then again the trap symbol is introduced (this is similar to the work of membranes 1, 2 in the proof of Theorem 5.1.9; the rule $X^{vi} \to (Y, out)$ cannot be used, because the membrane is of thickness 2, and hence the rule $X^{vi} \to (\#, out)$ must be used).

Assume now that wH has arrived in membrane 2, but not at the same time with X, which comes later or it does not come at all. If we use the rule

$a \to (\#, out)$, then the computation will produce nothing. We can use the rule $H \to (H', here)$, then $H' \to (H_1, out)$, irrespective of whether or not the rule $X \to (X', here)\tau$ is used in membrane 2. If the symbol H_1 cannot be sent back to membrane 2 (because that membrane has become thicker), then it will go to membrane 3 and the trap symbol is introduced. If H_1 is inside membrane 2, then at the next step either it will dissolve the membrane, a case which again produces the trap symbol (in membrane 1), or it is used in parallel with the rule $X \to (X', here)\tau$; in two steps, membrane 2 will be dissolved by the rule $X'' \to (X''', here)\delta$ and the trap symbol is introduced by the rule $X''' \to (\#, here)$ from R_1.

Thus, we have to send the string X to membrane 2 at the same time as the string wH. If this is the case, then in the first step membrane 2 becomes thicker, hence at the next step we cannot use the rule $H' \to (H_1, out)$. If any occurrence of a is present in w, then the trap symbol is produced, otherwise the string wH' will wait unchanged. At the next step, we use the rule $X'' \to (X''', here)\delta$ and the membrane returns to thickness 1. Therefore, at the next step we can send out the symbol H_1 (in parallel with using the rule $X''' \to (X^{iv}, here)$). At the next step H_1 comes back, while the membrane again gets thickness 2. When using the rules $X^v \to (X^{vi}, here)$ and $H_1 \to (H'', here)\delta$, the membrane returns to thickness 1, hence at the next step we can send out the strings Y and wH; we know that w is composed of only occurrences of b.

The work of membrane 3 is similar, with two additional features: one can replace each b by a here, and one can also finish the computation.

The symbol X cannot arrive in membrane 3, because it will immediately introduce the trap symbol. If wH arrives in membrane 3 and Y is not there, then either the membrane is dissolved (by the rule $b \to (a, here)\delta$) or H' is introduced, and H_2 is sent out; if H_2 cannot come back, then the trap symbol is introduced in the skin membrane. If H_2 comes back at the same time with using the rule $Y \to (Y, here)\tau$, then at the next step the membrane returns to thickness 1, but it is immediately made thicker by any of the rules $Y \to (Y, here)\tau$ and $Y \to (Y', here)\tau$, hence the rule $H'' \to (H, out)$ cannot be used. We have to apply the rule $H'' \to (\lambda, here)$ and the string will remain forever in membrane 3. If the evolution of Y and of its variants does not end with the rule $Y^{vi} \to (\lambda, out)$, then either the trap symbol is introduced (by the rule $Y^{vi} \to (\#, out)$) or X is sent out and it will eventually introduce the symbol $\#$, as we have seen above.

Assume now that the strings Y and wH arrive in membrane 3. After using the rules $Y \to (Y, here)\tau$ and $b \to (a, here)\delta$ for a while, we will pass to using the rules $Y \to (Y', here)\tau$ and $H \to (H', here)$. If we go to $Y \to (Y', here)\tau$ while still using $b \to (a, here)\delta$, then the computation will produce nothing: in the next step we will use $Y' \to (Y'', here)$, then $Y'' \to (Y''', here)\delta$, which will dissolve the membrane (it has the thickness 1); in the skin membrane we will use $Y''' \to (\#, here)$. If, however, we pass to using $H \to (H', here)$ while still using $Y \to (Y, here)\tau$, then we continue in a correct manner: the rule

$H' \to (H_2, out)$ cannot be used because the membrane has thickness 2, hence the rule $b \to (a, here)\delta$ must be used. This must be done for all ocurrences of b, otherwise the rule $b \to (\#, out)$ must be applied. Therefore, we have to leave the rule $Y \to (Y, here)\tau$ and use $Y \to (Y', here)\tau$ only after changing all occurrences of b into a. From now on the computation continues in a way similar to what happens in membrane 2: by the interplay of variants of Y and of H we check that the operation is correct and we send out of the membrane the strings X and $w'H$, with w' being the result of replacing all occurrences of b in w by a.

The process can be iterated, hence we can double the length of our string any number of times.

As we have seen above, the only way to finish the computation is to use simultaneously the rules $Y^{vi} \to (\lambda, out), H'' \to (E, out)$ from membrane 3, followed by $E \to (\lambda, out)$ in membrane 1. From the previous discussion, it is then clear that we have the equality $L(\Pi) \cap \{a\}^+ = \{a^{2^n} \mid n \geq 1\}$, therefore the language $L(\Pi)$ is not a matrix one (the family MAT^λ is closed under intersection with regular languages, while $\{a^{2^n} \mid n \geq 1\}$ is not in MAT^λ – see Subsection 2.3.7). □

From Theorem 5.1.1 we know that $ELSP_1(rw) = CF$ and $LSP_1(rw) \subset CF$. Somewhat surprisingly (this indicates the power of actions δ, τ, their intricate interplay, according to the way of using the rules in a rewriting P system), if we make use of thickness changing, then systems with only one membrane can generate non-context-free languages.

Theorem 5.1.11. $LSP_1(rw, \delta, \tau) - CF \neq \emptyset$.

Proof. Let us consider the system $\Pi = (V, V, [_1\ ]_1, \{AB, X_0, Y_0\}, R_1)$, with the alphabet

$$V = \{A, A', A'', A''', B, B', B'', B''', X_0, \ldots, X_7, Y_0, \ldots, Y_7, a, b, c, \#\},$$

and the following rules:

$$\begin{array}{lll}
A \to (aA'b, here), & X_0 \to (X_1, here), & Y_0 \to (Y_1, here), \\
B \to (cB', here), & X_1 \to (X_2, here), & Y_1 \to (Y_2, here), \\
A' \to (A'', here)\delta, & X_2 \to (X_3, here)\tau, & Y_2 \to (Y_3, here)\delta, \\
A' \to (\#, here), & & \\
B' \to (B'', here)\delta, & X_3 \to (X_4, here)\tau, & Y_3 \to (Y_4, here)\delta, \\
B' \to (\#, here), & & \\
A'' \to (A''', here), & X_4 \to (X_5, here), & Y_4 \to (Y_5, here), \\
B'' \to (B''', here), & X_5 \to (X_6, here), & Y_5 \to (Y_6, here), \\
A''' \to (A, here)\delta, & X_6 \to (X_7, here)\tau, & Y_6 \to (Y_7, here)\delta, \\
A''' \to (\#, here), & & \\
B''' \to (B, here)\delta, & X_7 \to (X_0, here)\tau, & Y_7 \to (Y_0, here)\delta, \\
B''' \to (\#, here), & & \\
A''' \to (ab, here)\delta, & & \\
B''' \to (c, out). & &
\end{array}$$

This system generates a non-context-free language. We examine only a few steps of a computation in Π. Note that the symbols X_i, Y_i are always deterministically processed, and that the actions δ, τ introduced by rules dealing with these symbols appear simultaneously. This means that these actions cancel each other, the membrane remains of thickness 1, but at the same time we can use rules dealing with symbols A, B (and their primed versions) and introducing δ (several actions δ at the same time have the same effect as one single action). In the initial configuration we can use any of the rules $A \to (aA'b, here)$ and $B \to (cB', here)$, but in the second step we have to use the remaining rules between these two: the rules $A' \to (A'', here)\delta$ and $B' \to (B'', here)\delta$ cannot be used, because we are not allowed to dissolve the skin membrane, while the rules $A' \to (\#, here)$ and $B' \to (\#, here)$ introduce the trap symbol # which will never be removed. Assume that after having the pair of strings $aA'bcB', X_2$ we use the rule $A' \to (A'', here)\delta$ (in parallel with $X_2 \to (X_3, here)\tau$ and $Y_2 \to (Y_3, here)\delta$), then we use $A'' \to (A''', here)$ (in parallel with $X_3 \to (X_4, here)\tau$ and $Y_3 \to (Y_4, here)\delta$). The membrane still has thickness 1, hence the string $aA'''bcB'$ cannot be processed without introducing the trap symbol #. Thus, we have to use both rules $A' \to (A'', here)\delta$ and $B' \to (B'', here)\delta$, then both rules $A'' \to (A''', here)$ and $B'' \to (B''', here)$, and we get the strings $aA'''bcB''', X_6, Y_6$. Asume that we now use the rule $A''' \to (A, here)\delta$ and then $A \to (aA'b, here)$. Again we get a string which can be rewritten only by rules which introduce the symbol #. Therefore, we have to return to $aAbcB$ when also returning to X_0, Y_0. The process can be iterated, and in this way we synchronously increase the number of occurrences of a, b, c.

From a string $a^n A''' b^n c^n B'''$ (in the presence of X_6, Y_6) we can use any of the rules $A''' \to (ab, here)\delta$ and $B''' \to (c, out)$. The rule $B''' \to (c, out)$ sends out a string containing the symbol A'''. If we use $A''' \to (ab, here)\delta$, then at the next step we have also to use $B''' \to (c, out)$: otherwise, after obtaining $a^{n+1} b^{n+1} c^{n+1} B'$, we cannot do anything else than introduce #.

Consequently, $L(\Pi) \cap a^* b^* c^* = \{a^n b^n c^n \mid n \geq 2\}$, which is not a context-free language. □

5.2 Some Variants and Their Power

We start by discussing some modifications of the rewriting P systems which were considered in the case of symbol–objects (and, in general, having biological motivations). Some of the definitions can be easily "translated" from systems with symbol–objects to systems with string–objects, while others need to be adapted carefully. The same assertion is valid for (universality) proofs – in the case of symbol–objects we process the whole multiset of symbols from a region in a maximally parallel manner, in the case of string–objects we work with sets of strings and each string is processed sequentially. Thus, we cannot

simply use a construction for string–objects in the case of symbol–objects and claim that the result is the same modulo the ordering of symbols.

5.2.1 Rule Creation

However, this passage from string–objects to symbol–objects is possible when we work with variants of membrane systems which in the case of symbol–objects use only one rule in each transition (while in the case of string–objects we have only one string in the system). These properties are fulfilled by systems with rule creation – see the proof of Theorem 3.6.3.

The definition of a rewriting membrane system with rule creation is obvious. Consider rules of the form $r : A \to (x, tar)/z$, where $z \in lab^*$ is a multiset of labels of possible rules; when using the rule r, one copy of it is consumed and the rules specified by z become available for the next step.

Formally, a rewriting P system with rule creation is a construct $\Pi = (V, T, \mu, M_1, \ldots, M_m, lab, R_{pos}, R_1, \ldots, R_m)$ (the alphabet, the terminal alphabet, the membrane structure, the sets of string–objects placed in the m regions of μ, the labels of possible rules, the list of possible rules, the multisets of rules present in the m regions of μ, respectively) similar to the kind of systems considered in Subsection 3.6.4. We denote by $[E]LSP_m(rw, in, rule)$ the family of languages generated by [extended] rewriting P systems with rule creation. By simple inspection of the proof of Theorem 3.6.3 one can see that also the following result is true (once introduced, the trap symbol # is forever unchanged, hence the string will never become terminal; thus, instead of the halting condition from the case of systems with symbol–objects here we use the selection by means of the terminal alphabet).

Theorem 5.2.1. $ELSP_1(rw, rule) = RE$.

5.2.2 Conditional Rewriting

The extension of rewriting P systems by considering promoters/inhibitors in the form of strings associated with rules and which must be present in the strings rewritten by rules, leads to trivial universality results (in the extended case), because of the direct resemblance to the semi-conditional context-free grammars, see again Subsection 2.3.7.

The problem becomes non-trivial if we check symbols only, in the permitting or the forbidding mode: this resembles the random context grammars, which are not universal if only permitting or only forbidding context conditions are used. In our framework, it turns out that the forbidding conditions lead to universality, while permitting conditions do not, which is a rather interesting result in view of the comparison of the power of promoters and of inhibitors.

The rules we use are of the forms $a \to (x, tar)|_U$ in the permitting case and $a \to (x, tar)|_{\neg U}$ in the forbidding case, where a is a symbol, x is a string,

and U is a set of symbols, from the alphabet of the system. A rule of the former type can be applied to a string $w = w_1aw_2$ only if $U \subseteq alph(w)$, and a rule of the latter type can be applied to $w = w_1aw_2$ only if $alph(w) \cap U = \emptyset$ (note that in both cases we also take into consideration the symbol a, hence a forbidding context rule of the form $a \to x|_{\neg U}$, with $a \in U$, can never be applied). Rules of the form $a \to (x, tar)$ are considered by default both with permitting and with forbidding conditions, and they are freely applied, without checking any condition.

We denote by $[E]LSP_m(rw, in, pro)$ and $[E]LSP_m(rw, in, inh)$ the families of languages generated by [extended] P systems of degree at most $m \geq 1$, with string–objects, using rewriting rules with permitting, respectively forbidding, conditions (promoters and inhibitors, hence the abbreviations *pro* and *inh*). When the degree of the systems is not bounded, we replace the subscript m with $*$.

Here are the two announced results. First, the universality of systems with inhibitors.

Theorem 5.2.2. $LSP_2(rw, in, inh) = RE$.

Proof. Consider a matrix grammar with appearance checking in the Z-binary normal form, with the standard notation, and construct the following system:

$$\begin{aligned}
\Pi &= (V, V, [_1[_2\ ]_2]_1, \{X_{init}A_{init}\}, \emptyset, R_1, R_2),\\
V &= N_1 \cup N_2 \cup T \cup \{Z\} \cup \{X_i \mid X \in N_1, 1 \leq i \leq k\},\\
R_1 &= \{X \to (X_i, here),\\
&\qquad A \to (x, in)|_{\neg(\{X\} \cup \{X_j \mid j \neq i\})} \mid \text{for } m_i : (X \to \alpha, A \to x), 1 \leq i \leq k\}\\
&\quad \cup \{X \to (Y, here)|_{\neg\{A\}} \mid \text{for } m_i : (X \to Y, A \to \#), k+1 \leq i \leq n\}\\
&\quad \cup \{Z \to (\lambda, out)\},\\
R_2 &= \{X_i \to (Y, out) \mid \text{for } m_i : (X \to Y, A \to x), 1 \leq i \leq k\}.
\end{aligned}$$

The fact that $L(G) = L(\Pi)$ is easy to check, bearing in mind that only one symbol from N_1 is always present. □

Surprisingly enough, the use of permitting conditions does not lead to universality, and it simply does not increase the power of rewriting P systems: the family of languages generated by such systems coincides with the family of languages generated by matrix grammars without appearance checking.

Theorem 5.2.3. $ELSP_2(rw, in, pro) = ELSP_*(rw, in, pro) = MAT^\lambda$.

Proof. The inclusion $ELSP_2(rw, in, pro) = ELSP_*(rw, in, pro)$ is obvious.

In order to prove the inclusion $MAT^\lambda \subseteq ELSP_2(rw, in, pro)$, let $G = (N, T, S, M)$ be a matrix grammar in the binary normal form, replace each terminal matrix $(X \to \lambda, A \to x)$ with $(X \to f, A \to x)$, where f is a new symbol, and then construct the system

$$
\begin{aligned}
\Pi &= (V, T, [_1[_2\ ]_2]_1, \{X_{init}A_{init}\}, \emptyset, R_1, R_2),\\
V &= N_1 \cup N_2 \cup T \cup \{X_i \mid X \in N_1, 1 \le i \le k\},\\
R_1 &= \{X \to (X_i, here),\\
&\qquad A \to (x, in)|_{X_i} \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}\\
&\quad \cup \{f \to (\lambda, out)\},\\
R_2 &= \{X_i \to (\alpha, out) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\}.
\end{aligned}
$$

The equality $L(\Pi) = L(G)$ is obvious (this system is very similar to that from the proof of Theorem 5.2.2, but using promoters and a terminal alphabet, because we do not have the Z-binary normal form available).

It remains to prove the inclusion $ELSP_*(rw, in, pro) \subseteq MAT^\lambda$. To this end, let us consider the system $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m)$; assume that the skin membrane has label 1.

We build a matrix grammar (without appearance checking) $G = (N, T, S, M)$, generating the same language as Π in the following way (h is the morphism defined by $h(a) = a', a \in V$):

$$
\begin{aligned}
N &= \{a' \mid a \in V\} \cup \{X_i \mid 0 \le i \le m\} \cup \{S\},\\
M &= \{(S \to X_i h(w)) \mid w \in M_i, 1 \le i \le m\}\\
&\quad \cup \{(X_i \to X_i, a'_1 \to a'_1, \ldots, a'_s \to a'_s, A' \to h(x)) \mid\\
&\qquad A \to (x, here)|_{\{a_1,\ldots,a_s\}} \in R_i, 1 \le i \le m\}\\
&\quad \cup \{(X_i \to X_j, a'_1 \to a'_1, \ldots, a'_s \to a'_s, A' \to h(x)) \mid\\
&\qquad A \to (x, in)|_{\{a_1,\ldots,a_s\}} \in R_i, 1 \le i \le m, \text{ and } j \text{ is the label}\\
&\qquad \text{of a membrane placed directly inside membrane } i\}\\
&\quad \cup \{(X_i \to X_j, a'_1 \to a'_1, \ldots, a'_s \to a'_s, A' \to h(x)) \mid\\
&\qquad A \to (x, out)|_{\{a_1,\ldots,a_s\}} \in R_i, 2 \le i \le m, \text{ and } j \text{ is the label}\\
&\qquad \text{of the membrane surrounding membrane } i\}\\
&\quad \cup \{(X_1 \to X_0, a'_1 \to a'_1, \ldots, a'_s \to a'_s, A' \to h(x)) \mid\\
&\qquad A \to (x, out)|_{\{a_1,\ldots,a_s\}} \in R_1\}\\
&\quad \cup \{(X_0 \to X_0, a' \to a) \mid a \in T\}\\
&\quad \cup \{(X_0 \to \lambda)\}.
\end{aligned}
$$

This is an extension of the construction from the proof of Theorem 5.1.4 in order to take the promoters into account. Consequently, $L(\Pi) = L(G)$, which completes the proof. □

Combining the results from Theorems 5.1.5 and 5.2.3, we obtain the following synthesis result:

Corollary 5.2.1. $MAT^\lambda = ELSP_4(rw, in) = ELSP_*(rw, in) = ELSP_2(rw, in, pro) = ELSP_*(rw, in, pro)$.

Another very interesting case, which has no counterpart in classic formal language theory, appears when the condition on the application of a rule is not checked on the rewritten string, but on other strings present in the same region of a P system. That is, we consider rules of the form $a \to (u, tar)|_w$, where w is a string. Such a rule is applied in a region i to a string $x = x_1 a x_2$ only if w is another string from region i; the result of the rewriting, $x_1 u x_2$, is sent to the membrane indicated by tar. After rewriting, a string is consumed, and in the next step we have the result of rewriting. If two different strings are rewritten into the same string, then in the region we have two copies of the same string (each initial string has exactly one descendant, irrespective of the form of the descendants). The "promoter string" w can be rewritten in the same step. It may be rewritten even by a rule of the form $a \to (u, tar)|_w$, hence having w as promoter, provided that two copies of w are present at that time in the region. As usual, rules $a \to (u, tar)$, hence without a promoting string, can be used freely at any time.

Let us denote by $[E]LSP_m(rw, in, cond)$ the family of languages generated by [extended] P systems of degree at most $m \geq 1$ with conditional rewriting rules as defined above.

This type of conditional use of rules is rather powerful.

Theorem 5.2.4. $LSP_1(rw, cond) = RE$.

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar in the Z-binary normal form, and construct the system

$$\Pi = (V, T, [_1\ ]_1, \{X_{init},\ A_{init}\}, R_1),$$

with the alphabet

$$V = N_1 \cup N_2 \cup \{X_i \mid X \in N_1, 1 \leq i \leq n\} \cup \{Z, \#\},$$

and the following rules:

1. For each matrix $m_i : (X \to Y, A \to x)$, $1 \leq i \leq k$, we introduce the rules:
   $X \to (X_i, here)$,
   $A \to (x, here)|_{X_i}$,
   $X_i \to (Y, here)$
   as well as all rules
   $\alpha \to (\#, here)|_{X_i}$, for $\alpha \in N_2 \cup T$.
2. For each matrix $m_i : (X \to Y, A \to \#)$, $k + 1 \leq i \leq n$, we introduce the rules:
   $X \to (X_i, here)$,
   $A \to (\#, here)|_{X_i}$,
   $X_i \to (Y; here)$.
3. We also consider the rules:
   $a \to (a, out)|_Z$, for all $a \in T$.

We start by introducing the two strings X_{init} and A_{init}, whose descendants will collaborate by means of the promoting conditions. At any time when a symbol $X \in N_1$ is present, any rule $X \to (X_i, here)$, for $1 \leq i \leq n$, can be used; because we consider only one possible continuation, at the next step exactly one symbol X_i will be present. Only after using such a rule can we use a rule of the form $A \to (x, here)|_{X_i}$ or $A \to (\#, here)|_{X_i}$. Assume that we have introduced the symbol–string X_i corresponding to a matrix $m_i : (X \to Y, A \to x)$, $1 \leq i \leq k$. In the presence of X_i we can use the rule $A \to (x, here)|_{X_i}$; if this rule cannot be used, then at least one of the rules $\alpha \to (\#, here)|_{X_i}$, for $\alpha \in N_2 \cup T$, is applicable, hence it must be used. (If the string derived from A_{init} is empty, then no string is sent out – remember that when comparing two languages we ignore the string λ.) Therefore, either we correctly simulate the matrix m_i, or the trap symbol $\#$ is introduced. If we have introduced X_i for some matrix $m_i : (X \to Y, B \to \#)$ with $k+1 \leq i \leq n$, then the only rule which can be used in the presence of X_i is $B \to (\#, here)|_{X_i}$. Thus, if B is present, then the trap symbol is introduced, otherwise the string derived from A_{init} remains unchanged (note that this time we do not have rules $\alpha \to (\#, here)|_{X_i}$ as above). At the next step, X_i is replaced by Y. This again corresponds to the correct simulation of the matrix m_i. Clearly, only strings which are terminal with respect to G can exit the system. Consequently, $L(G) = L(\Pi)$, and this concludes the proof. □

Obviously, if we have only one initial string and all rules of a system are conditional, then no rewriting is possible, hence by such systems we obtain only singleton languages. Systems of degree 1, with two initial strings, provide a characterization of context-free languages.

Theorem 5.2.5. *Extended P systems of degree 1 with conditional rewriting, with all rules being conditional, and starting from two strings, generate exactly the family of context-free languages.*

Proof. For a context-free grammar $G = (N, T, S, R)$ we consider the system $\Pi = (N \cup \{D\}, T, [_1\ ]_1, \{S, D\}, \{A \to (u, here)|_D,\ A \to (u, out)|_D \mid A \to u \in R\})$, where D is a new symbol; the equality $L(G) = L(\Pi)$ is obvious.

Conversely, consider a P system $\Pi = (V, T, [_1\ ]_1, \{w_1, w_2\}, R_1)$ as in the theorem. If $L(\Pi)$ is finite, then it is trivially context-free, hence let us assume that $L(\Pi)$ is infinite.

We assume that $R_1 = \{r_i : a_i \to x_i|_{z_i} \mid 1 \leq i \leq k\}$ and we denote $M = \{z_1, \ldots, z_k\}$. Consider two sequences of strings, one generated from w_1 and one generated from w_2, ending at the moment when a string is sent out of the system:

$$\sigma_i = (w_i = w_{i,0}), w_{i,1}, w_{i,2}, \ldots, w_{i,s_i},\ i = 1, 2.$$

Due to the non-determinism of choosing the rules and the place where we apply them, there are several sequences of this form for each $w_i, i = 1, 2$. In

order that a derivation step $w_{i,j} \Longrightarrow w_{i,j+1}, i = 1, 2$, is possible it is necessary that $w_{3-i,j} \in M$ (the "brother" string should be used as a promoter).

For each pair σ_1, σ_2, consider the last string w_{1,j_1}, w_{2,j_2} from M (that is, j_1, j_2 are maximal with this property). If $j_i = s_i$ (hence w_{i,j_i} is the last string in the sequence, the string sent out), then $w_{i,s_i} \in M$. If $j_i < s_i$, then $w_{3-i,j} \in M$ for all $j \geq j_{3-i}$; moreover, because these strings cannot be rewritten, we have $w_{3-i,j_i+1} = w_{3-i,j_i+2} = \ldots = w_{3-i,s_{3-i}}$. Let us denote

$$R(z) = \{r : a \to x|_z \mid z \in R_1\},$$

the set of rules promoted by a string $z \in M$. The string w_{i,s_i} is generated by the pure context-free grammar $G = (V, w_{i,j_i}, R(w_{3-i,j_i}))$. Conversely, all strings generated by such a grammar can be produced by Π, but not necessarily all of them are sent out. We proceed again as in the proof of Theorem 5.1.1, when proving the inclusion $ELSP_1(rw) \subseteq CF$, and select only those strings which are sent out correctly (only at the end of a derivation). This selection preserves from $L(G)$ a context-free sublanguage – let us denote it by $L(w_{i,j_i}, R(w_{3-i,j_i}))$ (note that it depends on the two sequences σ_1, σ_2). Conversely, each string from $L(w_{i,j_i}, R(w_{3-i,j_i}))$ is generated and sent out by Π – if it is also over T, then it is in $L(\Pi)$. There are finitely many languages $L(w_{i,j_i}, R(w_{3-i,j_i}))$ as above, because $w_{i,j_i} \in M$ and $w_{3-i,j_i} \in M$, while M is a finite set.

Therefore, we can write the language $L(\Pi)$ as the union of a subset of M (possibly equal to M) with a finite union of languages of the form $L(z, R(z'))$, for some $z, z' \in M$, intersected with T^*. This means that $L(\Pi)$ is a context-free language. □

The proof of Theorem 5.2.4 involves a system of degree 1, starting from two initial strings, but also having rules which are freely applied. This small difference from the system as discussed in the previous proof entails the surprising jump from CF to RE. On the other hand, we can modify the proof of Theorem 5.2.4 such that all rules are conditional, *by adding one more axiom string*, a dummy one, say D, which promotes all rules which are now without promoters. Again the difference between the systems from the proofs of Theorems 5.2.4 (modified) and 5.2.5 is very small – three axioms versus two axioms – but this is again the difference between context-freeness and Turing computability!

5.2.3 Conditional Communication

Another case of interest is to impose restrictions not on the rule application, but on the communication of string–objects through membranes. Such an extension does not makes sense in the case of symbol–objects, but it has a clear biological counterpart: the passage of molecules through membranes depends in many cases on the structure of the molecules. For instance, if a

molecule w_1uw_2 contains the subword u which can bind to a bigger molecule or to a molecule already bound to a membrane (because of the Watson–Crick complementarity, in the case of single-stranded DNA molecules), then the molecule cannot change its region (it cannot be moved *in* or *out*).

We investigate this type of conditional communication here, using both promoting and inhibiting conditions. These conditions can be of different forms (we can check substrings, prefixes, suffixes, symbols, the shape of the whole string, etc.), but we examine only a few of them here. Specifically, we consider systems

$$\Pi = (V, T, \mu, M_1, \ldots, M_m, (R_1, P_1, F_1), \ldots, (R_m, P_m, F_m)),$$

where $V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m$ are, as usual, the alphabet of the system, the terminal alphabet, the membrane structure (with m membranes injectively labelled by $1, 2, \ldots, m$), finite languages over V, representing the strings initially present in the regions $1, 2, \ldots, m$ of the system, and finite sets of context-free rules over V (without target indications) present in regions $1, 2, \ldots, m$ of the system, respectively, while P_i are *permitting conditions* and F_i are *forbidding conditions* associated with region $i, 1 \leq i \leq m$, which regulate the communication of strings produced in this region.

The conditions considered here can be of the following forms:

1. *Empty*: no restriction is imposed on strings, they either exit the current membrane or enter any of the directly inner membranes freely (but they cannot remain in the current membrane); we denote an empty permitting condition by $(true, tar), tar \in \{in, out\}$, and an empty forbidding condition by $(false, nottar), tar \in \{in, out\}$.
2. *Symbols checking*: each P_i is a set of pairs $(a, tar), tar \in \{in, out\}$, for $a \in V$, and each F_i is a set of pairs $(b, nottar), tar \in \{in, out\}$, for $b \in V$; a string w can go to a lower membrane only if there is a pair $(a, in) \in P_i$ with $a \in alph(w)$, and for each $(b, notin) \in F_i$ we have $b \notin alph(w)$; similarly, for sending the string w out of membrane i it is necessary to have $a \in alph(w)$ for at least one pair $(a, out) \in P_i$ and $b \notin alph(w)$ for all $(b, notout) \in F_i$.
3. *Substrings checking*: each P_i is a set of pairs $(u, tar), tar \in \{in, out\}$, for $u \in V^+$, and each F_i is a set of pairs $(v, nottar), tar \in \{in, out\}$, for $v \in V^+$; a string w can go to a lower membrane only if there is a pair $(u, in) \in P_i$ with $u \in Sub(w)$, and for each $(v, notin) \in F_i$ we have $v \notin Sub(w)$; similarly, for sending the string w out of membrane i it is necessary to have $u \in Sub(w)$ for at least one pair $(u, out) \in P_i$ and $v \notin Sub(w)$ for all $(v, notout) \in F_i$.

We say that we have conditions of the types *empty*, *symb*, sub_k, respectively, where k is the length of the longest string in all P_i, F_i.

The transitions in a system as above are defined in the following way. In each region, each string which can be rewritten by a rule from that region

is rewritten. The rule to be applied and the symbol rewritten by it are non-determistically chosen. Each string obtained in this way is checked against the conditions P_i, F_i from that region. If it fulfils the requested conditions, then it will be immediately sent out of the membrane or to an inner membrane, if any exists; if it fulfils both *in* and *out* conditions, then it is sent either out of the membrane or to a lower membrane, non-deterministically choosing the direction – and non-deterministically choosing the inner membrane in the case when several directly inner membranes exist. If a string does not fulfil any condition, or it fulfils only *in* conditions and there is no inner membrane, then the string remains in the same region. If a string cannot be rewritten (this can be the case with strings coming from other membranes), then it is directly checked against the communication conditions, and, as above, it leaves the membrane (or remains inside forever) depending on the result of this checking.

That is, the rewriting has priority over communication: we first try to rewrite a string and only after that do we try to communicate the result of the rewriting or the string itself if no rewriting is possible on it.

As usual, the result of a computation (not necessarily halting) consists of all strings sent out of the system. In the case of non-extended systems, all strings sent out are accepted; in the extended case only the strings over T are accepted. The language generated in this way by a system Π is denoted by $L(\Pi)$.

The family of all languages $L(\Pi)$, computed as above by [extended] systems Π of degree at most $m \geq 1$ and with permitting conditions of type α and forbidding conditions of type β, is denoted by $[E]LSP_m(rw, \alpha, \beta)$, $\alpha, \beta \in \{empty, symb\} \cup \{sub_k \mid k \geq 2\}$. If the degree of the systems is not bounded, then the subscript m is replaced by $*$.

The proofs of the theorems given below use several techniques which are repeated from one proof to another, so we will not give all the details, just basic constructions. As in previous proofs, too, we will still use matrix grammars in the binary normal form, but we will also use normal forms for Chomsky grammars, in particular the Kuroda normal form.

In general, the distinction between terminal and non-terminal alphabets can be simulated by the use of an additional membrane letting only terminal strings out, and this can be checked by forbidding conditions.

We start the investigation of the power of conditional communication by formally stating this last observation.

Lemma 5.2.1. *$ELSP_m(rw, \alpha, \beta) \subseteq LSP_{m+1}(rw, \alpha, \beta)$, for all $m \geq 1$ and all possible values for α, β such that $\beta \neq empty$.*

Proof. Starting from a given system Π with the total alphabet V and the terminal alphabet T, with a communication selection of a type (α, β) such that β allows at least the checking of symbols, we add one further membrane around the skin membrane of Π, as the skin membrane of the new system,

and we introduce no rewriting rule into it, but only the permitting condition $(true, out)$ and the forbidding conditions $(a, notout)$, for all $a \in V - T$. If we denote by Π' the system obtained, then we clearly have $L(\Pi) = L(\Pi')$ (no rewriting is possible in the skin membrane of Π', the two systems produce the same strings, but only strings over T can leave the system Π'). □

Corollary 5.2.2. *$LSP_*(rw, \alpha, \beta) = ELSP_*(rw, \alpha, \beta)$, for all α, β such that $\beta \neq empty$.*

In the case when all conditions are of type *empty*, after each rewriting, each string immediately changes the region; this kind of system with immediate, uncontrolled, communication may be interpreted as modelling cells with very porous membranes (or with a high thermal agitation of molecules from regions). As expected, they are not very powerful: having the conditions $(true, in), (true, out), (false, notin), (false, notout)$ associated with a region i is the same as replacing each rule $u \to v$ from region i with the couple of rules $u \to (v, in)$ and $u \to (v, out)$. As no other control is imposed on the use of rules, from Theorem 5.1.4 we obtain the following result.

Corollary 5.2.3. *$ELSP_*(rw, empty, empty) \subseteq MAT^\lambda$.*

It is of interest to mention some relations involving families $ELSP_m(rw, empty, empty)$ with a small m.

Theorem 5.2.6. *$FIN = [E]LSP_1(rw, empty, empty) \subset CF \subset ELSP_2(rw, empty, empty)$, and $LSP_2(rw, empty, empty) - CF \neq \emptyset$.*

Proof. When we have only one membrane, each string can evolve only once, because it has to exit the system after one evolution step, hence the inclusion $ELSP_1(rw, empty, empty) \subseteq FIN$ is obvious.

Also the converse inclusion holds: any given finite language $L \subset V^*$, $L = \{x_1, \ldots, x_n\}$, is generated by the system

$$\begin{aligned}\Pi = (&V \cup \{S\}, V, [_1\ ]_1, \{S\},\\ &(\{S \to x_i \mid 1 \leq i \leq n\}, \{(true, out)\}, \{(false, notout)\})).\end{aligned}$$

Now, if $G = (N, T, S, R)$ is a context-free grammar, then the language $L(G)$ is also generated by the system

$$\begin{aligned}\Pi = (&N \cup T, T, [_1[_2\ ]_2]_1, \{S\}, \emptyset,\\ &(R, \{(true, in), (true, out)\}, \{(false, notin), (false, notout)\}),\\ &(\{A \to A \mid A \in N\}, \{(true, out)\}, \{(false, notout)\})\end{aligned}$$

(after each use of a rule in the skin membrane we send the sentential form to the inner membrane, where only copying rules $A \to A$ are used; when the string is terminal, it is sent out of the system).

Consider now the system

$$\Pi = (V, V, [_1[_2\ ]_2]_1, \{AB\}, \emptyset, (R_1, P_1, F_1), (R_2, P_2, F_2)),$$

with the following components:

$$\begin{aligned}
V &= \{A, A', B, a, b, c\},\\
R_1 &= \{A \to aAb,\ A \to A',\ A' \to ab\},\\
P_1 &= \{(true, in), (true, out)\},\\
F_1 &= \{(false, notin), (false, notout)\},\\
R_2 &= \{B \to cB,\ B \to c\},\\
P_2 &= \{(true, out)\},\\
F_2 &= \{(false, notout)\}.
\end{aligned}$$

We obtain $L(\Pi) \cap a^*b^*c^* = \{a^nb^nc^n \mid n \geq 1\}$, which is not a context-free language (the current string is sent back and forth among the two membranes, at each step increasing the number of occurrences of a, b, and c; we can send a string over $\{a, b, c\}$ outside of the system only if in the last steps we use the rules $A \to A'$, $B \to c$, $A' \to ab$). This shows that $LSP_2(rw, empty, empty) - CF \neq \emptyset$ and completes the proof. □

If we use non-trivial conditions, that is different from (*empty, empty*), then the power of rewriting P systems is much increased. The following universality results suggest a trade-off between the number of membranes and the permitting/forbidding conditions we use.

Theorem 5.2.7. $LSP_2(rw, empty, sub_2) = RE$.

Proof. Let us consider a type-0 Chomsky grammar $G = (N, T, S, R)$, in the Kuroda normal form, with the non-context-free rules from R labelled in a one-to-one manner, and construct the system

$$\Pi = (V, T, [_1[_2\ ]_2]_1, \{XSX\}, \emptyset, (R_1, P_1, F_1), (R_2, P_2, F_2)),$$

with the following components:

$$\begin{aligned}
V &= T \cup N \cup \{X\}\\
&\quad \cup \{(A, r), \langle B, r\rangle \mid r : AB \to CD \in R\},\\
R_1 &= \{A \to x \mid A \to x \in R\}\\
&\quad \cup \{A \to (A, r),\ B \to \langle B, r\rangle \mid r : AB \to CD \in R\}\\
&\quad \cup \{X \to \lambda\},\\
P_1 &= \{(true, out), (true, in)\},\\
F_1 &= \{(X, notout)\}\\
&\quad \cup \{(A, notout),\ (A', notout) \mid A \in N\}\\
&\quad \cup \{((A, r), notout), (\langle B, r\rangle, notout) \mid \text{ for all } A, B \in N \text{ and}
\end{aligned}$$

$$
\begin{aligned}
& r : AB \to CD \in R\} \\
\cup\ & \{((A,r)\alpha, notin), ((A,r)(E,r'), notin), (\alpha\langle B,r\rangle, notin), \\
& (\langle B,r\rangle\langle E,r'\rangle, notin), (\langle B,r\rangle(E,r'), notin) \mid \\
& \text{for all possible } A,B,E \in N, \alpha \in N \cup T \cup \{X\}, \\
& \text{and } r, r' \text{ non-context-free rules in } R, r \neq r'\}, \\
R_2 = & \{(A,r) \to C,\ \langle B,r\rangle \to D \mid r : AB \to CD \in R\}, \\
P_2 = & \{(true, out)\}, \\
F_2 = & \{((A,r), notout), (\langle B,r\rangle, notout) \mid r : AB \to CD \in R\}.
\end{aligned}
$$

The equality $L(G) = L(\Pi)$ is easily proved: we can pass a string from membrane 1 to membrane 2 only if it is of the form $x(A,r)\langle B,r\rangle y$, where $r : AB \to CD \in R$ and $x, y \in (N \cup T)^*$, perhaps with x starting and y ending with X (the forbidding conditions prevent further occurrences of symbols of the form $(E,p), \langle E,p\rangle$ in the strings x, y; a string with only one occurrence of a symbol of the form $(E,p), \langle E,p\rangle$ cannot be sent to the inner membrane); in turn, a string $x(A,r)\langle B,r\rangle y$ can leave membrane 2 only after replacing (A,r) and $\langle B,r\rangle$ with symbols from N, hence completing the simulation of the rule r. Note also that only terminal strings with respect to G can be sent out of the system. □

When we check substrings in the permitting sense, we need both non-empty forbidding conditions and furher membranes. The permitting conditions prove to be weaker than the forbidding ones.

Theorem 5.2.8. $LSP_4(rw, sub_2, symb) = RE$.

Proof. Consider a type-0 Chomsky grammar in the Kuroda normal form $G = (N, T, S, R)$, with the non-context-free rules in R labelled in a one-to-one manner, and construct the system

$$\Pi = (V, T, [_1[_2[_3[_4\ ]_4]_3]_2]_1, \{S\}, \emptyset, \emptyset, \emptyset, (R_1, P_1, F_1), \ldots, (R_4, P_4, F_4)),$$

with the following components:

$$
\begin{aligned}
V = & T \cup N \cup \{A', A'' \mid A \in N\} \cup \{\$, \#\} \\
& \cup \{(A,r), \langle B,r\rangle \mid r : AB \to CD \in R\}, \\
R_1 = & \{A \to x \mid A \to x \in R\} \\
& \cup \{A \to (A,r) \mid r : AB \to CD \in R\} \\
& \cup \{E \to E,\ E'' \to E \mid E \in N\}, \\
P_1 = & \{(\lambda, out)\} \cup \{((A,r), in) \mid r : AB \to CD \in R\}, \\
F_1 = & \{(E, notout), (E', notout) \mid E \in N\} \\
& \cup \{((A,r), notout) \mid r : AB \to CD \in R\}, \\
R_2 = & \{B \to \langle B,r\rangle \mid r : AB \to CD \in R\}
\end{aligned}
$$

$$
\begin{aligned}
&\cup \{E' \to E'',\ E'' \to \# \mid E \in N\},\\
P_2 &= \{(E'', out) \mid E \in N\}\\
&\cup \{(\langle B, r\rangle, in) \mid r : AB \to CD \in R\},\\
F_2 &= \{(\langle B, r\rangle, notout) \mid r : AB \to CD \in R\}\\
&\cup \{(E', notin), (E'', notin) \mid E \in N\},\\
R_3 &= \{E' \to E \mid E \in N\},\\
P_3 &= \{((A, r)\langle B, r\rangle, in) \mid r : AB \to CD \in R\}\\
&\cup \{(E', out) \mid E \in N\},\\
F_3 &= \{(\$, notin), (\$, notout)\},\\
R_4 &= \{(A, r) \to C',\ \langle B, r\rangle \to D' \mid r : AB \to CD \in R\},\\
P_4 &= \{(C'D', out) \mid r : AB \to CD \in R\},\\
F_4 &= \{(\$, notout)\}.
\end{aligned}
$$

The symbol \$ never appears in a string (it is only used in some forbidding conditions which should always be false), while # is a trap symbol; once introduced it is never removed. A string can be sent from membrane 1 to membrane 2 only after using a rule of the form $A \to (A, r)$ associated with a rule $r : AB \to CD$ from R, and, conversely after using such a rule, the string *must* go to membrane 2 (this ensures the fact that exactly one rule $A \to (A, r)$ is used).

Similarly, a string can be sent from membrane 2 to membrane 3 after using one rule of the form $E \to \langle E, p\rangle$ (if no rule of this form can be used, then the string remains forever in membrane 2 and we get no output). Membrane 3 just checks whether or not the two rules used in membranes 1 and 2 correspond to the same non-context-free rule of R and the symbols which were rewritten are adjacent. If this is the case, then the string enters membrane 4, where the rule $r : AB \to CD$ is simulated; only when both C' and D' are introduced can the string leave membrane 4. One primed symbol is replaced in membrane 3 with its non-primed variant and the string immediately exits. In membrane 2 we cannot use a rule of the form $E \to \langle E, p\rangle$, because the string will remain here forever (it cannot exit because of $\langle E, p\rangle$ and cannot go to membrane 3 because of the primed symbol; even after using a rule $E' \to E''$, we cannot send the string to membranes 1 or 3). Thus, we have to use the rule $E' \to E''$ and exit. Assume that in the skin membrane we again introduce a symbol (A, r) before using the rule $E'' \to E$. The string goes to membrane 2. If we use the rule $E'' \to \#$, then the trap symbol is introduced. If we use a rule of the form $B \to \langle B, r\rangle$, then the string will remain here forever (hence, eventually, the rule $E'' \to \#$ should be used).

Therefore, in the skin membrane we have to remove the primes, and only after that can we start simulating another non-context-free rule of G. The context-free rules are simulated in the skin membrane at any time before using a rule $A \to (A, r)$. Thus, any derivation of G can be simulated in Π.

Clearly, only terminal strings with respect to G can be sent out, so we have the equality $L(G) = L(\Pi)$. □

By using one further membrane, as well as a final selection by a terminal alphabet, we can completely avoid the checking of forbidding conditions:

Theorem 5.2.9. $ELSP_5(rw, sub_2, empty) = RE$.

Proof. We start again from a type-0 Chomsky grammar in the Kuroda normal form $G = (N, T, S, R)$, with the non-context-free rules in R labelled in a one-to-one manner, and we construct the system

$$\Pi = (V, T, \mu, \emptyset, \emptyset, \emptyset, \{S\}, \emptyset, (R_1, P_1, F_1), \ldots, (R_5, P_5, F_5)),$$

with the following components:

$$\begin{aligned}
V &= T \cup N \cup \{A', A'', A''' \mid A \in N\} \cup \{f, \#\} \\
&\quad \cup \{(A, r), \langle B, r\rangle \mid r : AB \to CD \in R\}, \\
\mu &= [_1[_2[_4\ ]_4]_2[_3[_5\ ]_5]_3]_1, \\
R_1 &= \{f \to \lambda\} \\
&\quad \cup \{A' \to \#,\ A''' \to \#, A'' \to A \mid A \in N\}, \\
P_1 &= \{(\lambda, out)\} \\
&\quad \cup \{((A, r)\langle B, r\rangle, in) \mid r : AB \to CD \in R\} \\
&\quad \cup \{(A', in) \mid A \in N\}, \\
R_2 &= \{B \to \langle B, r\rangle \mid r : AB \to CD \in R\} \\
&\quad \cup \{E' \to E''', E''' \to \# \mid E \in N\}, \\
P_2 &= \{(f, out)\} \\
&\quad \cup \{(E''', in) \mid E \in N\} \\
&\quad \cup \{(\langle B, r\rangle, out) \mid r : AB \to CD \in R\}, \\
R_3 &= \{(A, r) \to C',\ \langle B, r\rangle \to \# \mid r : AB \to CD \in R\}, \\
P_3 &= \{(E', in), (E'', out) \mid E \in N\}, \\
R_4 &= \{A \to (A, r) \mid r : AB \to CD \in R\} \\
&\quad \cup \{A \to x,\ A \to xf \mid A \to x \in R\} \\
&\quad \cup \{A''' \to A \mid A \in N\}, \\
P_4 &= \{(f, out)\} \cup \{((A, r), out) \mid r : AB \to CD \in R\}, \\
R_5 &= \{\langle B, r\rangle \to D'' \mid r : AB \to CD \in R\} \\
&\quad \cup \{C' \to \# \mid C \in N\}, \\
P_5 &= \{(C'D'', out) \mid r : AB \to CD \in R\}.
\end{aligned}$$

All sets of forbidding conditions consist of the pairs $(false, notin)$, $(false, notout)$.

This system works as follows. We start in membrane 4 with the axiom of G. The context-free rules of G can be simulated here. If a terminal rule $A \to xf$ is used in membrane 4, then the string exits; if it is not terminal and a rule $B \to \langle B, r\rangle$ is used in membrane 2, then the string goes to membrane 1 and from here out of the system, but it is not accepted in the generated language. If the string is terminal, then it exits (f is erased in the skin membrane) and is introduced in $L(\Pi)$.

Assume that a string w is rewritten in membrane 4 by a rule $A \to (A, r)$ associated with a rule $r : AB \to CD \in R$. It exits; if no rule can be applied in membrane 2, then no output is produced. Assume that a rule $E \to \langle E, p\rangle$ is used in membrane 2. The string is immediately sent to membrane 1, where no rule can be applied. If the two tuple symbols are not associated with the same rule from R, then the string is sent out of the system, but it is not a terminal one. If the string is of the form $w_1(A, r)\langle B, r\rangle w_2$, for some $r : AB \to CD \in R$, then it can be sent to a lower membrane. If it arrives back in membrane 2, then it will exit either unchanged – if no rule of the form $E \to \langle E, p\rangle$ can be used – or after introducing one further symbol of the form $\langle E, p\rangle$.

The process is repeated; eventually, the string will arrive in membrane 3 (otherwise we either continue between membranes 1 and 2 or send the string out and it is not terminal). In membrane 3, the unique copy of the symbol (A, r) is replaced by C' and the string is sent to membrane 5. Here we also replace a symbol $\langle E, p\rangle$ with H''. The string exits only if the primed symbols, $C'H''$, correspond to a rule $r : AB \to CD \in R$, otherwise the rule $C' \to \#$ introduces the trap symbol $\#$. Assume that we are in the former case, hence the string is sent to membrane 3. If the string still contains a symbol of the form $\langle E, p\rangle$, then the trap symbol is introduced, otherwise the string is sent unchanged to the skin membrane. Consequently, we have to replace exactly two symbols A, B with $(A, r), \langle B, r\rangle$, in adjacent positions, hence this corresponds to simulating the rule $r : AB \to CD \in R$.

Now, in the skin membrane (if not using the rule $C' \to \#$) we use the rule $D'' \to D$ and the string is sent to one of membranes 2 and 3. From membrane 3 the string should enter membrane 5 unchanged, and here the rule $C' \to \#$ is used. Thus, we have to send the string to membrane 2. We have two cases. If we use a rule $E \to \langle E, p\rangle$, then the string is sent back to the skin membrane, where the only applicable rule is $C' \to \#$ and no terminal string will ever be obtained. If in membrane 2 we use the rule $C' \to C'''$, then the string is sent to membrane 4, where again we have two cases. If we use the rule $C''' \to C$, then we again have a string over $(N \cup T)^*$, and the process can be iterated. If, before using the rule $C''' \to C$, we use a rule $A \to (A, r)$, then the string should go to membrane 2. If we use the rule $C''' \to \#$ here, then the computation will produce nothing. If we use a rule $E \to \langle E, p\rangle$, then the string is sent to membrane 1, where the only applicable rule is $C''' \to \#$.

Consequently, we have to completely simulate the rule $r : AB \to CD \in R$, ending by using the rule $C''' \to C$ in membrane 4. Thus, $L(G) = L(\Pi)$. $\square$

Somewhat surprisingly, one additional membrane with respect to those used in the system from the proof of the previous theorem suffices in order to characterize RE even when checking symbols only as permitting conditions. This result should be compared with the fact that random context grammars without appearance checking do not generate all recursively enumerable languages; the condition in that case is imposed at the level of rules, about their applicability, but also in the form of symbol checking. We again have a confirmation that distributed computing (hence communication among separate computing agents) is a powerful strategy.

Theorem 5.2.10. $ELSP_6(rw, symb, empty) = LSP_6(rw, symb, symb) = RE$.

Proof. This time we start from a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the strong binary normal form. Each terminal matrix $(X \to \lambda, A \to x), X \in N_1, A \in N_2, x \in T^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. We continue to label the obtained matrix in the same way as the original one.

We construct the extended rewriting P system (of degree 6)

$$\Pi = (V, T, \mu, M_1, \ldots, M_6, (R_1, P_1, F_1), \ldots, (R_6, P_6, F_6)),$$

with the following components:

$$\begin{aligned}
V &= T \cup N_1 \cup N_2 \cup \{(X_i, j) \mid X \in N_1, 1 \le i \le k, 0 \le j \le k\} \\
&\quad \cup \{X_i \mid X \in N_1, i \in lab_1 \cup lab_2\} \\
&\quad \cup \{A_i, (A_i, j) \mid A \in N_2, 1 \le i \le k, 0 \le j \le k\} \\
&\quad \cup \{X', X'', X''' \mid X \in N_1\} \\
&\quad \cup \{f'', \#\}, \\
\mu &= [_1[_2[_3[_4\ ]_4]_3]_2[_5\ ]_5[_6\ ]_6]_1, \\
M_1 &= \{X_{init} A_{init}\}, \\
M_i &= \emptyset, \text{ for all } i = 2, \ldots, 6,
\end{aligned}$$

and with the following triples $(R_i, P_i, F_i), 1 \le i \le 6$ (the membranes with labels 2, 3, 4 will be used for simulating matrices $m_i, 1 \le i \le k$, while membranes with labels 5, 6 will simulate the matrices with labels in lab_1, lab_2, respectively):

$$\begin{aligned}
R_1 \ : \ & X \to X_i, \text{ for all } m_i : (X \to Y, B^{(j)} \to \#),\ k+1 \le i \le n, \\
& A \to (A_i, 0), \text{ for all } m_i : (X \to \alpha, A \to x), 1 \le i \le k, \\
& X_i \to Y', \text{ for all } m_i : (X \to \alpha, D \to \beta), 1 \le i \le n, \\
& X'' \to X''', \text{ for all } X \in N_1, \\
& f'' \to \lambda; \\
P_1 = & \{((A_i, 0), in) \mid A \in N_2, i \in lab_0\}
\end{aligned}$$

$$
\begin{aligned}
&\cup \{(X_i, in), (X', in), (X''', in) \mid X \in N_1,\ k+1 \le i \le n,\\
&\cup \{(a, out) \mid a \in T\};\\
F_1 &= \{(false, notin), (false, notout)\};\\
R_2 &: X_i \to \#, \text{ for all } X \in N_1, i \in lab_1 \cup lab_2,\\
&\quad X \to (X_i, 0), \text{ for all } X \in N_1, 1 \le i \le k,\\
&\quad A_i \to x, \text{ for all } m_i : (X \to Y, A \to x), 1 \le i \le k,\\
&\quad (A_i, j) \to \#, \text{ for all } A \in N_2, 1 \le j < i \le k,\\
&\quad X'' \to \#, \text{ for all } X \in N_1 \cup \{f\},\\
&\quad X''' \to X, \text{ for all } X \in N_1;\\
P_2 &= \{((X_i, 0), in) \mid X \in N_1, 1 \le i \le k\}\\
&\cup \{(X, out), (X'', out) \mid X \in N_1 \cup \{f\}\};\\
F_2 &= \{(false, notin), (false, notout)\};\\
R_3 &: (X_i, j) \to (X_i, j+1), \text{ for all } X \in N_1, 0 \le j < i \le k,\\
&\quad (X_i, i) \to \alpha'', \text{ for all } m_i : (X \to \alpha, A \to x), 1 \le i \le k;\\
P_3 &= \{((X_i, j), in) \mid X \in N_1, 1 \le j \le i \le k\}\\
&\cup \{(X'', out) \mid X \in N_1 \cup \{f\}\};\\
F_3 &= \{(false, notin), (false, notout)\};\\
R_4 &: (A_i, j) \to (A_i, j+1), \text{ for all } A \in N_2, 0 \le j < i \le k,\\
&\quad (A_i, i) \to A_i, \text{ for all } m_i : (X \to \alpha, A \to x), 1 \le i \le k,\\
&\quad A_i \to \#, \text{ for all } A \in N_2, 1 \le i \le k;\\
P_4 &= \{((A_i, j), out) \mid A \in N_2, 1 \le j < i \le k\}\\
&\cup \{(A_i, out) \mid A \in N_2, 1 \le i \le k\};\\
F_4 &= \{(false, notout)\};\\
R_{4+j,1} &: X_i \to \#, \text{ for all } X \in N_1, i \notin lab_j,\\
&\quad (A_i, 0) \to \#, \text{ for all } A \in N_2, 1 \le i \le k,\\
&\quad B^{(j)} \to \#,\\
&\quad X' \to X, \text{ for all } X \in N_1,\\
&\quad X''' \to \#, \text{ for all } X \in N_1;\\
P_{4+j} &= \{(a, out) \mid a \in T\};\\
F_{4+j} &= \{(false, notout)\}, \text{ for } j = 1, 2.
\end{aligned}
$$

Let us examine the work of this system.

Only strings over T are accepted in the generated language; # is a trap symbol, once introduced it will never be removed, hence the string will never turn to be terminal. From the skin membrane at any moment we can send out a string which contains at least one terminal symbol, but if any symbol not in T is present, then the string is not accepted in the generated language.

At any moment we have exactly one string in the system; initially, this is $X_{init}A_{init}$. Assume that in region 1 we have a string of the general form Xw, with $X \in N_1$ and $w \in (N_2 \cup T)^*$. We have to use a rule of the form $X \to X_i$, for $k+1 \leq i \leq n$, or a rule of the form $A \to (A_i, 0)$, for some $A \in N_2, 1 \leq i \leq k$. In the first case we start the simulation of a matrix with appearance checking, in the second case we start the simulation of the matrix $m_i, 1 \leq i \leq k$ (without appearance checking).

Assume that we are in the former case. The string must be communicated to a lower level membrane, that is one of membranes 2, 5, 6. In membrane 2 and in that of membranes 5, 6 which is different from $4+j$, the only applicable rule is $X_i \to \#$, hence the computation will not produce a terminal string. If the string X_iw arrives in the right membrane $4+j$, that is the membrane with $i \in lab_j$, and $B^{(j)}$ is not present in it, then no rule can be applied; at the next step the string exits unchanged. In this way we know that w does not contain the symbol $B^{(j)}$; if $B^{(j)}$ is present, then the symbol # is introduced. In the skin membrane we use the rule $X_i \to Y'$ which corresponds to the matrix $m_i : (X \to Y, B^{(j)} \to \#)$ from M. The string is again sent to a lower membrane; if it arrives in membrane 2, then it will remain here forever, unchanged; from any membrane 5, 6 it exits with Y' replaced with Y. This completes the simulation of the matrix m_i. The obtained string is of the initial form, hence the process can be iterated. If in the skin membrane we do not use the rule $X_i \to Y'$, but a rule of the form $A \to (A_i, 0)$, then the string can be sent to a lower membrane; in membrane 2 we can use only the rule $X_i \to \#$, and in membranes 5, 6 we can apply only the rule $(A_i, 0) \to \#$, hence no terminal string will be obtained.

Assume now that we have started by introducing a symbol $(A_i, 0)$, corresponding to a matrix $m_i : (X \to \alpha, A \to x), 1 \leq i \leq k$, with $\alpha \in N_1 \cup \{f\}$. The string should be communicated to a lower membrane. In membranes 5 and 6 the symbol # will be introduced, hence we have to send the string to membrane 2. The only applicable rule is $X \to (X_j, 0)$, for some $1 \leq j \leq k$. The string is sent to membrane 3, where $(X_j, 0)$ is replaced with $(X_j, 1)$. The obtained string is sent to membrane 4, where $(A_i, 0)$ is replaced with $(A_i, 1)$. The string returns to membrane 3. From now on, the string will go back and forth between membranes 3 and 4, and the second component of the symbols $(A_i, s), (X_j, t)$ is alternatively increased.

Now, we distinguish three cases:

Case 1: $i < j$. This means that at some step in membrane 4 we receive from membrane 3 a string of the form $(X_j, i)w_1(A_i, i-1)w_2$. We replace $(A_i, i-1)$ with (A_i, i) and no communication is possible (note that $((A_i, i), out)$ is not in P_4), hence one more rewriting is necessary. We replace (A_i, i) with A_i and the string is sent out. In membrane 3 we replace (X_j, i) with $(X_j, i+1)$ (this is possible, because $i+1 \leq j$), and the string is sent back to membrane 4, where the trap symbol is introduced (the rewriting has priority over checking the communication conditions).

Case 2: $i > j$. At some moment we produce in membrane 3 a string of the form $(X_j, j)w_1(A_i, j-1)w_2$, which is sent to membrane 4. Here we replace $(A_i, j-1)$ with (A_i, j), the string exits, in membrane 3 we replace (X_j, j) with $\alpha'', \alpha \in N_1 \cup \{f\}$, and the string is sent out. In membrane 2 we apply $(A_i, j) \to \#$ or $\alpha'' \to \#$, hence again the string will never lead to a terminal one.

Case 3: $i = j$. At some moment we pass a string $(X_i, i)w_1(A_i, i-1)w_2$ from membrane 3 to membrane 4. In membrane 4 we replace $(A_i, i-1)$ with (A_i, i) and, because we cannot exit, we replace this latter symbol with A_i. Returning to membrane 3, we replace (X_i, i) with α'' and we send the string $\alpha'' w_1 A_i w_2$ to membrane 2. We have to continue by using the rule $A_i \to x$ and the string $\alpha'' w_1 x w_2$ is sent to the skin membrane.

We have two subcases here. If we use the rule $\alpha'' \to \alpha'''$, then the string is sent to membrane 2 again (or to membranes 5, 6, where the trap symbol # will be introduced). The only applicable rule here is $\alpha''' \to \alpha$, which completes the simulation of the matrix m_i. Moreover, in membrane 1 we get a string of the form we started with, hence – if $\alpha \in N_1$ – the process can be iterated. If in the skin membrane we use a rule of the form $C \to (C_l, 0)$, then a string of the form $\alpha'' z_1(C_l, 0)z_2$ is sent to one of membranes 2, 5, 6 and in all of them the trap symbol is produced immediately.

Therefore, the only correct way (that is, leading to a terminal string) to proceed is to correctly simulate matrices from M.

At the moment when a matrix $(X \to f, A \to x)$ is simulated, the computation must stop: we have the string $f''z$ in the skin membrane; if we use a rule $C \to (C_l, 0)$, then the string $f'' z_1(C_l, 0)z_2$ is sent to any membrane 2, 5, 6, and the trap symbol is produced; therefore, the rule $f'' \to \lambda$ must be used in membrane 1 for rewriting $f''z$. The string should exit (no condition allows the movement to a lower membrane). If the string is terminal, then it belongs to the language $L(\Pi)$; if not, then it is "lost".

If at any moment we get a string of the form $Xw, w \in T^*$, then we will never get a terminal string: we can simulate at most matrices m_i, $k+1 \leq i \leq n$, but we cannot remove the symbol from N_1 present in the string. In turn, if we obtain a string without any symbol from N_1, but containing symbols from N_2, then this means that it is of the form $f''w$, hence, as we have seen above, the computation cannot lead to a terminal string.

Consequently, $L(\Pi) = L(G)$, and we have the equality $ELSR_6(rw, symb, empty) = RE$.

For the non-extended case we replace all sets $F_i, 2 \leq i \leq 6$, with $\{(\$, notin), (\$, notout)\}$, where \$ is a new symbol, to be added to V, and we consider

$$F_1 = \{(\$, notin)\} \cup \{(\alpha, notout) \mid \alpha \in V - T\}.$$

Only strings consisting of symbols in T can exit the system, but all the strings are processed as described above, hence the language is not changed. This completes the proof. □

5.2.4 Replicated Rewriting

We now pass to an interesting extension of the rewriting operation itself, somewhat suggested by the fact that in biology one can replicate certain molecules (for instance, the DNA molecules are replicated both *in vivo*, by genetic mechanisms, and *in vitro*, by PCR – polymerase chain reaction), and with important applications in devising membrane-systems-based algorithms able to solve **NP**-complete problems in polynomial time (see Section 7.4).

Specifically, we consider the operation of *rewriting with replication* (or of *replicated rewriting*), by means of rules of the form $a \to u_1||u_2||\dots||u_n$, where $u_1, \dots, u_n$ are strings over a given alphabet. When rewriting a string w_1aw_2 by such a rule we obtain at the same time n strings, namely, $w_1u_1w_2$, $w_1u_2w_2, \dots,$ $w_1u_nw_2$.

In the case of membrane systems, we have to specify target indications for each of the n obtained strings, hence the rules will be of the form $a \to (u_1, tar_1)||\dots||(u_n, tar_n)$, with the obvious meaning that the string $w_1u_iw_2$ will go to the region indicated by tar_i, $1 \le i \le n$.

The number n from a rule $a \to (u_1, tar_1)||\dots||(u_n, tar_n)$ is called *the replication index* of the rule. Clearly, when $n = 1$, a rule as above is a usual rewriting rule.

We denote by $[E]LSP_m(rw, in, repl_n)$ the family of languages generated by string–object P systems of degree at most $m \ge 1$ using replicated rewriting rules with the replication index at most $n \ge 1$ (and the target commands *here, out, in*). The subscripts m, n are replaced with $*$ when no bound is imposed on the corresponding parameter.

In a system with replicated rewriting we start as usual from a finite set of strings placed in the regions of the initial configuration, and we rewrite and replicate them independently to each other; this is true also during the computation, when there is no interaction between strings. Such an interaction can appear by means of a priority relation among rules, or by means of actions δ, τ – the proof of Theorem 5.1.11 illustrates the way several strings can influence each other by means of membrane thickness changes. However, when a priority or the actions δ, τ are present we get universality with standard rewriting rules, hence we do not need the replication.

On the other hand, because of the independence of evolution of strings, and because of the context-free form of the rewriting, a system with replicated rewriting can be simulated by one without replication: instead of a rule $a \to (u_1, tar_1)||\dots||(u_n, tar_n)$, we consider all rules $a \to (u_1, tar_1)$, $a \to (u_2, tar_2), \dots, a \to (u_n, tar_n)$; each string produced by using the replication rule can be obtained by using, in alternate computations, one of the n rewriting rules. That is, the following equalities hold.

Theorem 5.2.11. $[E]LSP_m(rw, in, repl_*) = [E]LSP_m(rw, in, repl_1) = [E]LSP_m(rw, in)$, *for all* $m \geq 1$ *or* $m = *$.

Thus, replication alone, or with priorities, or with δ, τ, is not of interest from the generative point of view. (It is of interest from the computational complexity point of view, as we will see in Section 7.4.) However, there is one further way of linking the evolution of separate string–objects: considering halting computations, as for the systems with symbol–objects. If we have two strings in the system, one producing a terminal string and the other one evolving forever, we may say that the computation is not successful as a whole, and no string is accepted. Because the two strings can be obtained by replication, hence starting from a common "ancestor", in this way we get a surprisingly powerful technique for reaching the universality again (without priorities and without actions δ, τ).

Let us denote by $[E]LSP_m(rw, in, repl_n, halt)$ the family of languages generated by P systems as above, but with the halting condition imposed (we accept the strings sent out of the system during a computation only if the computation stops).

Theorem 5.2.12. $LSP_6(rw, in, repl_2, halt) = RE$.

Proof. Consider a matrix grammar with appearance checking, $G = (N, T, S, M, F)$, in the strong binary normal form, with the standard notation. Each terminal matrix $(X \to \lambda, A \to x), X \in N_1, A \in N_2, x \in T^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. We continue to label the obtained matrix in the same way as the original one.

We construct the (non-extended) P system (of degree 6) with replicated rewriting

$$\Pi = (V, V, \mu, M_1, \ldots, M_6, R_1, \ldots, R_6),$$

with the following components:

$$\begin{aligned}
V &= T \cup N_1 \cup N_2 \cup \{(X_i, j), (X_i, j)' \mid X \in N_1, 1 \leq i \leq k, 0 \leq j \leq k\} \\
&\quad \cup \{X_i \mid X \in N_1, k+1 \leq j \leq n\} \\
&\quad \cup \{(A_i, j), (A_i, j)' \mid A \in N_2, 1 \leq i \leq k, 0 \leq j \leq k\} \\
&\quad \cup \{f, E, E', \#\}, \\
\mu &= [_1[_2[_3[_4\ ]_4]_3]_2[_5\ ]_5[_6\ ]_6]_1, \\
M_1 &= \{X_{init} A_{init}\}, \\
M_i &= \emptyset, \text{ for all } i = 2, \ldots, 6,
\end{aligned}$$

and with the following sets of rules (the membranes with labels 2, 3, 4 will be used for simulating matrices $m_i, 1 \leq i \leq k$, while the membranes with labels 5, 6 will simulate the matrices with labels in lab_1, lab_2, respectively):

$$
\begin{aligned}
R_1 : & X \to ((X_i, 0), here), \text{ and} \\
& A \to ((A_i, 0), in), \text{ for all } m_i : (X \to \alpha, A \to x), 1 \le i \le k, \\
& X \to (Y, here)||(X_i, in), \text{ for all } m_i : (X \to Y, B^{(j)} \to \#), \\
& \qquad k+1 \le i \le n, \\
& f \to (\lambda, out)||(f, in); \\
R_2 : & X_i \to (\#, here), \text{ for all } X \in N_1, k+1 \le i \le n, \\
& (X_i, 0) \to ((X_i, 0), in), \text{ for all } X \in N_1, 1 \le i \le k, \\
& E' \to (\lambda, out), \\
& A \to (\#, here), \text{ for all } A \in N_2, \\
& \# \to (\#, here); \\
R_3 : & (A_i, j) \to ((A_i, j+1)', here), \text{ for all } A \in N_2, 0 \le j < i \le k, \\
& (X_i, j) \to ((X_i, j+1)', in), \text{ for all } X \in N_1, 0 \le j < i \le k, \\
& E \to (\lambda, out); \\
R_4 : & (X_i, j)' \to ((X_i, j), here), for X \in N_1, 1 \le j < i \le k, \\
& (A_i, j)' \to ((A_i, j), out), \text{ for all } A \in N_2, 1 \le j < i \le k, \\
& (X_i, i)' \to (\alpha E', here), \text{ and} \\
& (A_i, i)' \to (xE, out), \text{ for all } m_i : (X \to \alpha, A \to x), 1 \le i \le k; \\
R_{4+j} : & (A_i, 0) \to (\#, here), \text{ for all } A \in N_2, 1 \le i \le k, \\
& X_i \to (\#, here), \text{ for all } X \in N_1, i \notin lab_j, \\
& B^{(j)} \to (\#, here), \\
& f \to (\#, here), \\
& \# \to (\#, here), \text{ for } j = 1, 2.
\end{aligned}
$$

The simulation of matrices $m_i, 1 \le i \le k$, is performed (by membranes 1, 2, 3, 4) in a way which has been used already several times – in particular, in the proofs of Theorems 5.1.5 and 5.2.10 – ensuring that both rules of a matrix m_i are simulated by means of the interplay of the second compounds of symbols $(X_i, j), (A_p, q)$: the trap symbol # is not introduced only when $i = p$ and $m_i : (X \to \alpha, A \to x)$. The details are left to the reader, and we discuss only the way of using the replication in order to simulate matrices with appearance checking rules and to check that we send out only terminal strings with respect to G.

Thus, assume that in the skin membrane we rewrite a string Xw by means of a rule of the form $X \to (Y, here)||(X_i, in)$, for $k+1 \le i \le n$ with $i \in lab_j$ for some $j = 1, 2$ (this means that we intend to simulate the matrix m_i). The string Yw remains in the skin membrane and can be processed as above, and the string $X_i w$ is sent to one of membranes 2, 5, 6. In membrane 2 it immediately introduces the trap symbol. The same happens in that of membranes 5, 6 which is different from $4+j$. If the string arrives in the correct place, that is in membrane $4+j$, then the trap symbol # is introduced if and

only if the string contains the symbol $B^{(j)}$. Therefore, the computation will halt if and only if the simulation of the matrix $m_i : (X \to Y, B^{(j)} \to \#), i \in lab_j, j = 1, 2$, is correct.

At the moment when we simulate a matrix $m_i : (X \to f, A \to x)$, we return to the skin membrane a string of the form fz. If z is not terminal and a rule of the form $A \to ((A_i, 0), in)$ is used, then either the string will remain forever in membrane 2, or the trap symbol will be introduced in membranes 5, 6 by one of the rules $(A_i, 0) \to (\#, here)$ or by $f \to (\#, here)$. If we do not use a rule $A \to ((A_i, 0), in)$, then in the skin membrane we have to use the rule $f \to (\lambda, out)||(f, in)$. The string z is sent out and the string fz is sent to one of the lower membranes. In membranes 5, 6 we immediately introduce the trap symbol. In membrane 2, if any non-terminal symbol $A \in N_2$ is still present in z, then a rule $A \to (\#, here)$ can be used and again the computation will never stop. Otherwise, the string remains here, no rule can be applied to it, but we know that the string z is a terminal one.

In conclusion, $L(\Pi)$ contains exactly the terminal strings which can be generated by G; that is, it is equal to $L(G)$. This completes the proof. □

It is an *open problem* (Problem Q14) whether or not the previous result is optimal, or the number of used membranes can be decreased. Note that the degree of the system from Theorem 5.2.12, namely six, is equal to the highest among the systems for which we have obtained universality, and this supports the conjecture that probably the result can be improved (perhaps by making better use of the replication).

Let us observe that when the matrix grammar from which we start the construction in the proof of Theorem 5.2.12 is without appearance checking, then four membranes suffice. That is, we have the following inclusion.

Corollary 5.2.4. $MAT^\lambda \subseteq LSP_4(rw, in, repl_2, halt)$.

Note that from Theorems 5.1.5 and 5.2.11 we already know that $MAT^\lambda \subseteq ELSP_4(rw, in, repl_1)$, hence with the use of a terminal alphabet; replication together with the halting condition compensate for the terminal alphabet extension, and, moreover, lead to languages not in MAT^λ, which is not true for systems of type (rw, in) – see Corollary 5.1.2.

Actually, a result which is stronger than $LSP_4(rw, in, repl_2, halt) - MAT^\lambda \neq \emptyset$ is true (we take into account that $ET0L - MAT^\lambda \neq \emptyset$).

Theorem 5.2.13. $LSP_4(rw, in, repl_2, halt) - ET0L \neq \emptyset$.

Proof. Consider the following system:

$$\begin{aligned}
\Pi &= (V, V, [_1[_2[_3[_4\ ]_4]_3]_2]_1, \emptyset, \{c\}, \emptyset, \emptyset, R_1, R_2, R_3, R_4),\\
V &= \{a, b, c, d, e, e', \#\},\\
R_1 &= \{c'' \to (\lambda, out),\ c \to (\#, here),\\
&\qquad e' \to (\#, here),\ \# \to (\#, here)\},
\end{aligned}$$

$$\begin{aligned}
R_2 &= \{c \to (cde, here),\ c \to (c'de, here),\ e' \to (e, here),\\
&\quad d \to (a, in)||(a, out),\ c' \to (c'', out)\},\\
R_3 &= \{e \to (be', here),\ e \to (be', in)||(be', out)\},\\
R_4 &= \{e \to (\#, here),\ \# \to (\#, here)\}.
\end{aligned}$$

We start in membrane 2 with the string c and we produce a string $c'(de)^n$, for some $n \geq 1$, by repeatedly using the rule $c \to (cde, here)$ and then the rule $c \to (c'de, here)$. When using the rule $d \to (a, in)||(a, out)$ we must have already used the rule $c \to (c'de, here)$, otherwise the trap symbol # would be produced in the skin membrane and the computation would never stop. In membrane 3, each occurrence of e introduces a copy of b and is replaced by e'. This replacement should be complete at the moment when using the rule $e \to (be', in)||(be', out)$, otherwise # is introduced in membrane 4. In membrane 2, each e' is replaced by e, then the rule $d \to (a, in)||(a, out)$ is again used (in this way we check whether or not any copy of e' is still present; in the affirmative case the rule $e' \to (\#, here)$ is used in the skin membrane). In membrane 3 each e will again introduce a copy of b. The process can be iterated at most n times, the number of copies of the symbol d in the string. The string can be sent out by using the rules $c' \to (c'', out), c'' \to (\lambda, out)$, from membranes 2 and 1, respectively. Thus, we obtain

$$L(\Pi) \cap \{a, b, e\}^* = \{(ab^n e)^n \mid n \geq 1\}$$

(the intersection with $\{a, b, e\}^*$ ensures that no symbol d is present in the string).

The family $ET0L$ is closed under erasing morphisms and intersection with regular languages, while $\{(ab^n)^n \mid n \geq 1\}$ is not an ET0L language (this is language L_{17} from Subsection 2.3.5). Consequently, $L(\Pi) \notin ET0L$. □

This result suggests that we should look for a result like that in Corollary 5.2.4 also for the family $ET0L$ – we can obtain it, but only when using five membranes.

Theorem 5.2.14. $ET0L \subset ELSP_5(rw, in, repl_2, halt)$.

Proof. The properness follows from the previous theorem; we have only to prove the inclusion.

Consider an ET0L system with only two tables, $G = (V, T, w, R_1, R_2)$. Assume that $R_i = \{a_{i,j} \to x_{i,j} \mid 1 \leq j \leq n_i\}$, for some $n_i \geq 1, i = 1, 2$. (Remember that each table in an ET0L system is *complete*: a rule $a \to x$ exists in each table for each symbol $a \in V$.) We construct the following P system (of degree 5):

$$\begin{aligned}
\Pi &= (V', T, \mu, \{Xw\}, \emptyset, \emptyset, \emptyset, \emptyset, R'_1, R'_2, R'_3, R'_4, R'_5),\\
V' &= V \cup \{a', a'' \mid a \in V\} \cup \{X, X_1, X_2, \#\},
\end{aligned}$$

$$
\begin{aligned}
\mu &= [_1[_2[_3\ ]_3]_2[_4\ ]_4[_5\ ]_5]_1, \\
R_1' &= \{X \to (\lambda, out)\} \\
&\quad \cup \{a \to (a', here),\ a \to (a'', here) \mid a \in V\} \\
&\quad \cup \{X \to (X, in)||(X_1, in),\ X \to (X, in)||(X_2, in)\}, \\
R_2' &= \{X_1 \to (\#, here),\ X_2 \to (\#, here), \\
&\qquad \# \to (\#, here),\ X \to (X, in)||(X, out)\} \\
&\quad \cup \{a_{1,j}' \to (x_{1,j}, here) \mid 1 \le j \le n_1\} \\
&\quad \cup \{a_{2,j}'' \to (x_{2,j}, here) \mid 1 \le j \le n_2\}, \\
R_3' &= \{a' \to (\#, here),\ a'' \to (\#, here) \mid a \in V\} \\
&\quad \cup \{\# \to (\#, here)\}, \\
R_4' &= \{X \to (\#, here),\ X_2 \to (\#, here),\ \# \to (\#, here)\} \\
&\quad \cup \{a \to (\#, here),\ a'' \to (\#, here) \mid a \in V\}, \\
R_5' &= \{X \to (\#, here),\ X_1 \to (\#, here),\ \# \to (\#, here)\} \\
&\quad \cup \{a \to (\#, here),\ a' \to (\#, here) \mid a \in V\}.
\end{aligned}
$$

This system works as follows. We start by placing the axiom string of G together with the auxiliary symbol X in the skin membrane. Assume that we have here a string Xz, for arbitrary $X \in N_1$ and $z \in V^*$. By using the rules from R_1', some symbols are primed or double primed; eventually we also use one of the rules $X \to (X, in)||(X_1, in)$, $X \to (X, in)||(X_2, in)$, the string is duplicated, and the copies are sent to lower membranes. The copy which contains the symbol X should arrive in membrane 2, otherwise the trap symbol # is introduced in membranes 4, 5; similarly, in membrane 2 the symbols X_1, X_2 are replaced by #. Moreover, if we have applied the first rule, that is the string contains the symbol X_1, then it must arrive in membrane 4 (in membrane 5 this symbol is replaced by #), and if we have applied the second rule, that is the string contains the symbol X_2, then it must arrive in membrane 5. In this way we check two facts: that *all* symbols a from z were replaced by a' or by a'', and, moreover, that we have introduced either only single primed symbols or only double primed symbols, but not both of them, otherwise the symbol # is introduced either in membrane 4, or in membrane 5. In this way, in membrane 2 we will correctly simulate either the table R_1 or the table R_2, without mixing the rules of the two tables.

In membrane 2, when simulating the rules of the two tables, we return to non-primed symbols. When using the rule $X \to (X, out)||(X, in)$ in membrane 2, a copy of the string is sent to the skin membrane and one is sent to membrane 3. In this latter membrane we check whether or not all primed or double primed symbols were replaced; that is, whether or not the simulation of the tables is done in the parallel manner specific to L systems (all symbols are rewritten in each step, and this is possible because the tables are complete). Only in the affirmative case can we continue without introducing the trap symbol.

We return to the skin membrane with a string similar to the one we started with. The process can be iterated. At any moment, we can use the rule $X \to (\lambda, out)$ and the string is sent out. If it is terminal, then it is accepted in the language of Π; if not, then it is "lost". Consequently, $L(\Pi) = L(G)$. □

As expected, the E0L languages can be generated by P systems with a smaller number of membranes.

Theorem 5.2.15. $E0L \subset ELSP_4(rw, in, repl_2, halt)$.

Proof. The strictness follows from Theorem 5.2.13, so we have to prove only the inclusion. Let $G = (V, T, w, R)$ be an E0L system. Let h be the morphism defined by $h(a) = a', a \in V$, where a' is a new symbol associated with a. We construct the P system (of degree 4):

$$\begin{aligned}
\Pi &= (V', T, [_1[_2[_3[_4\ ]_4]_3]_2]_1, \emptyset, \{cw\}, \emptyset, \emptyset, R_1, R_2, R_3, R_4),\\
V' &= V \cup \{a' \mid a \in V\} \cup \{c, c', \#\},\\
R_1 &= \{c' \to (\lambda, out),\ \# \to (\#, here)\}\\
&\quad \cup \{a \to (\#, here) \mid a \in V\},\\
R_2 &= \{a \to (h(x), here),\ a \to (h(x), in)||(h(x), out) \mid a \to x \in R\}\\
&\quad \cup \{c \to (c', out)\},\\
R_3 &= \{a' \to (a, here),\ a' \to (a, out)||(a, in) \mid a \in V\},\\
R_4 &= \{a' \to (\#, here),\ \# \to (\#, here)\}.
\end{aligned}$$

We start in membrane 2, from cw, with w being the axiom of G. After simulating some rules of R we use a rule $a \to (h(x), in)||(h(x), out)$, and a copy of the string is sent to the skin membrane and one to membrane 3. In the skin membrane we check whether a derivation step in G was completely simulated in membrane 2 (if any symbol $a \in V$ remains unprimed, then the rule $a \to (\#, here)$ must be used in membrane 1). In turn, in membrane 3 we replace primed symbols by their unprimed versions and at some moment we use a rule $a' \to (a, out)||(a, in)$. In membrane 4 we check whether or not the unpriming is complete (otherwise the trap symbol $\#$ is introduced). In the affirmative case, in membrane 2 we can continue the simulation of derivation steps with respect to G. At any moment, in membrane 2 we can use the rule $c \to (c', out)$, and the string is sent to membrane 1, where $c' \to (\lambda, out)$ sends the string outside the system. If it is terminal with respect to G, then it is accepted in $L(\Pi)$. Therefore, $L(G) = L(\Pi)$. □

By using one furher membrane, we can avoid the use of the terminal alphabet.

Corollary 5.2.5. $E0L \subset LSP_5(rw, in, repl_2, halt)$.

Proof. We proceed exactly as in the proof of Theorem 5.2.15, but we change the system Π as follows. We add one further membrane, $[_5\ ]_5$, directly into the skin membrane. Instead of the rule $c' \to (\lambda, out) \in R_1$, we consider the rule $c' \to (\lambda, out)||(c'', in)$, where c'' is a new symbol. We add the rule $c'' \to (\#, out)$ to R_2. Then, we also take

$$R_5 = \{a \to (\#, out) \mid a \in V - T\}.$$

The non-extended system Π' obtained in this way is equivalent to Π: when a string is sent out, a copy of it, marked in its left end with c'', is sent to one of membranes 2 or 5; in membrane 2 we introduce the trap symbol # by the rule $c'' \to (\#, out)$, and in membrane 5 we check whether the string is terminal – the computation stops only if this is the case. □

The replicated rewriting has a nice application: up to now, we have worked with sets of strings, without considering copies – hence multiplicities – of strings, and thus the result of a computation was a language. If we want to compute numbers, we need to make several changes to our systems: to work with multisets of strings, to consider halting computations (in order to know when a number is completed, is a "final" one), to "read" the result in an output membrane, and, moreover, to use operations which can increase the number of strings and the number of their copies. (It is not a problem to decrease the number of strings from a given membrane or from a given system: we can either "store" strings in some membranes, or send them outside the system, "losing" them in the environment.) Rewriting (as well as operations which will be considered in the subsequent sections, such as splicing, insertion, deletion, context adjoining, or context erasing) is not sufficient, because it just changes one string for another string. However, replication helps, since it can increase the number of strings.

Using it, we can consider membrane systems of the form $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m, i_o)$, where $M_1, \ldots, M_m$ are multisets of strings present in the m regions of μ in the initial configuration, and i_o is the output membrane, and we associate with them sets $N(\Pi)$ of the number of strings present in membrane i_o in the halting configurations.

Let us denote by $[E]NSP_m(rw, in, repl_n, halt)$ the family of sets $N(\Pi)$ computed by P systems of degree at most $m \geq 1$, using replicated rewriting rules with the replication index at most $n \geq 1$.

The proof of Theorem 5.2.12 can be modified in order to obtain a proof of the following result.

Theorem 5.2.16. $NSP_6(rw, in, repl_2, halt) = NRE$.

The idea is the following one. Consider a set $Q \in NRE$ and define the language $L_Q = \{a^n \mid n \in Q\}$. This is a recursively enumerable language. Take a matrix grammar $G = (N, \{a\}, S, M, F)$ generating the language L_Q and repeat the construction from the proof of Theorem 5.2.12. The obtained

system, Π, generates all strings of the form $a^n, n \in Q$, and sends them out of the system. Instead of sending them out, we want to keep them inside, and produce as many strings, of any given form, as there are many symbols in a string a^n. In this way, instead of the string a^n we will compute the number n, in the form of strings collected in the output membrane.

In the case of the system Π, we can proceed as follows. Instead of the rule $f \to (\lambda, out)||(f, in)$ from R_1 we introduce in this set the rules

$$f \to (f', in)||(f, in),$$
$$f' \to (f'', in),$$

for f', f'', two new symbols. We also add the rules

$$f' \to (\#, here),$$
$$f'' \to (\#, here)$$

to membrane 2, the rules

$$a \to (b, here)||(b, out),$$
$$f'' \to (\#, here)$$

to membrane 5, and

$$f' \to (\#, here)$$

to membrane 6. Thus, when a string of the form fz is obtained in the skin membrane, we produce two copies of it, $f'z$ and fz. As in the proof of Theorem 5.2.12, from fz we check whether or not z is terminal, and we stop only in the affirmative case. The string $f'z$ should be sent to membrane 5, otherwise the computation will continue forever in membranes 2 and 6 by means of the rules $f' \to (\#, here)$ and $\# \to (\#, here)$. In membrane 5 the only applicable rule is $a \to (b, here)||(b, out)$. The string $f'z_1bz_2$ is sent out to the skin region and will be rewritten by $f' \to (f'', in) \in R_1$, and has to go to membrane 6: in membrane 2 we introduce the trap symbol # immediately, and in membrane 5 we will keep forever a string of the form $f''x$, which will eventually be rewritten by $f'' \to (\#, here)$.

This means that starting from the string $f'a^n$ we can use the rule $a \to (b, here)||(b, out) \in R_5$ n times, once for each occurrence of a; at each application of this rule, a string is sent to membrane 1, and from here to membrane 6. The computation will stop only after exhausting the n occurrences of a and only if all strings $f''y$ are collected in membrane 6 (and, of course, when it stops correctly with respect to Π; in particular, the string fz should indeed be of the form fa^n, otherwise the computation will continue forever in membrane 2 – see again the proof of Theorem 5.2.12). In this way, in membrane 6 we end up with n strings, hence Theorem 5.2.16 is obtained. (Problem Q13, stated after Theorem 5.2.12, can now be supplemented with the question whether or not Theorem 5.2.16 can be improved concerning the number of membranes used; perhaps a direct proof, not derived from the proof of Theorem 5.2.12, can help.)

5.2.5 Parallel Rewriting

Rewriting the string–objects in a parallel manner is a mathematically natural counterpart of the sequential rewriting. There are many ways of defining parallel rewriting: (a) total parallelism (as in Lindenmayer systems; that is, rewriting all symbols of the string), (b) maximal parallelism (rewriting all symbols which can be rewritten, but not necessarily the whole string), (c) one-letter parallelism (choose a symbol and rewrite all its occurrences, possibly by different rules), (d) one-rule parallelism (choose a rule and apply it in all places where it can be applied), (e) rewrite one occurrence of each of the symbols which appear in a string, etc.

In the case of membrane systems there is a difficulty in choosing the right definition: how can we define the communication, where a string obtained by parallel rewriting should be placed in the next configuration of a computation? Again there are many possibilities: (1) to rewrite a string only by rules $a \to (u, tar)$ with the same target, (2) to consider systems which have several sets of rules associated with each membrane, each set being associated with a target (this is a generalization of the previous case, where we have only three possible sets of rules in each membrane, one for each of the targets *here, out, in*), (3) to use several rules with several targets, and to send the string to that destination which is indicated more frequently by the rules used (in the case when two targets appear the same number of times, then the string is sent to one of them, non-deterministically chosen), (4) as in case (3), but when two or three targets appear the same number of times the string is replicated and sent to all requested destinations, (5) using the conditional communication, as in Subsection 5.2.3.

Of particular interest seem to be cases (1) and (2), because the communication is defined in a simple and natural manner. However, the study of these cases is still a *research topic* (Problem Q15), and nothing has been done in this respect. On the other hand, case (3) looks rather "democratic" (the target is defined as the result of a "voting", by using the "majority rule"), although it is not clear if it resembles any biological phenomenon. What is interesting for case (3) is that it provides powerful possibilities of "programming" the functioning of a P system, at least in conjunction with mode (e) of parallelism. The system from the next theorem convincingly illustrates this assertion. Case (4) is also interesting, because it provides a way to replicate strings under the control of communication and not directly under the control of the rewriting rules.

We denote by $[E]LSP_m(rw, in, ppar)$ the family of languages generated by [extended] rewriting P systems of degree at most $m \geq 1$, using the target indications *here, out, in*, and at each step rewriting from each string one occurrence, non-deterministically chosen, of each symbol which appears in the string (hence, mode (e)), and sending the resulting string to the target which appears the maximal number of times in the used rules; if two targets appear the same number of times and this is the maximal number, then the

string is sent to one of these targets, non-deterministically chosen (hence, mode (3)); "ppar" comes from "partial parallelism".

Theorem 5.2.17. $ELSP_4(rw, in, ppar) = RE$.

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar in the strong binary normal form, given in the standard notation, and construct the partially parallel rewriting P system (of degree 4):

$$\Pi = (V, T, \mu, M_1, M_2, M_3, M_4, R_1, R_2, R_3, R_4),$$

with the following components:

$$\begin{aligned}
V &= N_1 \cup N_2 \cup T \cup \{X_i \mid X \in N_1, 0 \le i \le n\} \\
&\quad \cup \{A_i \mid A \in N_2, 0 \le i \le n\} \cup \{H, C_1, C_2, \#\}, \\
\mu &= [_1[_2\ ]_2[_3\ ]_3[_4\ ]_4]_1, \\
M_1 &= \{X_{init} A_{init} H\}, \\
M_2 &= M_3 = M_4 = \emptyset, \\
R_1 &= \{X \to (\alpha X_i, in), \\
&\qquad A \to (xA_i, in) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k\} \\
&\quad \cup \{X \to (YX_i, in) \mid m'_i : (X \to Y, B^{(j)} \to \#), k+1 \le i \le n\} \\
&\quad \cup \{X_i \to (\#, here) \mid X \in N_1, 0 \le i \le k\} \\
&\quad \cup \{A_i \to (\#, here) \mid A \in N_2, 0 \le i \le k\} \\
&\quad \cup \{C_1 \to (\lambda, in),\ C_2 \to (\lambda, in), \\
&\qquad H \to (H, in),\ H \to (\lambda, out)\} \\
&\quad \cup \{A \to (A, in) \mid A \in N_2\}, \\
R_2 &= \{X_i \to (X_{i-1}, out) \mid X \in N_1, 1 \le i \le k\} \\
&\quad \cup \{A_i \to (A_{i-1}, here) \mid A \in N_2, 1 \le i \le k\} \\
&\quad \cup \{X_0 \to (C_1, out) \mid X \in N_1\} \\
&\quad \cup \{A_0 \to (C_2, here) \mid A \in N_2\} \\
&\quad \cup \{C_1 \to (\#, here),\ C_2 \to (\#, here)\} \\
&\quad \cup \{X_i \to (\#, here) \mid X \in N_1, k+1 \le i \le n\}, \\
R_{2+j} &= \{X_i \to (\lambda, out), X \in N_1, i \in lab_j\} \\
&\quad \cup \{B^{(j)} \to (\#, here)\} \\
&\quad \cup \{X_i \to (\#, here) \mid X \in N_1, i \notin lab_j\} \\
&\quad \cup \{A_i \to (\#, here) \mid A \in N_2, 1 \le i \le k\}, \text{ for } j = 1, 2.
\end{aligned}$$

We start with the string $X_{init} A_{init} H$ in the skin membrane. Assume, in general, that we have a string XwH here. One copy of each non-terminal which can be rewritten must be rewritten. Assume that we rewrite X by a rule $X \to (\alpha Y_i, in)$, for some $1 \le i \le k$. If all $A \in N_2$ are rewritten by $A \to (A, in)$,

then the string $\alpha X_i wH$ is sent to a lower membrane (if no symbol from N_2 is present and we use the rule $H \to (\lambda, out)$, then the string can be sent out, but it is not terminal). From membrane 2 the string exits immediately by means of the rule $X_i \to (X_{i-1}, out)$ and then $X_{i-1} \to (\#, here)$ is applied in the skin membrane. In membranes 3, 4 one introduces the trap symbol by the rules $X_i \to (\#, here)$. Therefore, at least one symbol $A \in N_2$ should be rewritten by $A \to (xA_j, in)$. The string is sent to a lower membrane; in membranes 3, 4 the trap symbol is immediately produced. Assume that the string arrives in membrane 2. If it contains two or more than two symbols of the form A_j, B_l, for $A, B \in N_2$, then it remains in membrane 2 forever: we have two commands *here*, for A_j, B_l, and only one *out*, for X_i; when one of the subscripts becomes 0, the corresponding symbol is replaced by one of C_1, C_2, which after that introduces the trap symbol. Therefore, we must have only one symbol A_j in the string, and the string in membrane 2. We decrease the subscripts of X_i, A_j, simultaneously; this means that one introduces one command *out* (by the rule $X_i \to (X_{i-1}, out)$), and one *here* (by the rule $A_j \to (A_{j-1}, here)$). If the string is sent out, then the rules $X_{i-1} \to (\#, here)$, $A_{j-1} \to (\#, here)$ from R_1 must be used. Thus, we have to remain in membrane 2 and decrease the subscripts of X and A, repeatedly, until one of them reaches 0. If only X gets the subscript 0, then we obtain a string $\alpha X_0 w_1 A_l w_2 H$, and after that $\alpha C_1 w_1 A_{l-1} w_2 H$. If any of these strings is sent to region 1, then the trap symbol is introduced at least for A_l or A_{l-1}. If $\alpha C_1 w_1 A_{l-1} w_2 H$ remains in membrane 2, then $\#$ is produced by $C_1 \to (\#, here)$. If only A gets the subscript 0, then the result is the same: either $C_2 \to (\#, here) \in R_2$, or $X_l \to (\#, here) \in R_1$ must be used. If $i = j$, hence we get X_0, A_0 at the same time, then in membrane 2 we use the rules $X_0 \to (C_1, out), A_0 \to (C_2, out)$, and the string can be sent to the skin membrane (if it remains in membrane 2, then the rules $C_1 \to (\#, here)$ and $C_2 \to (\#, here)$ must be used). In region 1, the rules $C_1 \to (\lambda, in)$ and $C_2 \to (\lambda, in)$ are used, at the same time with rules $X \to (\alpha Z_j, in)$ or $X \to (ZX_l, in)$, and the computation continues. In this way, the matrix $m_i : (X \to Y,\ A \to x)$ has been correctly simulated.

If we start by using a rule $X \to (YX_i, in)$, for some $k + 1 \leq i \leq n$, then the string should be sent to membrane $2 + j$ such that $i \in lab_j$, otherwise the trap symbol is immediately introduced. Also in membrane $2 + j$ with $i \in lab_j$ we must have present no symbol $A_s, 1 \leq s \leq k$, because of the rules $A_s \to (\#, here) \in R_{2+j}$. If the string contains the symbol $B^{(j)}$, then the trap symbol is introduced, otherwise the only applicable rule is $X_i \to (\lambda, out)$, and the string returns to the skin membrane. Thus, the matrix $m_i : (X \to Y,\ B^{(j)} \to \#)$ was correctly simulated.

After obtaining a terminal string, we can use the rule $H \to (\lambda, out)$ and the string is sent out. (If we use the rule $H \to (H, in)$, then the string gets stuck in an inner membrane.) Thus $L(G) = L(\Pi)$. □

We end this section with the remark that the family $LSP_1(rw, ppar)$ contains languages which are not context-free. Indeed, consider the system

$$\Pi = (\{a, a', b, b', c, c'\}, \{a, a', b, b', c, c'\}, [_1\ ]_1, \{a'b'c'\}, R_1),$$

with the following rules:

$$R_1 = \{a' \to (aa', tar),\ b' \to (bb', tar),\ c' \to (cc', tar) \mid tar \in \{here, out\}\}.$$

At each step we have to use one rule for each symbol a', b', c'. If at least two of these rules are associated with the target *here*, then we have to continue. If at least two targets are equal to *out*, then the string has to be sent out of the system. Because at each step we increase the number of all symbols a, b, c by one, we obtain $L(\Pi) = \{a^n a' b^n b' c^n c' \mid n \geq 1\}$, which is not a context-free language.

5.3 Splicing Membrane Systems

The string–objects from the regions of a membrane system can evolve also by rules based on other operations than rewriting, and a natural candidate is splicing, the DNA-inspired operation defined in Subsection 2.3.10. In this way, a sort of uniformly biological model is obtained: both the membrane structure and the operations used inside its regions have origins in biology.

For the reader's convenience, let us first recall the definition of the splicing operation. Given an alphabet V and the symbols $\natural$ and \$ not in V, a splicing rule over V is a string of the form $r = u_1 \natural u_2 \$ u_3 \natural u_4$, where $u_1, u_2, u_3, u_4 \in V^*$. For such a splicing rule r and strings $x_1u_1u_2x_2, y_1u_3u_4y_2, x_1u_1u_4y_2, y_1u_3u_2x_2$ in V^* we write

$$(x_1u_1|u_2x_2, y_1u_3|u_4y_2) \vdash_r (x_1u_1u_4y_2, y_1u_3u_2x_2).$$

(We have indicated the sites of cutting the strings by vertical bars.)

Based on this operation, we define an extended *splicing P system* of *degree* $m \geq 1$ in the same way as a rewriting P system, that is as a construct

$$\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m),$$

where V is an alphabet, $T \subseteq V$ is the terminal alphabet, μ is a membrane structure consisting of m membranes labelled in a one-to-one manner with $1, \ldots, m$, $M_i \subseteq V^*, 1 \leq i \leq m$, are languages associated with the regions $1, \ldots, m$ of μ, while $R_i, 1 \leq i \leq m$, are finite sets of *evolution rules* associated with the regions $1, \ldots, m$ of μ, of the form $(r; tar_1, tar_2)$, where $r = u_1 \natural u_2 \$ u_3 \natural u_4$ is a splicing rule over V, the symbols $\natural$, \$ are not in V, and $tar_1, tar_2 \in \{here, out, in\}$.

The rules from a set R_i are used for splicing the string–objects from the region i. Initially we have in region i the strings from the set M_i.

There is a problem here: assume that in a region we have three strings, w_1, w_2, w_3, such that we can splice both w_1 with w_2, and w_1 with w_3. Are

we allowed to perform both these splicings or only one, non-deterministically chosen? Then, which strings are available at the next step? (This problem does not appear in the case of rewriting, because we rewrite single strings, the strings do not interact.) The way we choose to solve this problem is to assume that if a string is present, then it is present in arbitrarily many copies – as many as we need. This is a typical situation for the DNA area, where DNA molecules can be amplified by PCR in as many copies as we need. Thus, a string is not consumed by splicing, and at the next step we again have sufficient copies of it.

By splicing two strings x, y by a rule $(r; tar_1, tar_2)$ we get two new strings, w, z, which are sent to the regions indicated by tar_1, tar_2, respectively, in the know manner: *here* indicates that the string remains in the same region, *out* requires the string to exit the region, *in* says that the string has to enter any of the directly lower membranes, non-deterministically chosen; if there is no membrane inside the membrane where we want to apply the rule, then the rule cannot be applied. A string which is sent to another region than that where the rule is applied will leave the region, no copy of it remains behind, but from the next step on arbitrarily many copies will be available in the target region.

There is a subtle point here: if we already have a string w in a region i and we apply a splicing rule r to two strings x, y and we get the strings w, z such that w should exit the region, then the "old" w remains in region i, and only the newly generated string w has to leave. That is, we do not identify strings produced at different steps of the computation. (In the proofs below this situation will never appear.)

As usual, a computation is a sequence of transitions between configurations of a system Π starting from the initial configuration $(M_1, \ldots, M_m)$. The result of a computation is the language $L(\Pi)$ of all strings in T^* which are sent out of the system during the computation. (As we proceeded also in the rewriting case, we do not consider halting computations here.)

In order to have some control over the complexity of the splicing rules, we consider the following parameter: a splicing membrane system $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m)$ is said to have the *diameter* $dia(\Pi) = (n_1, n_2, n_3, n_4)$ if

$$n_i = \max\{|u_i| \mid (u_1\natural u_2\$u_3\natural u_4; tar_1, tar_2) \in R_j, 1 \leq j \leq m, \\ tar_1, tar_2 \in \{here, in, out\}\},\ 1 \leq i \leq 4.$$

We denote by $[E]LSP_m(spl, in, (n_1, n_2, n_3, n_4))$ the family of languages $L(\Pi)$ generated by [extended] splicing P systems as above of degree at most $m \geq 1$, and the diameter componentwise smaller than (n_1, n_2, n_3, n_4).

Theorem 5.3.1. $ELSP_2(spl, in, (1, 2, 2, 1)) = RE$.

Proof. Let $G = (N, T, S, R)$ be a type-0 Chomsky grammar in the Kuroda normal form. Consider a new symbol, B, and assume the symbols from $N \cup$

$T \cup \{B\}$ are numbered in any order, hence written as $\alpha_1, \ldots, \alpha_n$. Assume that R contains m productions: $u_i \to v_i, 1 \leq i \leq m$. Denote by R_{CF} the set of context-free rules from R and by R_{CS} the set of non-context-free rules from R. Consider also the set $R_0 = \{\alpha_i \to \alpha_i \mid 1 \leq i \leq n\}$.

We construct the following splicing P system:

$$
\begin{aligned}
\Pi &= (V, T, \mu, M_1, M_2, R_1, R_2),\\
V &= N \cup T \cup \{b, B, X, X_1, X_2, Y, Y_1, Y_2, Z_X, Z'_{X_1}, Z'_{X_2},\\
&\quad Z_Y, Z'_{Y_1}, Z'_{Y_2}, Z_\lambda, Z'_\lambda\}\\
&\quad \cup \{Z_{X_i}, Z_{Y_i} \mid 1 \leq i \leq n+m\} \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_{CS}\},\\
\mu &= [_1[_2\ ]_2]_1,\\
M_1 &= \{XBSY,\ X_2Z'_{X_2},\ Z'_{Y_1}Y_1,\ XZ_X, Z_\lambda,\ Z'_\lambda\}\\
&\quad \cup \{Z_{Y_i}b^iY_1 \mid 1 \leq i \leq n+m\}\\
&\quad \cup \{Z_{Y_i}Y'_i \mid u_i \to v_i \in R_{CS}\},\\
M_2 &= \{Z'_{Y_2}Y_2,\ X_1Z'_{X_1},\ Z_YY\} \cup \{X_1b^iv_iZ_{X_i} \mid 1 \leq i \leq n+m\},\\
R_1 &= \{r_1 : (\lambda\natural u_iY\$Z_{Y_i}\natural\lambda; in, out) \mid 1 \leq i \leq n+m\}\\
&\quad \cup \{r_2 : (\lambda\natural CY\$Z_{Y'_i}\natural\lambda; here, out),\\
&\qquad r_3 : (\lambda\natural AY'_i\$Z_{Y_i}\natural\lambda; in, out) \mid u_i \to v_i \in R_{CS}, u_i = AC\}\\
&\quad \cup \{r_4 : (\lambda\natural Z'_{X_2}\$X_1\natural b; here, out),\\
&\qquad r_5 : (\lambda\natural bY_2\$Z'_{Y_1}\natural\lambda; in, out)\}\\
&\quad \cup \{r_6 : (\lambda\natural Z_X\$X_1\natural\alpha; in, out) \mid \alpha \in N \cup T \cup \{B\}\}\\
&\quad \cup \{r_7 : (\lambda\natural BY\$Z_\lambda\natural\lambda; here, out),\\
&\qquad r_8 : (\lambda\natural Z'_\lambda\$X\natural\lambda; out, out)\},\\
R_2 &= \{r_9 : (\lambda\natural Y_1\$Z'_{Y_2}\natural\lambda; here, out),\\
&\qquad r_{10} : (\lambda\natural Z_{X_i}\$X\natural\lambda; out, out),\\
&\qquad r_{11} : (\lambda\natural Z'_{X_1}\$X_2b\natural\lambda; out, out)\}\\
&\quad \cup \{r_{12} : (\alpha\natural Y_2\$Z_Y\natural\lambda; out, out) \mid \alpha \in N \cup T \cup \{B\}\}.
\end{aligned}
$$

The idea of the proof is based on the "rotate and simulate" technique, well known in the H systems area. The sentential forms generated by G are simulated in Π in a circular permutation Xw_1Bw_2Y, $w_1, w_2 \in \{N \cup T\}^*$, with subscripted and/or primed variants of X and Y. Such a string will be present in a membrane of Π if and only if w_2w_1 is a sentential form of G. The non-terminal symbol Y can be removed only together with B, from strings of the form $XwBY$, $w \in (N \cup T)^*$. In this way the correct permutation of the string sent out of the system is ensured.

The simulation of a production in R and the circular permutation (we also say "rotation") of strings are done in the same way (as we will see below, rotation amounts to simulating rules from R_0).

Assume that in membrane 1 we have a string of the form Xwu_iY with $w, u_i \in (N \cup T \cup \{B\})^*$ (initially we have here the string $XBSY$).

The simulation of a rule from $R_{CF} \cup R_0$ starts by performing the splicing

$$(Xw|u_iY, Z_{Y_i}|b^iY_1) \vdash_{r_1} (Xwb^iY_1, Z_{Y_i}u_iY)$$

in the skin region; the first output string is sent into membrane 2 while the second string is sent out of the system.

The simulation of a rule $r_i : AC \to v_i$ from R_{CS} starts with the splicing

$$(XwA|CY, Z_{Y_i'}|Y_i') \vdash_{r_2} (XwAY_i', Z_{Y_i'}CY);$$

the first output string remains in membrane 1 and the second string leaves the system. Then, we perform

$$(Xw|AY_i', Z_{Y_i}|b^iY_1) \vdash_{r_3} (Xwb^iY_1, Z_{Y_i}AY_i');$$

the first output string is sent to membrane 2 and the second string leaves the system.

In both cases the suffix u_iY is changed to $b^iY_1, 1 \leq i \leq n+m$. The strings leaving the system do not belong to T^* so they do not contribute to the language generated by Π.

In membrane 2, a string of the form Xwb^iY_1 can enter a splicing

$$(Xwb^i|Y_1, Z_{Y_2}'|Y_2) \vdash_{r_9} (Xwb^iY_2, Z_{Y_2}'Y_1).$$

The second output string is sent to membrane 1 where no splicing rule can be applied to it; the string Xwb^iY_2 remains in membrane 2 and can enter the splicing operation

$$(X_1b^jv_j|Z_{X_j}, X|wb^iY_2) \vdash_{r_{10}} (X_1b^jv_jwb^iY_2, XZ_{X_j}), 1 \leq j \leq n+m.$$

Both output strings are sent to membrane 1 but only the first one can be involved in further splicing operations. A string of the form Xwb^iY_1 can also be spliced in membrane 2 by the rule r_{10} :

$$(X_1b^jv_j|Z_{X_j}, X|wb^iY_1) \vdash_{r_{10}} (X_1b^jv_jwb^iY_1, XZ_{X_j}), 1 \leq j \leq n+m.$$

Both output strings are sent to membrane 1. The second string cannot be involved in any splicing, the first string can enter a splicing

$$(X_2|Z_{X_2}, X_1|b^jv_jwb^iY_1) \vdash_{r_4} (X_2b^jv_jwb^iY_1, X_1Z_{X_2}),$$

but both strings, remaining in membrane 1, are no longer spliced.

We start now to check whether or not in the string $X_1b^jv_jwb^iY_1$ obtained as above we have $i = j$. This is done by repeatedly removing one occurrence of b from the vicinity of X_1 and one from the vicinity of Y_1. Only in the case when $i = j$ will we continue in a way which can lead to a terminal string,

namely, from $X_1 v_i w Y_2$. This simulates the use of the rule $r_i : u_i \to v_i$ from $R \cup R_0$: the string u_i was removed from the right-hand end of the string and v_i was added in the left-hand end of the string. When $v_i = u_i = \alpha_j$ for some $1 \leq j \leq m$, this corresponds to circularly permuting the string with one symbol.

Here are the details of the operation of checking whether or not $i = j$.

A string of the form $X_1 b^j v_j w b^i Y_2$ can be spliced in membrane 1 and so substitutes X_1 with X_2 and bY_2 with Y_1, by

$$(X_2|Z'_{X_2}, X_1|b^j v_j w b^i Y_2) \vdash_{r_4} (X_2 b^j v_j w b^i Y_2, X_1 Z'_{X_2})$$

(the first output string remains in membrane 1, the second string is sent out of the system) and

$$(X_2 b^j v_j w b^{i-1}|bY_2, Z_{Y_1}|Y_1) \vdash_{r_5} (X_2 b^j v_j w b^{i-1} Y_1, Z_{Y_1} bY_2)$$

(the first output string is sent to membrane 2, the second string leaves the system). In membrane 1 it is also possible to have

$$(X_1 b^j v_j w b^{i-1}|bY_2, Z_{Y_1}|Y_1) \vdash_{r_5} (X_1 b^j v_j w b^{i-1} Y_1, Z_{Y_1} bY_2).$$

The second resulting string is sent out of the system while the first string is sent to membrane 2, where we can perform

$$(X_1 b^j v_j w b^{i-1}|Y_1, Z'_{Y_2}|Y_2) \vdash_{r_9} (X_1 b^j v_j w b^{i-1} Y_2, Z'_{Y_2} Y_1).$$

The first output string remains in membrane 2, the second one is sent to membrane 1, and none of them will be involved in any further splicing operation.

In membrane 2 a string of the form $X_2 b^j v_j w b^{i-1} Y_1$ can be spliced and so substitutes Y_1 with Y_2 and $X_2 b$ with X_1, by

$$(X_2 b^j v_j w b^{i-1}|Y_1, Z'_{Y_2}|Y_2) \vdash_{r_9} (X_2 b^j v_j w b^{i-1} Y_2, Z'_{Y_2} Y_1)$$

(the first string remains in membrane 2, the second one is sent to membrane 1 and cannot be involved in any further splicing) and

$$(X_1|Z'_{X_1}, X_2 b|b^{j-1} v_j w b^{i-1} Y_2) \vdash_{r_{11}} (X_1 b^{j-1} v_j w b^{i-1} Y_2, X_2 b Z'_{X_1})$$

(both resulting strings are sent to membrane 1 but only the first one can be spliced). In membrane 2 it is also possible to have

$$(X_1|Z'_{X_1}, X_2 b|b^{j-1} v_j w b^{i-1} Y_1) \vdash_{r_{11}} (X_1 b^{j-1} v_j w b^{i-1} Y_1, X_2 b Z'_{X_1}).$$

Both strings are sent to membrane 1 but only the first one can be spliced with $X_2 Z_{X_2}$ by rule r_4 and we obtain $X_2 b^{j-1} v_j w b^{i-1} Y_1$, which remains in membrane 1 without entering any new splicing, and $X_1 Z_{X_2}$ which is sent out of the system (but it is not a terminal string).

The process of decreasing the number of occurrences of b from the left- and the right-hand ends of strings continues until either a string of one of the forms $X_1v_iwY_2, X_1b^kv_jwY_2$ is obtained in membrane 1, or a string of the form $X_2v_jwb^kY_1$ is obtained in membrane 2, $1 \leq k \leq n+m-1$.

As we have seen above, a string $X_1b^kv_jwY_2$ can be spliced with $X_2Z'_{X_2}$ by rule r_4, and we obtain the string $X_2^kv_jwY_2$, which remains in membrane 1 and it is no longer spliced, and $X_1Z'_{X_2} \notin T^*$ which is sent out of the system.

In membrane 2 a string $X_2v_jwb^kY_1$ can be spliced with $Z'_{Y_2}Y_2$, but the resulting strings will enter no further splicings.

Thus, the only continuation of interest is from a string of the form $X_1v_iwY_2$, for which we have

$$(X|Z_X, X_1|v_iwY_2) \vdash_{r_6} (Xv_iwY_2, X_1Z_X).$$

The first output string is sent to membrane 2 and the second one (not in T^*) is sent out of the system. In membrane 2 we perform

$$(Xv_iw|Y_2, Z_Y|Y) \vdash_{r_{12}} (Xv_jwY, Z_YY_2),$$

both output strings are sent to membrane 1, but only the first one can be involved in further splicing operations.

In this way, we have passed from Xwu_iY to Xv_iwY, simulating a production from R or rotating the substring bounded by X and Y with one symbol.

At any moment, a string of the form XwY can be spliced in membrane 1 by rules r_7 and r_8.

If we perform

$$(|Z'_\lambda, X|wY) \vdash_{r_8} (wY, XZ'_\lambda),$$

then the first output string is sent out of the system, but it is not terminal, while the second string remains in membrane 1 and cannot be involved in any further splicing.

If $w = xB, x \in (N \cup T)^*$, then we can perform

$$(Xx|BY, Z_\lambda|) \vdash_{r_7} (Xx, Z_\lambda BY).$$

The first output string can be spliced by

$$(|Z'_\lambda, X|x) \vdash_{r_8} (x, XZ'_\lambda),$$

and both strings x and XZ'_λ are sent out of the system. If $x \in T^*$, then $x \in L(\Pi)$, and the system Π has simulated a derivation of G.

Thus, all derivations in G can be simulated in Π and, conversely, all correct computations in Π correspond to correct derivations in G, hence $L(G) = L(\Pi)$. As $dia(\Pi) = (1, 2, 2, 1)$, the proof is complete. □

Clearly, if $r = u_1\natural u_2\$u_3\natural u_4$ and $r' = u_3\natural u_4\$u_1\natural u_2$, then $(x, y) \vdash_r (w, z)$ if and only if $(y, x) \vdash_{r'} (z, w)$, which proves the following auxiliary result.

Lemma 5.3.1. $[E]LSP_m(spl, in, (n_1, n_2, n_3, n_4)) = [E]LSP_m(spl, in, (n_3, n_4, n_1, n_2))$, *for all* $m \geq 1$ *and all* $n_i \geq 0, 1 \leq i \leq 4$.

From Theorem 5.3.1 and Lemma 5.3.1 we now get the following result.

Corollary 5.3.1. $ELSP_2(spl, in, (2, 1, 1, 2)) = RE$.

A normal form result like that in Theorem 5.1.7 holds also for splicing P systems: it is possible to have a unique set of rules, valid for all regions.

Theorem 5.3.2. *Each language* $L \in RE$ *can be generated by an extended splicing P system of degree 2 with global rules.*

Proof. Let $G = (N, T, S, R)$ be a type-0 Chomsky grammar (not necessarily in the Kuroda normal form, as we are not interested in the diameter of the system we are going to construct) and let B be a new symbol. Assume that $N \cup T \cup \{B\} = \{\alpha_1, \ldots, \alpha_n\}$ and that R contains m rules, $u_i \to v_i, 1 \leq i \leq m$. Consider also the rules $u_{m+j} \to v_{m+j}, 1 \leq j \leq n$, for $u_{m+j} = v_{m+j} = \alpha_j$.

We construct the splicing P system (of degree 2):

$$
\begin{aligned}
\Pi &= (V, V, \mu, M_1, M_2, R' = R'_1 \cup R'_2),\\
V &= N \cup T \cup \{B, X, X', Y, Y', Z_1, Z_2, Z'_1\} \cup \{X_i \mid 0 \leq i \leq n+m\}\\
&\quad \cup \{Y_i \mid 0 \leq i \leq n+m\},\\
\mu &= [_1[_2\ ]_2]_1,\\
M_1 &= \{Z'_1,\ X'Z_1,\ Z_1Y'\} \cup \{Z_1Y_i \mid 0 \leq i \leq n+m\}\\
&\quad \cup \{X_iv_iZ_1 \mid 1 \leq i \leq n+m\},\\
M_2 &= \{XSBY,\ XZ_2,\ Z_2Y\} \cup \{Z_2Y_i \mid 1 \leq i \leq m+n\}\\
&\quad \cup \{X_iZ_2 \mid 0 \leq i \leq m+n-1\},\\
R'_1 &= \{(X_iv_i\natural Z_1\$X\natural\lambda;\ in,\ out),\\
&\qquad (\alpha\natural Y_i\$Z_1\natural Y_{i-1};\ in,\ out) \mid 1 \leq i \leq m+n, \alpha \in N \cup T \cup \{B\}\}\\
&\quad \cup \{(\lambda\natural Y_0\$Z_1\natural Y';\ in,\ out)\}\\
&\quad \cup \{(X_0\natural\$X'\natural Z_1;\ out,\ here) \mid \alpha \in N \cup T \cup \{B\}\}\\
&\quad \cup \{(\lambda\natural BY\$Z'_1\natural\lambda;\ here,\ out),\\
&\qquad (X\natural\$\natural Z'_1;\ out,\ out)\},\\
R'_2 &= \{(\alpha\natural u_iY\$Z_2\natural Y_i; out, here),\\
&\qquad (X_i\natural\alpha\$X_{i-1}\natural Z_2; here, out) \mid 1 \leq i \leq m+n, \alpha \in N \cup T \cup \{B\}\}\\
&\quad \cup \{(X'\natural\$X\natural Z_2;\ here,\ here),\\
&\qquad (\lambda\natural Y'\$Z_2\natural Y;\ out,\ here)\}.
\end{aligned}
$$

The construction is based on the same rotate and simulate idea as the previous proof. We leave it to the reader to examine the way this system works. To this end, the following observations can be useful. The rules from

R_1' can be used only in membrane 1, and the rules from R_2' can be used only in membrane 2. This is ensured both by the target indications (a rule using the target *in* cannot be used in membrane 2), and by the special symbols Z_1, Z_1', Z_2 (Z_1, Z_1' appear only in rules from R_1' and never in strings from membrane 2, while Z_2 appears only in rules from R_2'; although strings containing Z_2 are sent from membrane 2 to membrane 1, they cannot enter any splicing here). The only way to start a computation which can lead to terminal strings which exit the system is to use the rule $(X_i v_i \natural Z\$X\natural\lambda;\ in,\ out)$ in membrane 1 and to continue with $(\alpha\natural u_j Y\$Z_2\natural Y_j;\ out,\ here)$ in membrane 2, with $i = j$ (otherwise we will reach a configuration where no string from the system can enter any splicing). In this way, one removes u_i from the end of the string and one appends v_i in the left-hand end. This both simulates the rules from R and circularly permutes the string with one symbol, in the case when $m + 1 \leq i \leq m + n$. The equality $L(G) = L(\Pi)$ is obtained. □

It is not known whether or not the results from Theorems 5.3.1 and 5.3.2 hold also in the non-extended case (Problem Q16).

Making use of two further membranes, one can avoid the terminal alphabet in the following way. Consider a system Π, such as those from the proofs of Theorems 5.3.1 and 5.3.2, and add two membranes $[_{-1}[_0\ ]_0]_{-1}$ around its skin membrane. Send from Π to membrane 0 a string $XwBY$ (hence without removing the symbol B), and perform here one further rotation of the string, but only simulating rules $a \to a$ for $a \in T$, as well as $B \to B$. When B arrives again in the right-hand end, we erase it, together with Y, and send the string to membrane -1. Here, we also remove the left marker X (or variants of it) and send the string out. Thus, what is obtained outside is a string over T. However, the cost seems to be too large – the challenge of Problem Q16 is to make the terminal alphabet useless without using any additional membrane besides the two used in the proofs of Theorems 5.3.1 and 5.3.2.

In Subsection 2.3.10 we have mentioned that, in some sense, splicing is "very close" to computational universality; by adding rather weak (at first sight) controls on the use of rules, we get universality. The previous results are not very good illustrations of this assertion, because two membranes are known to be universal in many cases – see again the results from the previous chapters. However, if we add something more to a splicing P system, then the universality is obtained for systems with only one membrane.

Such a powerful extension is to introduce the possibility of *rule creation*, as also considered for rewriting P systems. Specifically, the rules will be of the form $r : (u_1\natural u_2\$u_3\natural u_4; tar_1, tar_2)/z$, where z is a string of labels of possible rules. When a rule of this form is applied, it is consumed, and the rules indicated by z are available in the next step of the computation.

Formally, a splicing P system with rule creation is, similar to a rewriting P system with rule creation, a construct $\Pi = (V, T, \mu, M_1, \ldots, M_m, lab, R_{pos}, R_1, \ldots, R_m)$ (alphabet, terminal alphabet, membrane structure, sets of string–objects placed in the m regions of μ, labels of possible rules, the forms of

possible rules, multisets of splicing rules present in the m regions of μ, respectively). We denote by $[E]LSP_m(spl, in, rule)$ the family of languages generated by [extended] splicing P systems of degree at most $m \geq 1$ with rule creation.

Theorem 5.3.3. $ELSP_1(spl, rule) = LSP_2(spl, in, rule) = RE$.

Proof. Consider a type-0 Chomsky grammar $G = (N, T, S, R)$. As in the previous proofs, add to R all rules of the form $\alpha \to \alpha$, for $\alpha \in N \cup T$, as well as the rule $B \to B$, for a new symbol B (the set of rules obtained is still denoted by R), and label all these rules with $p_1, \ldots, p_k$, such that $p_i : a \to a$ for $a \in T, 1 \leq i \leq h$, and $p_k : B \to B$.

We first construct the non-extended splicing P system

$$\Pi = (V, V, [_1[_2\ ]_2]_1, M_1, \emptyset, lab, R_{pos}, R_1, \emptyset),$$

with the following components:

$$\begin{aligned}
V &= N \cup T \cup \{B, X, Y, Y', \bar{Y}, \bar{Y}', Z\},\\
M_1 &= \{XBSY, ZY, Z\bar{Y}, ZY', Z\bar{Y}', Z\} \cup \{XvZ \mid u \to v \in R\},\\
lab &= \{r_i, r_i', r_i'', r_i''', \bar{r}_i, \bar{r}_i', \bar{r}_i'' \mid 1 \leq i \leq k\} \cup \{r_0, r_0', r_0''\},\\
R_1 &= r_0 r_0' r_1 r_2 \ldots r_k \bar{r}_1 \bar{r}_2 \ldots \bar{r}_h,\\
R_{pos} &= \{r_i : (\lambda\natural uY\$Z\natural Y'; here, in)/r_i',\\
&\qquad r_i' : (Xv\natural Z\$X\natural\lambda); here, in)/r_i'',\\
&\qquad r_i'' : (\lambda\natural Y'\$Z\natural Y; here, in)/r_i \mid \text{for } r_i : u \to v, 1 \leq i \leq k\}\\
&\cup \{r_0 : (\lambda\natural BY\$Z\natural\bar{Y}; here, in)/r_k'\}\\
&\cup \{\bar{r}_i : (\lambda\natural\alpha Y\$Z\natural\bar{Y}'; here, in)/\bar{r}_i',\\
&\qquad \bar{r}_i' : (X\alpha\natural Z\$X\natural\lambda; here, in)/\bar{r}_i'',\\
&\qquad \bar{r}_i'' : (\lambda\natural\bar{Y}'\$Z\natural\bar{Y}; here, in)/\bar{r}_i \mid \text{for } r_i : \alpha \to \alpha, 1 \leq i \leq h\}\\
&\cup \{r_k' : (XB\natural Z\$X\natural\lambda; here, in)/\bar{r}_k'',\\
&\qquad \bar{r}_k'' : (\lambda\natural\bar{Y}'\$Z\natural\bar{Y}; here, in)/\lambda,\\
&\qquad r_0' : (\lambda\natural B\bar{Y}\$Z\natural\lambda; here, in)/r_0'',\\
&\qquad r_0'' : (\lambda\natural Z\$X\natural\lambda; out, in)/\lambda\}.
\end{aligned}$$

The system Π simulates the terminal derivations in the grammar G by using the same rotate and simulate technique as used in the previous proofs, making essential use of the fact that we have only certain rules available. Initially, we have the rules from R_1. Each sequence r_i (available), r_i' (created by r_i), r_i'' (created by r_i') ensures the simulation of the rule $p_i : u \to v$ from R (this means both the simulation of rules from the initial grammar G and the circular permutation of the string).

Membrane 2 is used only as a garbage collector, where all by-product strings which we do not need are sent.

The only way to send a string outside the system is to use the rule r_0'', which is available only after using the rule r_0', which, in turn, can be used only if the string ends with $B\bar{Y}$. Such a suffix appears only after using the rule r_0, which introduces the symbol $\bar{Y}$ and removes the symbol B; the next rule to be used is $\bar{r}_k'$, which, together with r_k'', moves the symbol B in the left end of the string.

From now on, the triples of rules $\bar{r}_i, \bar{r}_i', \bar{r}_i'', 1 \leq i \leq h$, are used exactly as the triples $r_i, r_i', r_i'', 1 \leq i \leq k$, in the presence of the barred version of Y, and permute the string, *provided that it is terminal.* If any non-terminal symbol is met, then the computation stops without sending out any string. If the string is terminal, then it can be completely permuted, B arrives again in the right-hand end, and the rules r_0', r_0'' can be applied, the end markers are removed, and the string is sent out. We know that it is terminal, and we introduce it in the language $L(\Pi)$, which is thus equal to $L(G)$.

The use of membrane 2 can be avoided, by sending out all strings which we do not need in the system, and considering T as the terminal alphabet of the system. Because all these strings contain the non-terminal symbol Z, they do not count when constructing the language $L(\Pi)$. □

The extension of splicing P systems by considering permitting/forbidding conditions, in the form of strings or symbols associated with rules and which must be present in the strings which are spliced by given rules, leads again to immediate universality results (in the extended case), because the H systems with permitting/forbidding conditions (in the form of symbols) are known to be universal. Also, taking into account the multiplicity of string–objects leads to universality for one-membrane P systems, just because H systems working with multisets of strings characterize the recursively enumerable languages. The reader is referred to [195] for rigorous formulations and for proofs of these assertions.

Thus, it is much more interesting to consider some weaker variants of splicing membrane systems. One possibility is to associate no target indication to splicing rules and to proceed as in the case of rewriting P systems with conditional communication of type (*empty, empty*): after each splicing, the resulting strings have to immediately move, either to the upper region, or to any of the directly lower regions, non-deterministically choosing the destination.

Knowing the power of the splicing operation it might not be surprising that we again get universality, but what is completely new in this case is that no bound on the number of membranes is known. The existence of such a bound follows from the universality of our systems, but finding a reasonable value remains an *open problem* – Problem Q17.

We denote by $[E]LSP_m(spl, move)$ the family of languages generated by splicing P systems of degree at most $m \geq 1$, with *immediate communication*, that is with rules of the form $u_1\natural u_2\$u_3\natural u_4$, hence with no target indication; after each splicing operation, the resulting strings should move from the current

region (this is equivalent to saying that in each region, each rule $u_1\natural u_2\$u_3\natural u_4$ appears in all four variants $(u_1\natural u_2\$u_3\natural u_4; tar_1, tar_2)$, $tar_1, tar_2 \in \{in, out\}$). The subscript m is replaced by $*$ when no bound on the degree of the systems is considered.

Theorem 5.3.4. *$FIN = ELSP_1(spl, move) \subset REG \subset ELSP_2(spl, move) \subset ELSP_*(spl, move) = RE$, and $LSP_2(spl, move) - REG \neq \emptyset$.*

Proof. The inclusion $ELSP_1(spl, move) \subseteq FIN$ is obvious (only one splicing is possible for each string from the initial configuration, as the resulting strings have to exit the system).

For the opposite inclusion, consider the system

$$\begin{aligned}\Pi = (&V \cup \{e\}, V, [_1\ ]_1, \{e\} \cup \{ex_i \mid 1 \leq i \leq n\},\\ &\{\lambda\natural e\$e\natural x_i \mid 1 \leq i \leq n\}),\end{aligned}$$

for given $L = \{x_1, \ldots, x_n\} \subset V^*$. The equality $L(\Pi) = L$ is obvious.

Consider now a deterministic finite automaton $A = (Q, V, s_0, F, \delta)$ and construct the splicing P system

$$\begin{aligned}\Pi &= (Q \cup V \cup \{e, e', f\}, V, [_1[_2\ ]_2]_1, \{e', s_0\} \cup \{eas \mid a \in V, s \in Q\},\\ &\quad \{es \mid s \in Q\} \cup \{ef\}, R_1, R_2),\\ R_1 &= \{\lambda\natural s\$e\natural a\delta(s) \mid a \in V, s \in Q\} \cup \{\lambda\natural f\$e'\natural\lambda\},\\ R_2 &= \{\lambda\natural s\$e\natural s \mid s \in Q\} \cup \{\lambda\natural s\$e\natural f \mid s \in F\}.\end{aligned}$$

The splicing rules from region 1 prolong the string with one symbol, following a parsing in the automaton A; the resulting strings are sent to membrane 2 (or outside the system, but they are non-terminal with respect to Π), where nothing happens, the state present in the right-hand end of the strings is just reproduced; thus, the process can be iterated. When we reach a final state, in membrane 2 one introduces the symbol f, which can be removed in the skin membrane and the string should exit, because no further rule can be applied to it. If we start from s_0, then we obtain a string from $L(A)$; if we start from a string eas (or, later, from es), then the symbol e is never removed; because it is not a terminal symbol, the string obtained is not accepted in the generated language. Therefore, we have $L(A) = L(\Pi)$, which proves the inclusion $REG \subseteq ELSP_2(spl, move)$.

This inclusion is proper – actually, even $LSP_2(spl, move)$ contains non-regular languages. In order to see this, let us consider the (non-extended) splicing P system

$$\begin{aligned}\Pi &= (\{a, b, c, d, e\}, \{a, b, c, d, e\}, [_1[_2\ ]_2]_1, \{caabd, ebd\}, \{cae\}, R_1, R_2),\\ R_1 &= \{b\natural d\$e\natural bd\},\\ R_2 &= \{ca\natural e\$c\natural a\}.\end{aligned}$$

Each string ca^nb^md from the skin membrane is spliced and we get the strings $ca^nb^{m+1}d$ and ed, which can be sent out or into the inner membrane. In membrane 2, the string ed cannot be spliced, while in the string $ac^nb^{m+1}d$ we also increase the number of occurrences of a, and we return to the skin membrane the string $ca^{n+1}b^{m+1}d$ (and ce, which is never spliced). The process is iterated. Because we start from $caabd$, the terminal strings sent out are of the form $ca^nb^nd, n \geq 2$, or $ca^nb^{n+1}d$, or ed, hence their language is not regular.

The most significant relation is the inclusion $RE \subseteq ELSP_*(spl, move)$.

Consider a type-0 Chomsky grammar $G = (N, T, S, R)$. We add to R all rules of the form $\alpha \to \alpha$, for $\alpha \in N \cup T$, as well as the rule $B \to B$, for a new symbol B. We continue to denote by R the augmented set of rules and we assume that its rules are labelled in a one-to-one manner with $p_1, \ldots, p_k$.

We construct the splicing P system of degree $2k+3$, with the membranes labelled with $0, 1, 1', \ldots, k, k', f, f'$, where 0 is the skin membrane:

$$
\begin{aligned}
\Pi &= (V, T, \mu, M_0, M_1, M_{1'}, \ldots, M_k, M_{k'}, M_f, M_{f'}, \\
&\qquad R_0, R_1, R_{1'}, \ldots, R_k, R_{k'}, R_f, R_{f'}), \\
V &= N \cup T \cup \{B, X, Y, Y', Y'', Y_f, Z\} \cup \{Y_i \mid 1 \leq i \leq k\}, \\
\mu &= [_0[_1[_{1'}\ ]_{1'}]_1 \cdots [_k[_{k'}\ ]_{k'}]_k[_f[_{f'}\ ]_{f'}]_f]_0, \\
M_0 &= \{XBSY, Z, ZY_f\} \cup \{ZY_i \mid 1 \leq i \leq k\}, \\
M_i &= \{ZY\} \cup \{XvZ \mid u \to v \in R\}, \\
M_{i'} &= \{ZY'\}, \text{ for all } 1 \leq i \leq k, \\
M_f &= \{Z, ZY''\}, \\
M_{f'} &= \{ZY'\}, \\
R_0 &= \{\lambda\natural uY\$Z\natural Y_i \mid \text{ for } r_i : u \to v, 1 \leq i \leq k\} \\
&\quad \cup \{\lambda\natural BY\$Z\natural Y_f,\ \lambda\natural Y''\$Z\natural\lambda\}, \\
R_i &= \{Xv\natural Z\$X\natural\lambda \mid r_i : u \to v\} \\
&\quad \cup \{\lambda\natural Y'\$Z\natural Y\}, \\
R_{i'} &= \{\lambda\natural Y_i\$Z\natural Y'\}, \text{ for all } 1 \leq i \leq k, \\
R_f &= \{\lambda\natural Z\$X\natural\lambda,\ \lambda\natural Y'\$Z\natural Y''\}, \\
R_{f'} &= \{\lambda\natural Y_f\$Z\natural Y'\}.
\end{aligned}
$$

The idea underlying this construction is again the rotate and simulate technique. The strings of interest are those bounded by the markers X, Y and by their variants, and such strings are always the first of the two strings obtained by splicing. The second strings always contain the non-terminal Z, and they will never lead to terminal strings. Thus, we ignore these "by-product" strings, and consider below only the first result of each splicing operation.

Assume that we have a string XwY in the skin membrane and we use a rule $\lambda\natural uY\$Z\natural Y_i$, corresponding to a rule $r_i : u \to v$ from R, $1 \leq i \leq k$. The

obtained string, $Xw'Y_i$, for $w = w'u$, contains non-terminal symbols, hence in order to reach a terminal string we have to send it into an inner membrane. Assume that we send it into a membrane j with $j \neq i$. After using the local rule $Xv'\natural Z\$X\natural\lambda$, for $r_j = u' \to v'$, we send the string $Xv'w'Y_i$ either to the skin membrane, or to the lower membrane j'. No rule can be applied in these regions to this string, hence it remains forever unchanged, without leading to a string in $L(\Pi)$. The same result is obtained if the string $Xw'Y_i$ is sent to membrane f (we can remove the left marker X, but nothing more can be done).

Thus, the string $Xw'Y_i$ should be sent to membrane i. Here we splice it by using the rule $Xv\natural Z\$X\natural\lambda$, corresponding to the rule r_i of R, and we get the string $Xvw'Y_i$. If this string is sent out of region i, then no splicing can be applied, and the string is "lost". If the string is sent to the inner membrane i', then we can use the rule $\lambda\natural Y_i\$Z\natural Y'$ here, the string $Xvw'Y'$ is sent to region i, where we use the rule $\lambda\natural Y'\$Z\natural Y$, and the string $Xvw'Y$ is obtained. If we send it to the inner membrane i', then the string gets stuck, as no splicing can be applied to it; if we send the string to the skin membrane, then the process can be iterated.

In this way, we have passed from $Xw'uY$ to $Xvw'Y$, for $r_i : u \to v$. If this was an initial rule from G, then we have simulated a derivation step in the grammar (modulo a circular permutation of the string). If we had $r_i : \alpha \to \alpha$, for some $\alpha \in N \cup T \cup \{B\}$, then the string has been circularly permuted with one symbol. Thus, we can simulate all derivations with respect to G.

If in the skin membrane we use the rule $\lambda\natural BY\$Z\natural Y_f$, then this means that the string was of the form $XwBY$, hence in the same permutation as in the grammar G. We obtain the string XwY_f. If we send it to any membrane $i, 1 \leq i \leq k$, then no output will be obtained, because the string gets stuck either in membrane i', or in the skin membrane. If we send it to membrane f, then we obtain here the string wY_f, which should be sent to the inner membrane f' (if we send it to the skin membrane, then no splicing can be applied to it). In membrane f' we replace Y_f by Y' and the string wY' is sent to region f. Here we replace Y' by Y''. If the string wY'' is sent to membrane f' again, then it is "lost"; if the string is sent to the skin membrane, then we can use here the rule $\lambda\natural Y''\$Z\natural\lambda$, which produces the string w. If this string is sent to any lower membrane, or if it is sent outside of the system, but it is not a terminal string, then it is "lost". If the string w is sent out and it is from T^*, then it is accepted in the language $L(\Pi)$.

Consequently, $L(\Pi) = L(G)$, and this concludes the proof. □

5.4 Contextual Membrane Systems

We recall from Subsection 2.3.10 the fact that in Marcus contextual grammar one uses rules of the form $(C, (u, v))$, where C is a language and (u, v) is a

pair of strings; C is called the *selector* and (u, v) is called the *context* of the *rule* $(C, (u, v))$. For two strings $w, z \in V^*$ we write $w \Longrightarrow z$ if $w = w_1 x w_2, z = w_1 u x v w_2$, for some contextual rule $(C, (u, v))$, such that $x \in C$ (the context (u, v) is adjoined to the string x from the selector C associated with the context). It is also possible to use a contextual rule $(C, (u, v))$ in the erasing mode: $w_1 u x v w_2 \Longrightarrow w_1 x w_2$ if $x \in C$ (the context (u, v) is erased if it appears around a string from the selector). In general, when one of the strings u, v is empty, the rule (irrespective of whether it is adjoining or erasing) is called *one-sided.*

Clearly, such rules can be used in order to evolve the string–objects of a membrane system. We briefly explore this possibility here.

Formally, a contextual P system (of degree $m \geq 1$) is a construct $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m)$, with the components exactly as in the rewriting or splicing P system, but with $R_1, \ldots, R_m$ containing adjoining and erasing contextual rules, given in the following forms: $(C, (u, v); tar)_a$ (adjoining rules), $(C, (u, v); tar)_e$ (erasing rules); in both cases, $(C, (u, v))$ is a contextual rule and $tar \in \{here, out, in\}$ is a target indication. The rules are applied in the usual (sequential) maximally parallel manner: each string is processed by only one rule, but all strings which can evolve must do it; the strings resulting from the use of rules, by adjoining the context in the case of adjoining rules, and erasing it in the case of erasing rules, will go to the region indicated by *tar*. The language $L(\Pi)$ consists of all strings over T which are sent out of the system during the computation (the computations do not need to be halting).

A contextual P system is said to be *of weight* $(n_1, n_2; n_3, n_4)$ if the context (u, v) of any adjoining rule has $|u| \leq n_1, |v| \leq n_2$, and the context (u, v) of any erasing rule has $|u| \leq n_3, |v| \leq n_4$. Note that by the weight of a system we can control both the size of contexts and the fact that only one-sided contexts of a given type, adjoining or erasing, are used.

Also, we say that a system Π *has selection of type* X, where X is a family of languages, if all its selectors belong to X. Thus, we can have finite selection, regular selection, etc. We consider here at most regular selectors, because it is natural to work with selectors of a type as simple as possible. It is also of interest to have the minimal type of selection, that is singleton selection, when the selectors consist of a single string. Clearly, instead of a rule $(\{x_1, \ldots, x_n\}, (u, v); tar)_\alpha$, with finite selection, we can consider the n rules with singleton selection $(\{x_i\}, (u, v); tar)_\alpha, 1 \leq i \leq n$, and the generated language is the same. Hence, it is enough to consider singleton and regular selection, indicated here by *one* and *reg*, respectively.

We denote by $[E]LSP_m(con_X, in, (n_1, n_2; n_3, n_4))$ the family of all languages $L(\Pi)$ generated by contextual systems Π with selection of type X, of degree at most m, and of weight componentwise smaller than $(n_1, n_2; n_3, n_4)$.

The power of contextual P systems is very large – but we must remember that "context-sensitivity and erasing are universal", and we have here both these two necessary features.

Theorem 5.4.1. $ELSP_2(con_{one}, in, (0,3;0,3)) = RE$.

Proof. Let $G = (N, T, S, R)$ be a type-0 grammar in the Kuroda normal form, and construct the system

$$\begin{aligned}
\Pi &= (N \cup T \cup \{c_1, c_2\}, T, [_1[_2\ ]_2]_1, \{S\}, \emptyset, R_1, R_2),\\
R_1 &= \{(\{A\}, (\lambda, c_2x); in)_a \mid A \to x \in R\}\\
&\quad \cup \{(\{AB\}, (\lambda, c_1CD); in)_a \mid AB \to CD \in R\}\\
&\quad \cup \{(\{\lambda\}, (\lambda, \lambda); out)_a\},\\
R_2 &= \{(\{\lambda\}, (\lambda, Ac_2); out)_e,\\
&\qquad (\{\lambda\}, (\lambda, ABc_1); out)_e\}.
\end{aligned}$$

The equality $L(G) = L(\Pi)$ is obvious: for each rule $u \to v \in R$ we add αv in membrane 1 and then we erase $u\alpha$ in membrane 2, with $\alpha = c_2$ for context-free rules and $\alpha = c_1$ for non-context-free rules. □

The weight can be slightly improved (no digit 3 will be used) by adding one further membrane.

Theorem 5.4.2. $ELSP_3(con_{one}, in, (0,2;2,0)) = RE$.

Proof. Let $G = (N, T, S, R)$ be a type-0 grammar in the Penttonen normal form. We construct the contextual P system

$$\begin{aligned}
\Pi &= (V, T, [_1[_2[_3\ ]_3]_2]_1, \{S\}, \emptyset, \emptyset, R_1, R_2, R_3),\\
V &= N \cup T \cup \{\alpha' \mid \alpha \in N\} \cup \{c\},\\
R_1 &= \{(\{XY\}, (\lambda, Z'); in)_a \mid XY \to XZ \in R\}\\
&\quad \cup \{(\{X\}, (\lambda, \alpha_1'\alpha_2); in)_a \mid X \to \alpha_1\alpha_2 \in R\}\\
&\quad \cup \{(\{\lambda\}, (X, \lambda); here)_e \mid X \to \lambda \in R\}\\
&\quad \cup \{(\{\lambda\}, (\lambda, \lambda) : out)_a\},\\
R_2 &= \{(\{\alpha\}, (\alpha'c, \lambda); out)_e \mid \alpha \in N \cup T\}\\
&\quad \cup \{(\{\alpha'\}, (X, \lambda); in)_e \mid X \in N, \alpha \in N \cup T\},\\
R_3 &= \{(\{\alpha'\}, (\lambda, c\alpha); out)_a \mid \alpha \in N \cup T\}.
\end{aligned}$$

The rules of the grammar G are simulated in the system Π as follows.

Consider a rule $X \to \alpha_1\alpha_2 \in R$. In membrane 1 we pass from a string w_1Xw_2 to $w_1X\alpha_1'\alpha_2w_2$, which is sent to membrane 2. Here we erase X from the left of the primed symbol, and the string $w_1\alpha_1'\alpha_ew_2$ is sent to membrane 3. Here we append $c\alpha_1$ to the right of α_1' and the string $w_1\alpha_1'c\alpha_1\alpha_2w_2$ is sent to

membrane 2. In membrane 2 we erase $\alpha_1' c$ and the string $w_1\alpha_1\alpha_2 w_2$ is sent back to the skin membrane.

The simulation of rules $X \to \lambda$ is done in membrane 1 only.

To simulate a rule $XY \to XZ$ from R, the symbol Z' is adjoined to the right of XY in membrane 1 and the obtained string is moved to membrane 2, where Y is erased from the left of Z' and the string is sent to membrane 3. Here, cZ is adjoined to the right of XZ' and the string is sent out. In membrane 2, $Z'c$ is erased from the left of Z and the string is sent back to the skin membrane.

At any time we can send the string out of membrane 1, but it is accepted only if it is terminal. Thus, $L(G) = L(\Pi)$. □

In the previous two characterizations of recursively enumerable languages we have used extended systems. The terminal alphabet can be avoided in the following way. Instead of starting from strings x, consider strings of the form cxc, where c is a new symbol. Instead of the rule $(\{\lambda\}, (\lambda, \lambda); out)_a$, as used in the previous proofs, consider the erasing rule $(T^*, (c, c); out)_e$. Then, only terminal strings are sent out, hence T is no longer necessary – but the price is the use of one rule with a regular selector.

Can the extended feature be avoided also for systems with finite selection? (Problem Q18.) We *conjecture* that the answer is positive.

5.5 Insertion–Deletion Membrane Systems

We recall that insertion–deletion rules are a sort of dual of contextual adjoining–erasing rules: this time, a string is inserted in or deleted from a given context. Specifically, an insertion rule is of the form $(u, x, v)_a$, with the meaning that a string $w = w_1uvw_2$ is transformed by such a rule to $w' = w_1uxvw_2$ (the string x was inserted in the context (u, v)). A deletion rule is of the form $(u, x, v)_e$, and it is used as follows: from $z = z_1uxvz_2$ we pass to $z' = z_1uvz_2$ (the substring x was deleted from the context (u, v)). For uniformity with the previous section, we indicate the type of rules by adding the subscripts a and e, for "insertion", respectively "deletion".

In the case of insertion–deletion membrane systems, the rules will have associated targets, hence they will be of the form $(u, x, v; tar)_\alpha$, with $tar \in \{here, out, in\}$ and $\alpha \in \{a, e\}$.

We say that an insertion–deletion P system $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m)$ (the meaning of the components is the same as in the previous sections) is *of weight* $(n_1, n_2; n_3, n_4)$ if

$$
\begin{aligned}
n_1 &= \max\{|x| \mid (u, x, v)_a \in R_i, 1 \le i \le m\},\\
n_2 &= \max\{|u| \mid (u, x, v)_a \in R_i, \text{ or } (v, x, u)_a \in R_i, 1 \le i \le m\},\\
n_3 &= \max\{|x| \mid (u, x, v)_e \in R_i, 1 \le i \le m\},\\
n_4 &= \max\{|u| \mid (u, x, v)_e \in R_i, \text{ or } (v, x, u)_e \in R_i, 1 \le i \le m\}.
\end{aligned}
$$

We denote by $[E]LSP_m(insdel, in, (n_1, n_2; n_3, n_4))$ the family of all languages $L(\Pi)$ generated by insertion–deletion systems Π of degree at most $m \geq 1$, and of weight componentwise smaller than $(n_1, n_2; n_3, n_4)$.

As expected, the insertion–deletion P systems are also very powerful. Actually, by insertion only (and two membranes) we can generate non-context-free languages.

Theorem 5.5.1. $LSP_2(insdel, in, (2, 1; 0, 0)) - CF \neq \emptyset$.

Proof. Consider the (non-extended) system

$$\begin{aligned}
\Pi &= (\{a, b, c\}, \{a, b, c\}, [_1[_2\ ]_2]_1, \{abc\}, \emptyset, R_1, R_2),\\
R_1 &= \{(a, ab, b; in)_a,\ (\lambda, \lambda, \lambda; out)_a\},\\
R_2 &= \{(b, c, c; out)_a\}.
\end{aligned}$$

The equality $L(\Pi) = \{a^n b^n c^n \mid n \geq 1\}$ is obvious (in membrane 1 we increase the number of occurrences of a and b, and in membrane 2 we also increase the number of occurrences of c). □

If we use systems with a "large enough" weight, then one component suffices in order to characterize the recursively enumerable languages.

Theorem 5.5.2. $ELSP_1(insdel, (3, 1; 2, 0)) = RE$.

Proof. Take a type-0 grammar $G = (N, T, S, R)$ in the Geffert normal form, with $N = \{S, A, B, C, D\}$ and rules of the forms $S \to uSv, S \to x$, for $u, v, x \in (\{A, B, C, D\} \cup T)^*$, and $AB \to \lambda, CD \to \lambda$. The rules $S \to uSv$ and $S \to x$ can be easily replaced by rules of the forms $X \to \alpha, X \to \alpha Y, X \to Y\alpha$, for X, Y non-terminals (new, or equal to S), and $\alpha \in \{A, B, C, D\} \cup T$ (this is a standard construction: for instance, from $S \to ABCSC$ we pass to $S \to AX_1$, $X_1 \to BX_2$, $X_2 \to CX_3, X_3 \to SC$, with X_1, X_2, X_3 associated with this rule). Consider the grammar $G' = (N', T, S, R')$ of this form, and construct the following insertion–deletion P system:

$$\begin{aligned}
\Pi &= (N' \cup T \cup \{H, c\}, T, [_1\ ]_1, \{SH\}, R_1),\\
R_1 &= \{(X, cx, \alpha; here)_a \mid X \to x \in R', \alpha \in N' \cup T \cup \{H\}\}\\
&\quad \cup \{(\lambda, Xc, \lambda; here)_e \mid A \in N'\}\\
&\quad \cup \{(\lambda, AB, \lambda; here)_e,\ (\lambda, CD, \lambda; here)_e\}\\
&\quad \cup \{(\lambda, H, \lambda; out)_e\}.
\end{aligned}$$

The insertion rules simulate the context-free rules $X \to x$ from R' in the following way. From a string $w_1 X w_2$, with X being the unique non-terminal different from A, B, C, D, we pass to $w_1 X c x w_2$ by using the rule $(X, cx, \alpha; here)_a$. Because the right-hand contexts α must be different from c, we cannot again use the same symbol X, hence we have to remove the substring Xc by the rule $(\lambda, Xc, \lambda; here)_e$, which completes the simulation.

The erasing rules $AB \to \lambda$, $CD \to \lambda$ are simulated by deletion rules of Π. If a rule $(\lambda, AB, \lambda; here)_e$ or $(\lambda, CD, \lambda; here)_e$ is used before removing the symbol c, then this is correct with respect to the grammar G', because X is different from A, B, C, D. Consequently, $L(G) = L(\Pi)$. □

The weight can be significantly reduced if we use a "more structured" system.

Theorem 5.5.3. $ELSP_4(insdel, in, (1,1;1,1)) = RE$.

Proof. Let $G = (N, T, S, R)$ be a type-0 grammar in the Penttonen normal form. Remember that the context-free rules $X \to x \in R$ have $|x| \leq 2$. Without loss of generality, we may assume that we have either $|x| = 2$ or $|x| = 0$: we replace each rule $X \to \alpha$ by $X \to \alpha Z$, $Z \to \lambda$, where Z is a new symbol. We construct the insertion–deletion P system (of degree 4)

$$
\begin{aligned}
\Pi &= (V, T, [_1[_2[_3[_4\ ]_4]_3]_2]_1, \{Sc_2\}, \emptyset, \emptyset, \emptyset, R_1, R_2, R_3, R_4),\\
V &= N \cup T \cup \{\langle\alpha\rangle \mid \alpha \in N\} \cup \{\langle\alpha_1\alpha_2\rangle \mid \alpha_1, \alpha_2 \in N \cup T\}\\
&\quad \cup \{c_1, c_2\} \cup \{\alpha' \mid \alpha \in N \cup T\},\\
R_1 &= \{(X, \langle\alpha_1\alpha_2\rangle, \lambda; in)_a \mid X \to \alpha_1\alpha_2 \in P\}\\
&\quad \cup \{(X, \langle Z\rangle, Y; in)_a \mid XY \to XZ \in P\}\\
&\quad \cup \{(\lambda, X, \lambda; here)_e \mid X \to \lambda \in P\}\\
&\quad \cup \{(\lambda, \lambda, \lambda; out)_e\},\\
R_2 &= \{(\langle Z\rangle, Y, \lambda)_e \mid Y, Z \in N\}\\
&\quad \cup \{(\lambda, X, \langle\alpha_1\alpha_2\rangle; in)_e \mid X \in N, \alpha_1, \alpha_2 \in N \cup T\}\\
&\quad \cup \{(\lambda, \alpha', \lambda; out)_e \mid \alpha \in N \cup T\}\\
&\quad \cup \{(\lambda, c_1, \lambda; out)_e\},\\
R_3 &= \{(\langle Z\rangle, Z, \lambda; in)_a \mid Z \in N\}\\
&\quad \cup \{(\lambda, \alpha_1, \langle\alpha_1\alpha_2\rangle; in)_a,\\
&\qquad (\alpha_1, \alpha_2, \alpha_2'; out)_a \mid \alpha_1, \alpha_2 \in N \cup T\}\\
&\quad \cup \{(c_1, \lambda, \lambda; out)_a\},\\
R_4 &= \{(X, c_1, \langle Z\rangle; here)_a \mid X, Z \in N\}\\
&\quad \cup \{(c_1, \langle Z\rangle, \lambda; out)_e \mid Z \in N\}\\
&\quad \cup \{(\langle\alpha_1\alpha_2\rangle, \alpha_2', \beta; here)_a \mid \alpha_1, \alpha_2 \in N \cup T, \beta \in N \cup T \cup \{c_2\}\}\\
&\quad \cup \{(\lambda, \langle\alpha_1\alpha_2\rangle, \alpha_2'; out)_e \mid \alpha_1, \alpha_2 \in N \cup T\}.
\end{aligned}
$$

The simulation of a rule $XZ \to YZ$ is controlled by the symbol $\langle Z\rangle$, while the simulation of a rule $X \to \alpha_1\alpha_2$ is controlled by the symbol $\langle\alpha_1\alpha_2\rangle$.

Consider, for instance, a string w_1XYw_2 in the skin region and apply the rule $(X, \langle Z\rangle, Y; in)_a \in R_1$. The string obtained, $w_1X\langle Z\rangle Yw_2$, is sent to membrane 2, where Y is erased by the rule $(\langle Z\rangle, Y, \lambda)_e$, and the string obtained, $w_1X\langle Z\rangle w_2$, is sent to membrane 3. Here we insert Z to the right

of $\langle Z\rangle$, and we send the string $w_1X\langle Z\rangle Zw_2$ to membrane 4. In membrane 4 we first add the symbol c_1 to the left of $\langle Z\rangle$, by the rule $(X, c_1, \langle Z\rangle; here)_a$, then we remove the symbol $\langle Z\rangle$ (this is possible only in the presence of c_1) and we send the string $w_1Xc_1Zw_2$ to membrane 3. The string is sent forward to membrane 2 unchanged, by the rule $(c_1, \lambda, \lambda; out)_a$, and in membrane 2 we remove the symbol c_1. The string obtained, w_1XZw_2, is sent to the skin membrane. It is the correct string obtained by using the rule $XY \to XZ$ from the grammar G.

The reader can check that also the context-free rules from R are correctly simulated in Π, hence we have the equality $L(G) = L(\Pi)$. □

Note that in the previous result we insert and delete strings of the minimal non-trivial length, one, and the contexts for these operations are also of a minimal non-trivial length, one. We will immediately see (Theorem 5.5.4) that the contexts (u, v) can be made (λ, λ), providing that the insertion–deletion operations are performed in a conditional way, depending on the strings present in the same membrane. Otherwise stated, the control by the immediate context of an insertion–deletion rule, checked on the processed string, can be replaced by the control of the "crowd", by checking whether or not a given string is present at the same time in the region of the system where we are working.

Specifically, we consider rules of the form $(u, x, v; tar)_\alpha|_w$, where $\alpha \in \{a, e\}$, and w is a string. Such a rule is applied in a region i to a string y, and we adjoin or remove its substring x from the context (u, v) only if w is another string from region i; the result of this operation is sent to the membrane indicated by *tar*. After this operation, the string y (its copy involved in this operation) is consumed. The "promoter string" w can be processed at the same step.

We denote by $[E]LSP_m(insdel, in, (n_1, n_2, ; n_3, n_4), cond)$ the family of languages generated by [extended] P systems of degree at most $m \geq 1$ and of weight componentwise smaller than $(n_1, n_2; n_3, n_4)$, with conditional insertion–deletion rules, using target commands *here, out, in.* As usual, any of the parameters m, n_1, n_2, n_3, n_4 is replaced by $*$ when it is not bounded.

As we already mentioned, the conditional use of rules makes the contexts of insertion–deletion operations unnecessary.

Theorem 5.5.4. $ELSP_1(insdel, (*, 0; 2, 0), cond) = RE.$

Proof. Consider a type-0 grammar $G = (N, T, S, R)$ in the Geffert normal form, with $N = \{S, A, B, C, D\}$ and R consisting of context-free rules of the forms $S \to uSv$ and $S \to u$, with $u, v \in (T \cup \{A, B, C\})^+$, and the erasing rules $AB \to \lambda, CD \to \lambda$ (if one of u, v is empty, then we can replace it with AB, and this does not change the generated language). We label the context-free rules from R with $r_1, \ldots, r_k$; we write them in the form $r_i : S \to x_i, 1 \leq i \leq k$. We also add the rule $r_0 : S \to SAB$; it is clear that it does not change the generated language. We construct the conditional insertion–deletion P system

$$
\begin{aligned}
\Pi &= (V, T, [_1\ ]_1, \{S, EF_0, H\}, R_1),\\
V &= N' \cup T \cup \{c, S, A, B, C, D, E, F, H, H', \#\} \cup \{F_i \mid 0 \le i \le k\},\\
R_1 &= \{(\lambda, cx_i, \lambda; here)_a|_{EF_i},\\
&\qquad (\lambda, F_i, \lambda; here)_a|_H,\\
&\qquad (\lambda, \lambda, \lambda; here)_a|_{EF_i} \mid 0 \le i \le k\}\\
&\quad \cup \{(\lambda, \#, \lambda; here)_a|_H,\\
&\qquad (\lambda, F, \lambda; here)_a|_H,\\
&\qquad (\lambda, H', \lambda; here)_a|_{EF_0F}\}\\
&\quad \cup \{(\lambda, F_i, \lambda; here)_e|_H \mid 0 \le i \le k\}\\
&\quad \cup \{(\lambda, Sc, \lambda; here)_e|_E,\\
&\qquad (\lambda, AB, \lambda; here)_e|_{EF_0F},\\
&\qquad (\lambda, CD, \lambda; here)_e|_{EF_0F},\\
&\qquad (\lambda, AB, \lambda; here)_e|_{HH'},\\
&\qquad (\lambda, CD, \lambda; here)_e|_{HH'},\\
&\qquad (\lambda, \lambda, \lambda; out)_a|_{HH'}\}.
\end{aligned}
$$

We start with three strings, S, EF_0, H, in the system. The non-terminal $\#$ is a trap symbol, since there is no deletion rule containing it, hence by using the insertion rule $(\lambda, \#, \lambda; here)_a|_H$ we get strings which never lead to terminal strings and, also, are not promoters for the rules from R_1. Therefore, the use of this rule must be avoided. This means that at each step we have to process the strings produced from S, but by any of the rules from R_1 different from $(\lambda, \#, \lambda; here)_a|_H$.

Assume that we are in a step when three strings z, EF_i, H are available, with $z \in (T \cup \{S, A, B, C, D\})^*$ containing only one occurrence of S (initially, $z = S$ and $i = 0$; there is no change of the generated language because we always start with this rule). If any insertion rule different from $(\lambda, H', \lambda; here)_a|_{EF_0F}$ is applied to the string H, then the resulting strings can never be used as a promoter for rules used to process the string z and its successors. Similarly, to EF_i we have to apply the deletion rule $(\lambda, F_i, \lambda; here)_e|_H$ or the insertion rule $(\lambda, F, \lambda; here)_a|_H$. Thus, the only possibility which does not block the work of the system or does not introduce the trap symbol $\#$ is that one when we apply the insertion rule $(\lambda, cx_i, \lambda; here)_a|_{EF_i}$ to z, the deletion rule $(\lambda, F_i, \lambda; here)_e|_H$ to EF_i, and the insertion rule $(\lambda, \lambda, \lambda; here)_a|_{EF_i}$ to H. In the case of strings EF_i and H the results are unique: E and H, respectively. In the case of z we can have several possibilities, but the only variant which can lead to a terminal string is when cx_i is inserted immediately after the occurrence of S in z: at the next step we use either the deletion rule $(\lambda, Sc, \lambda; here)_e|_E$, or an insertion rule $(\lambda, \#, \lambda; here)_a|_H, (\lambda, F, \lambda; here)_a|_H$; in both these latter cases no terminal string can be sent out of the system.

Therefore, from $z = z_1Sz_2$ we first pass to $z_1Scx_iz_2$ and then to $z_1x_iz_2$, which corresponds to using the context-free rule $S \to x_i$ of G. If in parallel

with the second step we insert a symbol F_j in the string E, obtaining the string EF_j (the string F_jE is not useful as a promoter), and λ is inserted in the string H, then the process can be iterated. In this way we can simulate any derivation in the grammar G which uses context-free rules.

At any time, we can use the insertion rule $(\lambda, \lambda, \lambda; here)_a|_{EF_i}$ for the "main" string, that produced from the initial string S, and the insertion rule $(\lambda, F, \lambda; here)_a|_H$ for the string derived from EF_0. If we are in the case when $i = 0$ (the rule r_0 can be simulated any number of times and this does not change the generated language, because eventually all substrings AB introduced by this rule will be erased) and if we get the string EF_0F, then one of the deletion rules $(\lambda, AB, \lambda; here)_e|_{EF_0F}$, $(\lambda, CD, \lambda; here)_e|_{EF_0F}$ can be used. At the same time, H is changed to HH' or $H'H$ – but only the first string can be used as a promoter – and any rule is applied to EF_0F. At the next step, the string obtained from EF_0F is sent out of the system by the rule $(\lambda, \lambda, \lambda; out)_a|_{HH'}$ (this is a non-terminal string), and from now on the string HH' remains unchanged. It promotes the rules $(\lambda, AB, \lambda; here)_e|_{HH'}$, $(\lambda, CD, \lambda; here)_e|_{HH'}$, which can remove all substrings AB, CD from the obtained string, as well as the rule $(\lambda, \lambda, \lambda; out)_a|_{HH'}$, which sends the current string out of the system. If the string sent out is terminal, then it is from $L(G)$ and contributes to the language $L(\Pi)$, otherwise it is "lost".

Thus, $L(G) = L(\Pi)$, and this concludes the proof. □

5.6 Bibliographical Notes

Membrane systems with string–objects were considered in [186], both with rewriting and splicing rules (with a different definition in the splicing case: the splicing rules also have an associated string which enters the splicing operation, hence the splicing is used as a binary relation).

Theorem 5.1.5 is from [56]. The equality $LSP_3(rw, tar, pri) = RE$ is given in [186]; the improvements from Theorems 5.1.6 and 5.1.7 are from [176]. The 2-normal form was introduced in [245, 247], where also other results similar to Theorem 5.1.8 can be found, but using communication by means of electrical charges (also a 3-normal form is given in [245] for systems without priorities, but using polarizations). Several related normal forms are given in [121], from where the proof of Theorem 5.1.8 is borrowed.

The universality of P systems using the membrane permeability control (but not priorities) is first proved in [247], without a bound on the number of membranes. Theorem 5.1.9 is from [63], where also Theorems 5.1.10 and 5.1.11 (with a different proof) can be found. Systems with rule creation, both for the rewriting and the splicing case, were considered in [12], where results like Theorems 5.2.1 and 5.3.3 can be found. The conditional use of rules was considered in [56], where Theorems 5.2.2 and 5.2.3

are proved. Rewriting P systems with a leftmost derivation are investigated in [56]: in each step, each string which can be rewritten is rewritten in the leftmost symbol α for which a rule $\alpha \to v$ exists in that region. It is proved that $LSP_5(rw, in, leftmost) = ELSP_4(rw, in, leftmost) = RE$. Here are other results from [56] about systems with leftmost rewriting: $ET0L \subset ELSP_3(rw, in, leftmost)$, $0L \subset LSP_2(rw, in, leftmost)$, $E0L \subseteq ELSP_2(rw, in, leftmost)$. Theorems 5.2.4 and 5.2.5 (with a different proof) are from [146], where also splicing and insertion–deletion systems (actually, single grammars, because one membrane is sufficient in all cases), with the conditional use of rules in the sense of Theorem 5.2.4, are considered; Theorem 5.5.4 is from [146].

Subsection 5.2.3 is based on [29], where also prefix–suffix checking and shape checking are considered (in the first case the promoter/inhibitor must be a prefix or a suffix of a string, in the second case the whole string must belong to a given regular language). The replicated rewriting was considered in [39] (together with another operation which multiplies the strings, splitting: we consider rules $a \to u_1|u_2$ and, using a rule of this form, from w_1aw_2 we get both w_1u_1 and u_2w_2), and in [117]. The universality is given in [117] without a bound on the number of membranes; Theorem 5.2.12 is from [142], where also systems with less than six membranes are investigated (Theorems 5.2.13, 5.2.14, and 5.2.15). The closure properties of families $ELSP_m(rw, in, repl_2, halt)$ are also briefly investigated in [142]; in fact, only results of the following form are obtained: *if h is a morphism, R a regular language, and $L \in ELSP_m(rw, in, repl_2, halt)$, then $h(L)$ and $L \cap R$ belong to $ELSP_{m+1}(rw, in, repl_2, halt), m \geq 1$.* The question of investigating closure properties of these families in the usual sense is formulated as an *open problem* in [142], and we recall it here, for all families of languages from this chapter which are not (known to be) equal to families of languages from the Chomsky hierarchy (Problem Q19). Also of interest are "absorption results", as those from [142] mentioned above. In general, given a (finite or infinite) hierarchy of language families, $FL_m, m \geq 1$, and an operation g with languages – say, an n-ary one – such that the families FL_m are not (known to be) closed under the operation g, it is of interest to find results of the following form (Problem Q20): *if $L_1, \ldots, L_n \in FL_m$, then $g(L_1, \ldots, L_n) \in FL_{m+p_g}$, for some p_g depending on g.* When $p_g = 0$ we have a usual closure property.

Theorem 5.2.16 is given in [39], but also using the splitting operation, as well as a crossover operation, related to splicing (one gives "jumping sites", strings x, and from uxv, $u'xv'$ one produces the strings uxv' and $v'xu$).

The partial parallelism, as used in Theorem 5.2.17, was considered in [114], where the universality is given without a bound on the number of membranes. Six membranes were proven in [177] to be sufficient, then the result was improved to the form of Theorem 5.2.17 in [121]. In [114] the form of parallelism denoted at the beginning of Subsection 5.2.5 with (b) is also considered (maximal parallelism: rewrite all symbols which can be

rewritten), but with a rather special mode of deciding where the resulting string should be sent: one chooses that target, one of *here, out, in*, which appears exactly once in the used rules; if all targets appear at least twice, then the computation is blocked and no result is produced. Rewriting P systems with this type of parallelism and this type of communication protocol is proven to be universal (but no upper bound on the number of membranes is given). These results can also be found in [108].

The universality of splicing P systems is obtained in [186] for four membranes; the result is improved to three membranes in [201], then to two membranes in [182]. Also the diameter of splicing rules is considered in [182]. From this point of view, the results from [182] were improved in [69], where the proof of Theorem 5.3.1 can be found. Theorem 5.3.2 is from [176].

P systems with immediate communication, as in Corollary 5.2.3 and Theorem 5.3.4, were considered in [159]. Both [201] and [69] also investigate the case of systems with the membranes arranged in the nodes of a graph and using the splicing operation. Splicing P systems are also investigated in [202], where universality results are proved by directly simulating universal classes of H systems by means of P systems; specifically, H systems with permitting or forbidding contexts, as well as so-called test-tube H systems, are considered (see [195] for the definition of these types of H systems).

Contextual P systems were investigated in [112] and [134]. Section 5.4 is based on the second paper (the first paper, [112], contains many results about the comparison of families of languages generated by contextual P systems with families of languages generated by contextual grammars of various types). Hybrid P systems, containing both rewriting rules and contextual rules, are investigated in [111], where a universality result is given for systems with seven membranes, using rewriting rules and adjoining contextual rules with regular selectors.

Insertion–deletion P systems are introduced in [116], where Theorem 5.5.3 can be found, as well as families which cover MAT^{λ} and $ET0L$.

Surveys of results about string–object P systems can also be found in [151, 153], as well as in [194] and [155].

A variant of the systems with "worm-like-objects", as considered in [39], was investigated in [161]: systems with three membranes, using rewriting, splitting, and the so-called suffix–prefix concatenation (one gives pairs (u, v) and one passes from w_1, w_2 to $w_1 w_2$ only if $u \in Suf(w_1)$ and $v \in Pref(w_2)$), are shown to be universal. The result is obtained even with length-increasing rewriting and splitting, but then using four membranes.

Rewriting and splicing P systems with global rules are also investigated in [107].

In a series of papers, [58, 60, 62, 65, 227], rewriting, splicing, and cut-and-paste P systems of the generalized form also mentioned at the end of Chapter 3 are considered, with operators acting on rules, with rules moving from one membrane to another one, etc. Many universality results can be

found in these papers – proved either starting from Chomsky grammars, from matrix grammars, from graph-controlled grammars, or simulating H systems of types already known to be universal. In Subsection 8.6.1 we will recall some definitions and results from this area.

The PhD theses [108] and [242] are devoted to a large extent to string–object P systems.

6. Networks of Membranes

In the systems considered in the previous chapters, the membrane structure corresponds to a cell-like structure, with the membranes arranged hierarchically. Such an arrangement is described by a tree. From a mathematical point of view it is then natural to consider related machineries whose underlying membrane structure is described by a graph of a given form, more restrictive or more general than a tree. More restrictive can be, for instance, a binary tree (each membrane directly contains two membranes), a linear tree (with all membranes arranged vertically, one into another), or a star (a tree with only two levels, hence with all internal membranes placed in the skin membrane). Because many universality results given in the previous chapters refer to systems with only two membranes, we implicitly have a sort of "normal form" in what concerns the underlying tree: of depth 2, linear.

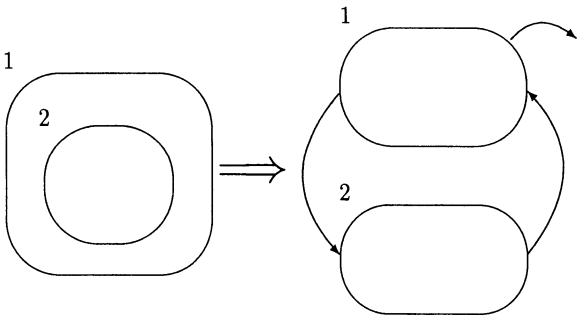


Fig. 6.1. Passing to a ring

A more interesting direction of research is to look for membrane structures which are not necessarily described by a tree. This means considering membranes placed in the nodes of a graph and communicating along the edges of the graph. Graphs of simple forms are the first to be considered – for instance, rings. From the many two-membrane universality results we directly obtain universality results for P systems with the membranes placed in the nodes of a ring graph: the two membranes can be considered as placed in the same level, communicating in both directions, as suggested in Figure 6.1;

membrane 1 can send objects to the environment, membrane 2 will be the output one.

In an easy way we can also get a system such that any two membranes communicate (at most) in a one-way manner: starting from a system with two membranes, labelled by 1 and 2, just consider one more membrane, labelled with 3, and let the communication from membrane 1 to membrane 2 go directly but send objects from membrane 2 to membrane 1 only through membrane 3. Some problems appear here concerning the synchronization of the two "main" membranes, 1 and 2, because the communication from 2 to 1 will take two time units, but this can be arranged. A case where this can be done will be presented in the next section.

Membranes forming a network can have not only a mathematical motivation, but also a biological one. Cells are almost never alone; they form tissues, organs, bodies, they interact with each other, either directly or by means of the common environment. A specially interesting case is that of neurons, which are linked in a specific manner and work together in a complex net.

Thus, it is rather natural to "exit the cell" and to look for "tissue-like" or "neural-like" membrane systems – and this is the topic we deal with in this chapter.

6.1 The Splicing Case

One additional membrane, introduced to obtain a one-way communication graph as suggested above, can also be used to improve on other parameters of the obtained system. We illustrate this only for the splicing case.

For P systems with the membranes placed in the nodes of a (directed) graph we have to use target indications of the forms *here*, with the usual meaning, go_j, for "go to the membrane with label j", and *out*, for"go out of the system". In the case of a ring, *go* suffices, because the destination is unique, but we can use *go* instead of go_j in general, in the same way as we use *in* instead of in_j: "go to any membrane accessible from the present node".

Instead of a membrane structure given as a parenthetic expression, we now have to specify the used graph. As always the nodes are the m membranes of the system; we have only to give the links among membranes, as directed pairs (i, j) with the meaning that membrane i can send objects to membrane j.

Thus, a splicing P system with the membranes arranged in the nodes of a graph will be given as a construct $\Pi = (V, T, g, M_1, \ldots, M_m, R_1, \ldots, R_m)$, where all components are as usual, and g is a subset of $\{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ specifying the links among membranes.

Let us denote by $[E]LStP_m(spl, go, (n_1, n_2, n_3, n_4))$ the family of languages generated by [extended] splicing P systems with the membranes arranged on a ring (hence with a one-way communication among them), of degree at most $m \geq 1$, with the diameter componentwise at most (n_1, n_2, n_3, n_4).

In general, we will add a "t" in front of "P" in the names of families generated by tissue-like P systems.

The number of membranes from the next result is a direct consequence of the discussion above, but the improvement in the diameter of the splicing rules with respect to Theorem 5.3.1 and its corollary is due to the extra facilities provided by the third membrane.

Theorem 6.1.1. $ELStP_3(spl, go, (0,2,1,0)) = ELStP_3(spl, go, (1,0,0,2)) = RE$.

Proof. According to Lemma 5.3.1, we only have to prove one of the equalities with RE. We proceed as in the proof of Theorem 5.3.1: we start from a type-0 grammar in the Kuroda normal form, and, with the same notations as in the mentioned proof, we construct the system

$$
\begin{aligned}
\Pi &= (V, T, g, M_1, M_2, M_3, R_1, R_2, R_3),\\
V &= N \cup T \cup \{B, X, X', Y, Y', Z_X, Z_{X'}, Z_Y, Z_{Y'}, Z_\lambda, Z'_\lambda\}\\
&\quad \cup \{X_i, Y_i, Z_i, Z_{X_i}, Z_{Y_i} \mid 1 \le i \le n+m\}\\
&\quad \cup \{Y'_i, Z_{Y'_i} \mid u_i \to v_i \in R_{CS}\},\\
g &= \{(1,2), (2,3), (3,1)\},\\
M_1 &= \{XBSY,\ X'Z_{X'},\ Z_\lambda,\ Z'_\lambda\} \cup \{Z_{Y_i}Y_i \mid 1 \le i \le n+m\}\\
&\quad \cup \{X_iZ_{X_i} \mid 1 \le i \le n+m-1\} \cup \{Z_{Y'_i}Y'_i \mid u_i \to v_i \in R_{CS}\},\\
M_2 &= \{Z_{Y'}Y'\} \cup \{X_iv_iZ_i \mid 1 \le i \le n+m\}\\
&\quad \cup \{Z_{Y_i}Y_i \mid 1 \le i \le n+m-1\},\\
M_3 &= \{XZ_X,\ Z_YY\} \cup \{X_iZ_{X_i} \mid 2 \le i \le n+m\},\\
R_1 &= \{(\lambda\natural u_iY\$Z_{Y_i}\natural\lambda; go, out) \mid 1 \le i \le n+m\}\\
&\quad \cup \{(\lambda\natural CY\$Z_{Y'_i}\natural\lambda; here, out),\\
&\qquad (\lambda\natural AY'_i\$Z_{Y_i}\natural\lambda; go, out) \mid u_i \to v_i \in R_{CS}, u_i = AC\}\\
&\quad \cup \{(\lambda\natural Z_{X_{i-1}}\$X_i\natural\lambda; go, out) \mid 2 \le i \le n+m\}\\
&\quad \cup \{(\lambda\natural Z_{X'}\$X_1\natural\lambda; go, out),\\
&\qquad (\lambda\natural BY\$Z_\lambda\natural\lambda; here, out),\\
&\qquad (\lambda\natural Z'_\lambda\$X\natural\lambda; out, out)\},\\
R_2 &= \{(\lambda\natural Z_i\$X\natural\lambda; go, go) \mid 1 \le i \le n+m\}\\
&\quad \cup \{(\lambda\natural Y_i\$Z_{Y_{i-1}}\natural\lambda; go, go) \mid 2 \le i \le n+m\}\\
&\quad \cup \{(\lambda\natural Y_1\$Z_{Y'}\natural\lambda; go, go)\},\\
R_3 &= \{(\lambda\natural Z_{X_i}\$X_i\natural\lambda; go, here) \mid 2 \le i \le n+m\}\\
&\quad \cup \{(\lambda\natural Z_X\$X'\natural\lambda; go, go),\\
&\qquad (\lambda\natural Y'\$Z_Y\#, here, go)\}.
\end{aligned}
$$

The fact that $L(G) = L(\Pi)$ can be proved in the same way as in the proof of Theorem 5.3.1. Note that only membrane 1 can send strings out of the system. □

6.2 Using Symport/Antiport Rules

Especially useful for communication in networks of membranes are the symport/antiport rules.

Imagine a network of elementary membranes such that some of them are linked by "communication channels", others not; the membranes contain multisets of symbol–objects, as well as communication rules stating how the membranes which are linked can directly communicate among them, and how all membranes can communicate with the environment.

The communication with the environment can be realized through the usual symport and antiport rules, as considered in Chapter 4, that is rules of the form (x, in) and (x, out) (symport rules), and rules of the form $(x, out; y, in)$ (antiport rules), but in the case of the direct communication between membranes also the target must be indicated. More specifically, the symport rules will be of the form (i, x, j), meaning that the multiset of objects specified by the string x will be removed from membrane i and introduced into membrane j. Similarly, the antiport rules will be of the form $(i, x/y, j)$, meaning that the multiset of objects specified by the string x will move from membrane i to membrane j, simultaneously with the passage of the multiset of objects specified by y, which move from membrane j to membrane i. Note that the symport rules (i, x, j) are directional, while the antiport rules $(i, x/y, j)$ allow just for an exchange of objects between membranes i and j and therefore they are symmetric: both rules $(i, x/y, j), (j, y/x, i)$ define the same communication/exchange. Each membrane can interact with the environment (which is a major difference with cell-like systems, where only the skin membrane can communicate with the environment), but not necessarily all pairs of membranes can directly communicate with each other. However, through the environment, each membrane can communicate with any other membrane, in two steps: one membrane sends out objects which the second membrane can bring inside.

For uniformity, in what follows we will specify the rules for communication with the environment in a manner similar to the way the rules for the internal communication are specified – to this end we assume that the environment is labelled by label 0. Thus, instead of $(x, in) \in R_i$ we will write $(0, x, i)$ and instead of $(x, out) \in R_i$ we will write $(i, x, 0)$.

More formally, a *tissue-like membrane system* (of degree $m \geq 1$) *with symport/antiport* is a construct

$$\Pi = (O, w_1, \ldots, w_m, E, R, i_o),$$

where O is the alphabet of *objects*, $w_1, \ldots, w_m$ are strings over O representing the multisets of objects initially present in the membranes of the system, E is the set of objects present in arbitrarily many copies in the *environment*, $i_o \in \{1, \ldots, m\}$ is the output membrane, and R is a finite set of rules of the following forms:

1. $(i, x, 0)$ and $(0, x, i)$, for $1 \leq i \leq m$ and $x \in O^+$ (symport rules for the communication with the environment),
2. $(i, x/y, 0)$, for $1 \leq i \leq m$ and $x, y \in O^+$ (antiport rules for the communication with the environment),
3. (i, x, j), for $1 \leq i, j \leq m$ and $x \in O^+$ (symport rules for the communication between membranes),
4. $(i, x/y, j)$, with $i, j \in \{1, 2, \ldots, m\}, i \neq j$, and $x, y \in O^+$ (antiport rules used for the communication between membranes).

The work of such a system is as normal in membrane systems: in each time unit marked by a global clock, the rules from each membrane are used in a non-deterministic maximally parallel manner. This means that all objects which can pass through a membrane i (coming in or going out, according to the rules from R of the forms $(i, x, j), (j, x, i), (i, x/y, j)$) must do it. In this way, we obtain transitions among the configurations of the system. As usual, a configuration is a snapshot of a state of the system, and it is represented by a vector $(z_1, \ldots, z_m, z_0)$, where each $z_i, 1 \leq i \leq m$, gives the multiset of objects present in membrane i, and z_0 is the multiset of objects, other than those from E, which were sent to the environment during the computation. The initial configuration is $(w_1, \ldots, w_m, \lambda)$. A computation is successful only when it halts; that is, a configuration is reached where no rule can be used. The result of a successful computation consists of the number of objects present in the halting configuration in membrane i_o. We denote by $N(\Pi)$ the set of natural numbers computed in this way by the system Π.

The tissue-like P systems can be classified according to the characteristics of the *communication graph* $\gamma(R)$ defined by the communication rules from the set R as follows: $\gamma(R) = (\{1, 2, \ldots, m\}, U)$, where U is the set of undirected edges $\{i, j\}$ such that there is in R at least one rule of the form (i, x, j) or (j, x, i) or $(i, x/y, j)$ (that is, the membranes with labels i, j communicate directly by using rules from R). As basic parameters describing tissue-like P systems (besides the weight of the symport/antiport rules, as in Chapter 4) we consider the degree of the systems, and the maximal number of nodes in a connected component of the graph $\gamma(R)$. (A subgraph γ' of a graph γ such that (i) there is a path between each pair of nodes in γ', and (ii) γ' is maximal, no node of γ can be added to γ' such that property (i) is preserved, is called a *connected component* of γ.) Thus, intuitively, a small size of the maximal connected component of $\gamma(R)$ indicates a "loose" direct communication within the system Π.

The family of all sets of numbers $N(\Pi)$, computed by systems with at most m membranes, using symport rules of weight at most p and an-

tiport rules of weight at most q, and such that the maximal size of a connected component of the underlying communication graph is n, is denoted by $NOtP_{m,n}(sym_p, anti_q)$.

Let us consider an **example**, illustrating the previous definition. In order to make this example easier to follow, we do not give the system in a formal way, but rather in a pictorial way – see Figure 6.2. The rules are placed near the membrane with which they are associated or next to the link between membranes along which the objects are transported. The output membrane is the one with label 1, the initial contents of the membranes are given in the figure, while in the environment we have arbitrarily many copies of the objects $a, c_1', c_2', c_2'', d_1, d_1', d_2, d_2', d_2'', f, g, \#$.

This system computes the set of numbers $\{2^n \mid n \geq 2\}$. We do not give full details of the proof of this statement, but instead we provide some hints which should allow the reader to construct such a proof. # is a trap object: after it is introduced into membrane 3, the computation will never stop, because of the rule $(3, \#/\#, 0)$. We start with two copies of the object a in membrane 1. Assume now that at some moment of the computation we have n copies of a in membrane 1. For a while, the objects c_1, c_2 carry pairs of copies of a to membrane 2. The two objects c_1, c_2 must go together, otherwise the one which arrives alone in membrane 2 must pass through membrane 3, introducing # into this membrane. At any moment, the objects c_1 and c_2 can be replaced in membrane 1 by c_1' and c_2', respectively. The object c_1' checks whether any copy of a is still present in membrane 1, and in the affirmative case it moves into membrane 3, where # is produced. If no copy of a is present in membrane 1, then c_1' waits; then at the next step c_2' is replaced by c_2'', which passes together with c_1' to membrane 2. The objects c_1' and c_2'' are exchanged through membrane 2 with d_1 and d_2, respectively, which now carry back to membrane 1 all copies of a, and, at the same time, an equal number of copies of f (enough copies of f were brought into membrane 2 by g: we need two steps for bringing aa from membrane 1 to membrane 2, and in each step one copy of f is transported by g inside membrane 2). Now, d_1 and d_2 proceed as c_1 and c_2, checking whether or not all copies of a were moved into membrane 1 (if the result of this checking is negative, then the rule $(2, d_1'a, 3)$ is used, and then # is introduced in membrane 3). In membrane 1, all objects f are changed into a, and at the same time d_1' and d_2'' are changed into c_1 and c_2, respectively – thus the process of doubling the number of occurrences of a can be iterated. At any moment, d_2'' can choose not to go to membrane 1 together with d_1', but rather go to membrane 3 together with g. Then the computation stops because no further rule can be applied (d_1' remains in membrane 2, hence in membrane 1 we have only copies of the object a).

Before investigating the power of tissue-like systems with symport/antiport, let us examine in some further detail the communication in such a system.

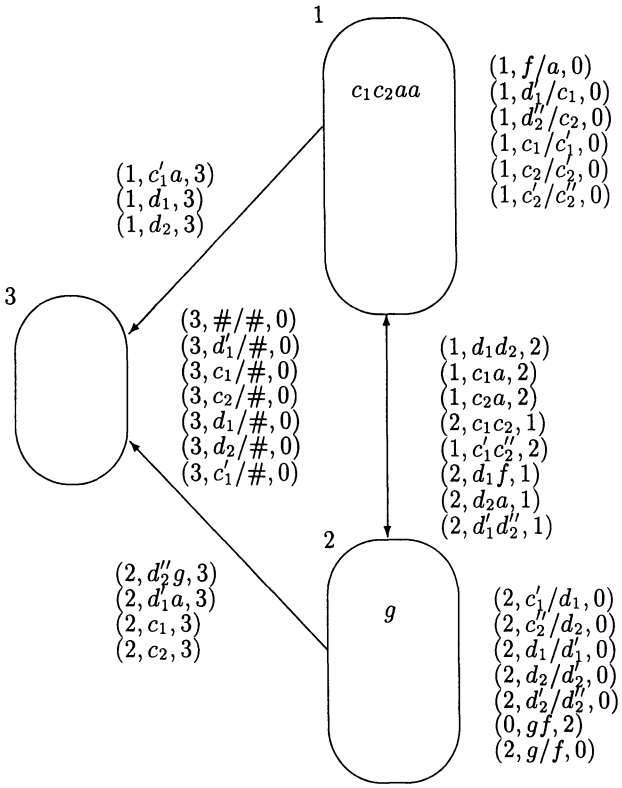


Fig. 6.2. Example of a tissue-like system with symport/antiport

First, we emphasize the fact that membranes can communicate both directly, through rules of the form $(i, x/y, j)$, and indirectly, through rules of the form $(i, x, 0), (0, x, j)$. In the latter case, the objects given by x are expelled from membrane i and are now available in the environment so that they can be introduced into membrane j, provided that a suitable rule exists.

Second, the rules $(i, x/y, j)$ establish a two-way communication among components i and j, while the rules (i, x, j) define a one-way communication. In order to understand the role of one-way communication, we will consider the case when we have only symport rules for communication between components, and between components and the environment (hence, the corresponding family of sets of numbers is $NOtP_{m,n}(sym_p, anti_0)$) and then, as a special subcase, we consider the case when for each pair i, j of components we have rules of at most one of the types (i, x, j) and (j, y, i). We say that such a system is *one-way*; let us denote by $1NOtP_{m,n}(sym_p, anti_0)$ the family of sets of numbers which can be computed by one-way systems.

It is easy to see that, using more membranes, for each tissue-like system Π without antiport rules we can construct a one-way system Π', equivalent to Π, in a way similar to that discussed at the beginning of this chapter: for each pair i, j of membranes which communicate in both directions, we introduce two new membranes, one for communicating from i to j, and one for communicating from j to i. Thus, the communication between i and j is done in a one-way manner, in two steps in each direction. In order to preserve the synchronization of the system, we introduce such additional membranes also for the communication with the environment. The fact that we have only symport rules ensures the equivalence of Π and Π'. Clearly, the number of components of Π' is quadratic with respect to the number of components of Π, as we can have $m(m-1)/2$ links among the m membranes of Π, and hence $m(m-1)+2m$ new membranes. However, the number of membranes can be made much smaller, as proved below.

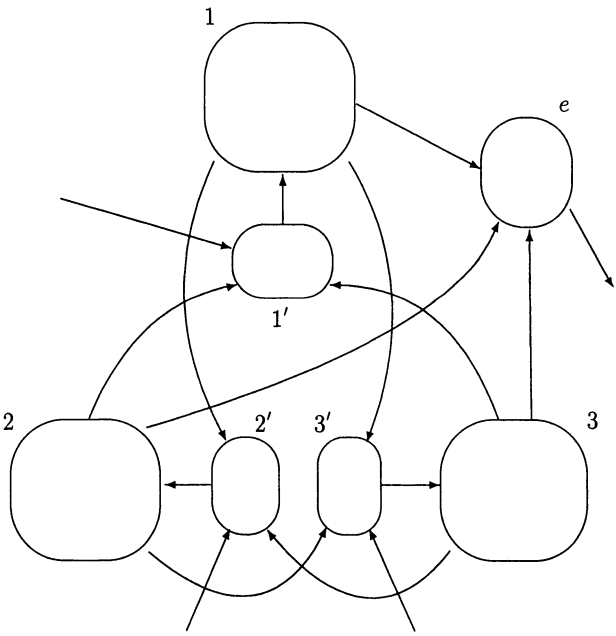


Fig. 6.3. The idea of the proof of Theorem 6.2.1

Theorem 6.2.1. $NOtP_{m,n}(sym_p, anti_0) \subseteq 1NOtP_{2m+1,2m+1}(sym_p, anti_0)$.

Proof. Instead of an easy but somewhat tedious formal proof, we explain the essential points used in the construction, illustrated in Figure 6.3 (for the case when $m = 3$). Consider a tissue-like system $\Pi = (O, w_1, \ldots, w_m, E, R, i_o)$, with R containing only symport rules. For each membrane i of Π we consider a "buffer" membrane, i'. All objects which are sent to membrane i,

by rules of the forms (j, x, i) and $(0, y, i)$, are first sent to membrane i', by rules (j, x, i') and $(0, y, i')$, respectively, and then to i, by rules (i', x, i) and (i', y, i), respectively. Symmetrically, if we have a rule (i, x, j), then we first use a rule (i, x, j') and then (j', x, j). In order to send objects into the environment without losing the synchronization, one "buffer" membrane for the environment is sufficient: consider a membrane labelled by e, and instead of rules $(i, x, 0)$ use the rules (i, x, e) and $(e, x, 0)$. Clearly, the weight of the symport rules is not changed, and the newly obtained system is equivalent to Π (and it has $2m+1$ membranes). Because all membranes of the new system can be connected (for instance, to the new membrane e), its communication graph can have only one connected component, of size $2m+1$. □

Clearly, cell-like membrane systems of degree 1 can be considered tissue-like systems, even with one-way communication, and conversely; consequently, $NOP_1(sym_p, anti_q) = 1NOtP_{1,1}(sym_p, anti_q)$.

On the other hand, it is easy to see that starting from a cell-like P system Π with symport/antiport rules, of any degree, we can construct an equivalent tissue-like system with symport/antiport rules, Π'. More specifically, each membrane of Π corresponds to a membrane of Π'. The corresponding membranes of the two systems have the same labels. The rules associated with the skin membrane of Π (assume that it is labelled by 1) are now used for the communication with the environment, changed from (x, in) to $(0, x, 1)$, from (x, out) to $(1, x, 0)$, and from $(x, out; y, in)$ to $(1, x/y, 0)$ (note that the only membrane of Π' which communicates with the environment is the one labelled by 1). Then, if membrane i is directly inside membrane j, its rules (x, out) are changed to (i, x, j), the rules (x, in) are changed to (j, x, i), and the antiport rules $(x, out; y, in)$ are changed to $(i, x/y, j)$. The output membrane of the two systems has the same label. The equivalence of Π and Π' constructed in this way is obvious. Thus we get the following result.

Theorem 6.2.2. *$NOP_m(sym_p, anti_q) \subseteq NOtP_{m,m}(sym_p, anti_q)$, for all $m \geq 1$ and all $p, q \geq 0$.*

By combining Theorem 6.2.2 with Theorem 4.2.2, we get the computational completeness of tissue-like P systems with symport rules using (at least) five membranes.

Corollary 6.2.1. *$NOtP_{5,5}(sym_2, anti_0) = NRE$.*

Now, in view of Theorem 6.2.1, we obtain the computational completeness of one-way systems (with at least 11 membranes).

Corollary 6.2.2. *$1NOtP_{11,11}(sym_2, anti_0) = NRE$.*

It is highly probable that these results can be improved as far as the number of membranes is concerned (note that we use only symport rules of minimal weights, as encountered in biology). If we also allow antiport rules, then indeed the number of membranes may be made smaller.

Theorem 6.2.3. $1NOtP_{3,3}(sym_2, anti_1) = NRE$.

Proof. Let us consider a matrix grammar $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form, in the standard presentation. We add a "dummy" object e to N_2 and each rule $A \to x$ in matrices of type 2 is completed to a rule $A \to xe^i$ such that $|xe^i| = 2$. Without any loss of generality, we may assume that in each derivation in G, the first matrix to be used after the matrix $m_0 : (S \to X_{init}A_{init})$ is a matrix of type 2.

We construct the tissue-like system (of degree 3) with

$$\begin{aligned}
\Pi &= (O, w_1, w_2, w_3, E, R, 1),\\
O &= N_1 \cup N_2 \cup \{\alpha_i, \alpha_i' \mid \alpha \in N_1 \cup \{D\}, 1 \le i \le n\}\\
&\quad \cup \{\alpha_i, \alpha_i', \alpha_i'' \mid \alpha \in N_2 \cup \{e\}, 1 \le i \le n\}\\
&\quad \cup \{a, b, b', c, e, f, f', f'', g, g', D, Z\},\\
w_1 &= X_{init}A_{init}D,\\
w_2 &= bc,\\
w_3 &= b,\\
E &= N_1 \cup N_2 \cup \{\alpha_i, \alpha_i' \mid \alpha \in N_1 \cup \{D\}, 1 \le i \le n\}\\
&\quad \cup \{\alpha_i, \alpha_i', \alpha_i'' \mid \alpha \in N_2 \cup T \cup \{E\}, 1 \le i \le n\}\\
&\quad \cup \{a, b', e, f, f', f'', g, g', Z\},
\end{aligned}$$

with the set R of rules constructed as follows:

1. For each matrix $m_i : (X \to Y, A \to \alpha\beta), 1 \le i \le k$, of type 2, hence with $X, Y \in N_1, A \in N_2, \alpha, \beta \in N_2 \cup \{a, e\}$, we introduce the rules:

$$\begin{aligned}
&(1, XA, 2),\ (0, c\beta_i'', 2),\ (2, X/X_i, 0),\ (2, A/\alpha_i', 0),\ (2, X_ib, 0),\\
&(2, X_i\beta_i'', 3),\ (2, \alpha_i', 3),\ (3, X_ib, 0),\ (3, X_i\alpha_i', 1),\\
&(3, \beta_i'', 1),\ (1, X_i/Y, 0),\ (1, \alpha_i'/\alpha, 0),\ (1, \beta_i''/\beta, 0).
\end{aligned}$$

2. For each matrix $m_i : (X \to Y, A \to \#), k+1 \le i \le n$, of type 3, hence with $X \in N_1, Y \in N_1 \cup \{Z\}, A \in N_2$, we introduce the rules:

$$\begin{aligned}
&(1, DX, 2),\ (1, g'A, 2),\ (2, X/X_i, 0),\ (2, D/D_i, 0),\\
&(2, X_ib, 0),\ (2, X_ig, 3),\ (2, D_if, 3),\ (3, X_i/X_i', 0),\\
&(3, g/g', 0),\ (3, f/f', 0),\ (3, f'/f'', 0),\ (3, X_i'D_i, 1),\ (3, X_i'b, 0),\\
&(3, g', 1),\ (3, f'', 1),\ (1, X_i'/Y, 0),\ (1, D_i/D, 0),\ (1, f''g, 0).
\end{aligned}$$

3. We also consider the following rules:

$$\begin{aligned}
&(1, e, 0),\ (1, ZD, 2),\ (2, c, 0),\ (0, cf, 2),\ (0, cg, 2),\\
&(2, bb', 0),\ (0, bb', 2),\ (2, Zc, 3),\ (2, g'b, 0),
\end{aligned}$$

as well as the rules

$$(1, X, X, 0) \text{ for all } X \in N_1.$$

Each derivation in G can be simulated in Π and the only computations of Π which can halt are those which correctly simulate derivations in G. As long as any object $X \in N_1$ is present in membrane 1, the computation must continue, because of the rule $(1, X/X, 0)$. Also, if the object b ever exits the system, then the computation will continue forever, because of the rules $(2, bb', 0), (0, bb', 2)$.

At each moment, only one object from $N_1 \cup \{Z\}$ is present in the system (at the beginning, in membrane 1). Assume that we start by using a rule of the form $(1, XB, 2)$, for some $X \in N_1, B \in N_2$. At the same time, c exits membrane 2 and brings into membrane 2 any object f, g, or β_l'', for some $\beta \in N_2 \cup \{a, e\}$ and $1 \leq l \leq k$. From membrane 2, the objects X, B exit to the environment and are replaced with some objects X_i, α_j', for $\alpha \in N_2 \cup \{a, e\}$ and $1 \leq i, j \leq k$, with $m_j : (U \to U', B \to \alpha\gamma)$. If the object introduced by c in membrane 2 is β_i'' (hence with the same subscript as X_i, and corresponding to the matrix $m_i : (X \to Y, A \to \alpha\beta)$), then the rule $(2, X_i\beta_i'', 3)$ can be used; otherwise we have to use the rule $(2, X_i b, 0)$ and the computation will never stop. At the same time, α_j' goes to membrane 3.

Now, for membrane 3, we have to use either the rule $(3, X_i\alpha_i', 1)$, in the case when $i = j$ and α corresponds to the matrix m_i, or the rule $(3, X_i b, 0)$, which again leads to an endless computation (the object b together with b' will pass forever through membrane 2). Also β_i'' goes into membrane 1. In the next step, all objects $X_i, \alpha_i', \beta_i''$ exit membrane 1 and are replaced by Y, α, β, respectively, which completes the simulation of the matrix m_i. Each occurrence of the dummy object e is immediately sent out of the system by the rule $(1, e, 0)$.

At any moment, the object X can go from membrane 1 to membrane 2 together with the object D, which will entail the simulation of a matrix $m_i : (X \to Y, A \to \#)$, $k+1 \leq i \leq n$, in the following way. From membrane 2, both X and D exit and are replaced with X_i, D_j, for some $k+1 \leq i, j \leq n$. During the previous computation, the object c had enough time to bring into membrane 2 at least one copy of f and at least one copy of g. By using the rules $(2, X_i g, 3), (2, D_j f, 3)$ we move all these objects to membrane 3 (again, if the rule $(2, X_i g, 3)$ cannot be used, then we have to use $(2, X_i b, 0)$ and the derivation will never stop). All objects X_i, g, and f exit membrane 3, and are replaced by their primed versions. The objects X_i', D_j go into membrane 1 if and only if $i = j$; if this is not the case, then the rule $(3, X_i' b, 0)$ leads to an endless computation. At the same time g' goes into membrane 1, while f' exits membrane 3 and is replaced by f''. In membrane 1 we check now whether or not the object A corresponding to the matrix m_i is present. In the affirmative case, the rule $(1, g'A, 2)$ must be used, and g' will move out of the system the object b from membrane 2, which will lead to a non-halting computation. If no copy of A is present in membrane 1, then the object g' waits; in the next step, the object f'' comes from membrane 3, and together they exit the system. In the meantime, X_i' was replaced by Y and D_i by D,

hence we correctly simulate the matrix m_i and we return to a configuration the same as that we started with.

We can continue in this manner, simulating the use of matrices of types 2 and 3 from G. If we introduce the object Z, which indicates that the derivation is terminal, then this object will go to membrane 2 together with the object D, and from there to membrane 3 together with the object c. In this way, in membrane 1 we only have the occurrences of the object a produced during the computation, and the computation stops, because c is "locked" by membrane 3. Consequently, we have $N(\Pi) = length(L(G))$, and this completes the proof (clearly, the communication is done in a one-way manner, along the cycle $1 \to 2 \to 3 \to 1$). □

It is worth noting that in the previous system we use antiport rules only for the communication with the environment, while the communication between membranes is done only by symport rules.

In the proof of Theorem 6.2.3 we use a system composed of three membranes, with each two communicating directly. It seems to us that systems with only two membranes are not universal. However, it is natural to consider systems with more than two membranes arranged in a loosely structured network, in the sense that the membranes do not necessarily communicate directly, but only through the environment. (This means that the communication graph is not necessarily connected.) It turns out that the collaboration of two "dipoles" again gives the universality.

Theorem 6.2.4. $NOtP_{4,2}(sym_2, anti_1) = NRE$.

Proof. Let us start from a matrix grammar with appearance checking $G = (N, \{a\}, S, M, F)$ in the Z-binary normal form. Without any loss of generality, we may assume that for all matrices $m_i : (X \to Y, A \to \#)$ we have $X \neq Y$ (a matrix $(X \to X, A \to \#)$ changes nothing, hence it can be removed). We construct a tissue-like P system Π with four membranes, labelled by 1, 2, 3, 4, in the following way:

$$
\begin{aligned}
\Pi &= (O, w_1, w_2, w_3, w_4, E, R, 2),\\
O &= N_1 \cup N_2 \cup \{X_i \mid X \in N_1, 1 \leq i \leq n\}\\
&\quad \cup \{\alpha_i', \alpha_i'', \alpha_i''' \mid \alpha \in N_2 \cup \{g, e\}, 1 \leq i \leq n\}\\
&\quad \cup \{c_i \mid 1 \leq i \leq n\} \cup \{d_i, d_i' \mid k+1 \leq i \leq n\} \cup \{a, b, d, d', d'', f\},\\
w_1 &= X_0 A_0 d c_1 c_2 \ldots c_n,\\
w_2 &= b,\\
w_3 &= d_{k+1} \ldots d_n,\\
w_4 &= d_{k+1}' \ldots d_n f,\\
E &= N_1 \cup N_2 \cup \{X_i \mid X \in N_1, 1 \leq i \leq n\}\\
&\quad \cup \{\alpha_i', \alpha_i'', \alpha_i''' \mid \alpha \in N_2 \cup \{g, e\}, 1 \leq i \leq n\}
\end{aligned}
$$

$$
\begin{aligned}
&\cup \{a, d', d''\},\\
R = &\{(1, X/X, 0) \mid X \in N_1\}\\
&\cup \{(1, c_iX, 2),\ (1, dA, 2),\ (2, X/X_i, in),\ (2, A/\alpha_i', 0),\\
&\quad (2, c_i, 0),\ (0, c_i\beta_i'', 2),\ (2, \alpha_i'/\alpha_i''', 0),\\
&\quad (2, c_i\alpha_i''', 1),\ (2, \alpha_i'''b, 1),\ (2, \beta_i''X_i, 1),\ (2, \beta_i''b, 1),\\
&\quad (1, X_i/Y, 0),\ (1, \alpha_i'''/\alpha, 0),\ (1, \beta_i''/\beta, 0) \mid\\
&\quad \text{for all } m_i : (X \to Y, A \to \alpha\beta), 1 \le i \le k,\\
&\quad \text{with } X, Y \in N_1, A \in N_2, \alpha, \beta \in N_2 \cup \{e\}\}\\
&\cup \{(1, c_iX, 0),\ (3, d_i/c_i, 0),\ (3, c_i/f, 4),\ (3, f, 0),\\
&\quad (4, c_id_i', 0),\ (0, d_i, 1),\ (0, f, 1),\ (1, d_iA, 2),\\
&\quad (2, d_ib, 0),\ (0, bb', 2),\ (2, b'/b', 0),\ (1, d_if, 0),\\
&\quad (0, c_id_i', 1),\ (1, d_i'/Y, 0),\ (0, d_if, 4),\ (0, d_i', 4),\\
&\quad (4, d_i, 3),\ (2, b/c_i, 0) \mid\\
&\quad \text{for all } m_i : (X \to Y, A \to \#), k+1 \le i \le n\}\\
&\cup \{(2, d/d', 0),\ (2, d'/d'', 0),\ (2, d'', 1),\\
&\quad (1, b/b, 0),\ (1, Z, 2),\ (2, Zb, 0)\}\\
&\cup \{(1, ZA, 2) \mid A \in N_2\}\\
&\cup \{(1, a, 2)\}.
\end{aligned}
$$

Note that only membranes in the pairs (1, 2) and (3, 4) communicate directly (this time, in both directions for each pair).

This system works as follows.

Let us consider first the simulation of matrices m_i with $1 \le i \le k$. Because we always have only one copy of one object from N_1, only one rule $(1, c_iX, 2)$ can be used, simultaneously with a rule $(1, dA, 2)$, if any object $A \in N_2$ is present in membrane 1. The objects X, c_i, A, d exit immediately from membrane 2; X is replaced by X_h, A is replaced by α_j', d is replaced by d', for some $1 \le h, j \le k$, and $\alpha \in N_2 \cup \{a, e\}$. At the next step, c_i returns together with an object β_i'', for some $\beta \in N_2 \cup \{a, e\}$, while d' is replaced by d'', α_j' is replaced by α_j''', and X_h waits. Now, $c_i\alpha_j'''$ can go into membrane 1, provided that $i = j$, otherwise the rule $(2, \alpha_j'''b, 1)$ can be used and the object b will pass forever through membrane 1 by using the rule $(1, b/b, 0)$. Similarly, β_i'' can go into membrane 1 together with X_h, provided that $i = h$, otherwise the rule $(2, \beta_i''b, 1)$ must be used and the computation never stops. In this way, we check that we have $i = j = h$ and that the objects X, A, α, β correspond to the same matrix $m_i : (X \to Y, A \to \alpha\beta)$. At the same time, d'' is moved into membrane 1.

At the next step, the objects $X_i, \alpha_i''', \beta_i'', d''$ exit from membrane 1 and are replaced by Y, α, β, d, respectively. In this way, the matrix m_i is correctly simulated and we can continue by simulating another matrix, because we

have returned to a configuration that is similar to the one that we started with. At any moment, each occurrence of a is sent to membrane 2.

Consider now the simulation of a matrix m_i, with $k+1 \leq i \leq n$. At any moment, we can use the rule $(1, c_i X, 0)$. From the environment, the object c_i will enter membrane 3, releasing the object d_i into the environment (by the rule $(3, d_i/c_i, 0)$). In the next step, d_i will enter membrane 1 and c_i goes to membrane 4 in exchange for f (the rules used are $(0, d_i, 1), (3, c_i/f, 4)$, respectively). If any copy of the object A is present in membrane 1, then the rule $(1, d_i A, 2)$ must be used, and this will lead to an endless computation: d_i will bring b outside membrane 2, which makes possible the use of the rule $(0, bb', 2)$ and the computation will continue forever by using the rule $(2, b'/b', 0)$. If there is no copy of A in membrane 1, then d_i waits.

After arriving in membrane 4, c_i exits together with d_i'; at the same time, f exits membrane 3. If d_i' again enters membrane 4, then c_i has to enter membrane 2, in exchange for b, which will bring d' inside membrane 2 and the computation will never stop. Thus, in the next step, all three objects c_i, d_i', f must enter membrane 1; then, d_i, f exit together, d_i' exits in exchange for Y (and this completes the simulation of the rule $X \to Y$), and c_i remains in membrane 1. Now, d_i' returns to membrane 4, at the same time as d_i and f which enter membrane 4. Simultaneously, a new object c_j can exit membrane 1, and the process is iterated. If the same c_i again exits membrane 1, then this is not according to a derivation in G, as we do not have $X = Y$ – therefore the computation will never stop. From membrane 4, the object d_i returns to membrane 3, and this completes the simulation of the matrix m_i.

Consequently, we can correctly simulate all matrices $m_i, 1 \leq i \leq n$. When the special object Z is introduced in membrane 1, signalling the end of the derivation in G, then we send this object to membrane 2, by the rule $(1, Z, 2)$, and then we use the rule $(2, Zb, 0)$, which removes the objects Z, b from membrane 2. In this way, we also ensure the correct termination of computations which correspond to terminal derivations in G. Consequently, $N(\Pi) = length(L(G))$, which completes the proof. □

What can we compute with systems of membranes that are completely unstructured, swimming together in the environment, never communicating directly but only through the environment? This is an *open problem* (Problem Q21), but we conjecture that the power of such systems is quite limited (more specifically, we *conjecture* that $NOtP_{*,1}(sym_*, anti_*) \subseteq NREG$). Conversely, it is also not known if all semilinear sets of natural numbers can be generated by a tissue-like P system with no direct communication among membranes. However, such systems with only two membranes, and using only symport rules, can generate the set of all natural numbers: consider the system

$$\Pi = (\{a, c\}, c, \lambda, \{a\}, \{(1, c, 0),\ (0, ca, 1),\ (0, c, 2)\}, 1).$$

The "carrier" c exits membrane 1, returns here with any number of copies of a, and at some time gets locked in membrane 2, thus ending the computation.

6.3 Neural-like Networks of Membranes

We now pass to considering a class of networks of membranes of a different type, mainly inspired by the way the neurons cooperate, processing impulses in the complex net established by synapses (see again Section 2.2).

A possible mathematical model of this symbol processing machinery can be a net of membranes, each of them containing a multiset of objects and also associated with a *state*, according to which the objects are processed. The membranes can communicate along channels ("axons") specified in advance.

We stress the fact that in this section each membrane has a *state* according to which it behaves, and this is a major difference with respect to the networks of membranes from the previous sections of this chapter, and, furthermore, with respect to the cell-like P systems from the previous chapters.

6.3.1 Definitions and Examples

The standard rules we are going to use are of the form $sw \to s'w'$, where s, s' are states and w, w' are multisets of objects. Some of the elements of w' may be marked with the indication "go", and this means that they have to leave the membrane immediately and pass to the membranes to which we have direct links through synapses (in order to stress the neural motivation, the terminology that we use will contain several notions from this area; in particular, the membranes will also be called *cells* and the links among them *synapses*). Bearing in mind the communication of neurons, by chemical and electrical signals, the communication (the transfer of symbol–objects) among the membranes of our systems can be done in a replicative manner (the same symbol is sent to all adjacent membranes), or in a non-replicative manner; in the second case we can send all the symbols to only one neighbouring membrane, or we can distribute them, non-deterministically. This is another very important novel feature of these neural-like systems.

Formally, a *neural-like P system*, of *degree* $m \geq 1$, is a construct

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

where:

1. O is a finite non-empty alphabet (of *objects*);
2. $\sigma_1, \ldots, \sigma_m$ are *cells*, of the form

$$\sigma_i = (Q_i, s_{i,0}, w_{i,0}, R_i), 1 \leq i \leq m,$$

   where:
   a) Q_i is a finite set (of *states*);
   b) $s_{i,0} \in Q_i$ is the *initial state*;
   c) $w_{i,0} \in O^*$ is the *initial multiset* of objects of the cell;

d) R_i is a finite set of *rules* of the form $sw \to s'xy_{go}z_{out}$, where $s, s' \in Q_i$, $w, x \in O^*$, $y_{go} \in (O \times \{go\})^*$, and $z_{out} \in (O \times \{out\})^*$, with the restriction that $z_{out} = \lambda$ for all $i \in \{1, 2, \ldots, m\}$ different from i_o;
e) $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ (*synapses* among cells);
f) $i_o \in \{1, 2, \ldots, m\}$ indicates the *output cell.*

A system as above is said to be *cooperative* if it contains at least one rule $sw \to s'w'$ such that $|w| > 1$, and *non-cooperative* in the opposite case. The objects which appear in the left-hand multiset w of a rule $sw \to s'w'$ are sometimes called *impulses*, while those from w' are also called *excitations.*

Note that we allow rules of the forms $s \to s', s \to s'w'$, although they may be considered as "non-realistic" from the biological point of view, and we also allow synapses of the form $(i, i), 1 \leq i \leq m$, although in the proofs below we do not need such circular synapses.

Any m-tuple of the form $(s_1w_1, \ldots, s_mw_m)$, with $s_i \in Q_i$ and $w_i \in O^*$, for all $1 \leq i \leq m$, is called a *configuration* of Π; thus, $(s_{1,0}w_{1,0}, \ldots, s_{m,0}w_{m,0})$ is the *initial configuration* of Π.

Using the rules from the sets $R_i, 1 \leq i \leq m$, we can define *transitions* among the configurations of the system. To this end, we first consider three *modes of processing the impulse–objects* and three *modes of transmitting excitation–objects* from one cell to another one.

Let us denote $O_{go} = \{(a, go) \mid a \in O\}$, $O_{out} = \{(a, out) \mid a \in O\}$, and $O_{tot} = O \cup O_{go} \cup O_{out}$.

For $s, s' \in Q_i, x \in O^*, y \in O^*_{tot}$, we write

$$
\begin{aligned}
sx \Longrightarrow_{min} s'y \quad &\text{iff} \quad sw \to s'w' \in R_i, w \subseteq x, \text{ and } y = (x - w) \cup w', \\
sx \Longrightarrow_{par} s'y \quad &\text{iff} \quad sw \to s'w' \in R_i, w^k \subseteq x, w^{k+1} \not\subseteq x, \\
&\qquad \text{for some } k \geq 1, \text{ and } y = (x - w^k) \cup w'^k, \\
sx \Longrightarrow_{max} s'y \quad &\text{iff} \quad sw_1 \to s'w'_1, \ldots, sw_k \to s'w'_k \in R_i, k \geq 1, \text{ such that} \\
&\qquad w_1 \ldots w_k \subseteq x, y = (x - w_1 \ldots w_k) \cup w'_1 \ldots w'_k, \\
&\qquad \text{and there is no } sw \to s'w' \in R_i \text{ such that} \\
&\qquad w_1 \ldots w_k w \subseteq x.
\end{aligned}
$$

In the first case, only one occurrence of the multiset from the left-hand side of a rule is processed (replaced by the multiset from the right-hand of the rule, at the same time changing the state of the cell); in the second case a maximal change is performed with respect to a chosen rule, in the sense that as many copies as possible of the multiset from the left-hand side of the rule are replaced by the corresponding number of copies of the multiset from the right-hand side; in the third case a maximal change is performed with respect to all rules which use the current state of the cell and introduce the same new state after processing the objects.

We also write $sx \Longrightarrow_\alpha sx$, for $s \in Q_i, x \in O^*$, and $\alpha \in \{min, par, max\}$, if there is no rule $sw \to s'w'$ in R_i such that $w \subseteq x$. This encodes the case

when a cell cannot process the current objects in a given state (it can be "unblocked" after receiving new impulses from the cells which are active and can send objects to it).

From the biological point of view, the maximal mode is perhaps more realistic, as we may assume that the cells (neurons included) can process chemical objects (impulses) in a rather efficient – hence maximal – manner. Mathematically, the *max* mode corresponds to the assumption that each set R_i of rules is union-closed, in the following sense: if two rules $sw_1 \to s'w'_1$, $sw_2 \to s'w'_2$ are in R_i, then also the rule $s(w_1 \cup w_2) \to s'(w'_1 \cup w'_2)$ is in R_i. This implies that, if R_i contains at least one rule $sw \to s'w'$ with a non-empty $w \cup w'$, then R_i is infinite (but "finitely generated" by the union-closure of a finite set of rules).

Now, recall that the multiset w' from a rule $sw \to s'w'$ contains symbols from O, but also symbols of the form (a, go) (or, in the case of the cell i_o, of the form (a, out)). Such symbols will be sent to the cells related by synapses to the cell σ_i where the rule $sw \to s'w'$ is applied, according to the following three modes:

- *replicative* (indicated by *repl*): each symbol a, for (a, go) appearing in w', is sent to each of the cells σ_j such that $(i, j) \in syn$;
- *unique destination* (indicated by *one*): all symbols a appearing in w' in the form (a, go) are sent to one of the cells σ_j such that $(i, j) \in syn$, non-deterministically chosen; more exactly, in the case of modes *par* and *max* of using the rules, we first perform all applications of rules, and after that we send all obtained symbols to a unique descendant of the cell (that is, we do not treat separately the objects introduced by each rule, but all of them in a package);
- *non-deterministic distribution* (indicated by *spread*): the symbols a appearing in w' in the form (a, go) are non-deterministically distributed among the cells σ_j such that $(i, j) \in syn$.

In the case of modes *repl* and *spread* there is no difference between treating the objects produced by rules in a package, as in the case of the mode *one*, or separately for each rule: for *repl* anyway the symbols are sent to all descendants, while for *spread* the distribution is completely random.

In order to formally define the transition among the configurations of Π we need some further notations. For a multiset w over O_{tot}, we consider the projections on O, O_{go}, and O_{out}, that is $pr_O(w)$, $pr_{O_{go}}$, and $pr_{O_{out}}$ – remember (from Subsection 2.3.12) that they are multisets, too.

Moreover, for a node i in the graph defined by *syn* we denote $anc(i) = \{j \mid (j, i) \in syn\}$ and $succ(i) = \{j \mid (i, j) \in syn\}$ (the ancestors and the successors of node i, respectively).

Now, for two configurations $C_1 = (s_1w_1, \ldots, s_mw_m), C_2 = (s'_1w''_1, \ldots, s'_mw''_m)$ we write $C_1 \Longrightarrow_{\alpha,\beta} C_2$, for $\alpha \in \{min, par, max\}, \beta \in \{repl, one, spread\}$, if there are $w'_1, \ldots, w'_m$ in O^*_{tot} such that

$$s_i w_i \Longrightarrow_\alpha s'_i w'_i,\ 1 \le i \le m,$$

and

- for $\beta = repl$ we have:

$$w''_i = pr_O(w'_i) \cup \bigcup_{j \in anc(i)} pr_{O_{go}}(w'_j);$$

- for $\beta = one$ we have:

$$w''_i = pr_O(w'_i) \cup \bigcup_{j \in I_i} pr_{O_{go}}(w'_j),$$

  where I_i is a subset of $anc(i)$ such that the set $anc(i)$ was partitioned into $I_1, \ldots, I_m$; at this transition, all non-empty multisets of objects of the form $\bigcup_{j \in I_k} pr_{O_{go}}(w'_j)$, $1 \le k \le m$, should be sent to some receiving cells (added to some multisets $w''_l, 1 \le l \le m$);
- for $\beta = spread$ we have:

$$w''_i = pr_O(w'_i) \cup z_i,$$

  where z_i is a submultiset of the multiset $\bigcup_{j \in anc(i)} pr_{O_{go}}(w'_j)$ such that $z_1, \ldots, z_m$ are multisets with the property $\bigcup_{j=1}^{m} z_j = \bigcup_{j \in anc(i)} pr_{O_{go}}(w'_j)$, and such that all $z_1, \ldots, z_m$ are sent to some receiving cells (added to some multisets $w''_l, 1 \le l \le m$).

Obviously, objects are always sent to a cell i only from its ancestors, that is from cells j such that a direct synapse exists from j to i. In the case of the cell σ_{i_o} we also remove from w'_{i_o} all symbols $a \in O$ which appear in w'_{i_o} in the form (a, out).

During any transition, some cells can do nothing: if no rule is applicable to the available multiset of objects in the current state, then a cell waits until new objects are sent to it from its ancestor cells. It is also worth noting that each transition lasts one time unit, and that the work of the net is synchronized: the same clock marks the time for all cells.

A sequence of transitions among configurations of the system Π is called a *computation* of Π. A computation which ends in a configuration where no rule in no cell can be used is called a *halting* computation. (Note that we do not use final states for marking the end of computations, as in finite automata, although such a variant can be considered.) The result of a halting computation is the number of objects sent to the environment from the output cell i_o. We denote by $N_{\alpha,\beta}(\Pi)$, for $\alpha \in \{min, par, max\}, \beta \in \{repl, one, spread\}$, the set of all natural numbers computed in this way by a system Π, in the mode (α, β). The family of all sets $N_{\alpha,\beta}(\Pi)$, computed by all cooperative

neural-like P systems with at most $m \geq 1$ cells, each of them using at most $r \geq 1$ states, is denoted by $NOnP_{m,r}(coo, \alpha, \beta)$; when non-cooperative systems are used, we write $NOnP_{m,r}(ncoo, \alpha, \beta)$ for the corresponding family of sets of numbers. When any of the parameters m, r is not bounded we replace it with $*$. (The letter "n" in front of "P" points to the fact that we deal here with neural–like P systems.)

We have 18 families of the form $NNP_{*,*}(\gamma, \alpha, \beta)$, but, as we will see below, not all of them are different.

Before investigating the power and the properties of neural-like systems, let us examine two **examples**, in order to clarify the definitions and to illustrate the way of working of these devices.

Consider first a simple system:

$$\begin{aligned}
\Pi_1 &= (\{a\}, \sigma_1, \sigma_2, \sigma_3, syn, 1),\\
\sigma_1 &= (\{s\}, s, a, \{sa \to s(a, go),\ sa \to s(a, out)\}),\\
\sigma_2 &= (\{s\}, s, \lambda, \{sa \to s(a, go)\}),\\
\sigma_3 &= (\{s\}, s, \lambda, \{sa \to s(a, go)\}),\\
syn &= \{(1,2), (1,3), (2,1), (3,1)\}.
\end{aligned}$$

A neural-like P system can be graphically represented as in Figure 6.4 with Π_1, with ovals associated with the cells (these ovals will contain the initial state, the initial multiset, and the set of rules, and will be labelled with $1, 2, \ldots, m$), with arrows indicating the synapses, and with an arrow leaving from the output cell.

The reader can easily check that we have

$$\begin{aligned}
&N_{\alpha,repl}(\Pi_1) = \{n \mid n \geq 1\}, \text{ for } \alpha \in \{min, max\},\\
&N_{\alpha,\beta}(\Pi_1) = \{1\}, \text{ for } \alpha \in \{min, par, max\}, \beta \in \{one, spread\},\\
&N_{par,repl}(\Pi_1) = \{2^n \mid n \geq 0\}.
\end{aligned}$$

Indeed, in the non-replicative mode of communication, no further symbol is produced, hence we only generate the number 1. In the replicative case, the symbols produced by the rule $sa \to s(a, go)$ from cell 1 are doubled by communication. When the rules are used in the parallel mode, then all symbols are processed at the same time by the same rule, which means that all symbols present in the system are doubled from one step to the next one, therefore the powers of 2 are obtained. When the rules are used in the minimal mode, the symbols are processed or sent out one by one, hence all natural numbers can be obtained. In the maximal mode, we can send copies of a at the same time to cells 2 and 3, and outside the system, hence again any number of symbols can be sent out.

Let us now consider a system with a more sophisticated functioning:

$$\Pi_2 = (\{a, a', b, b', c, c'\}, \sigma_1, \sigma_2, \{(1,2), (2,1)\}, 1),$$

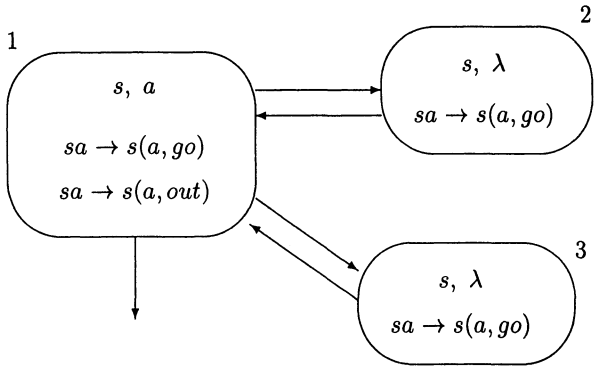


Fig. 6.4. An example of a neural-like P system

$$
\begin{aligned}
\sigma_1 &= (\{s, s', s'', s''', s^{iv}, s^{v}, s^{vi}\}, s, a, R_1),\\
R_1 &= \{sa \to saa(b, go),\\
&\quad sa \to s'aa(b, go)(b, go)(c, go),\\
&\quad s'b \to s'',\\
&\quad s''a \to s''a'(a, out),\\
&\quad s''a \to s'''a'(a, out)(c', go),\\
&\quad s'' \to s'',\\
&\quad s'''a \to s^{vi},\\
&\quad s'''b' \to s^{iv},\\
&\quad s^{iv}a' \to s^{iv}a(a, out),\\
&\quad s^{iv}a' \to s^{v}a(a, out)(c, go),\\
&\quad s^{iv} \to s^{iv},\\
&\quad s^{v}a' \to s^{vi},\\
&\quad s^{v}b \to s'',\\
&\quad s^{vi} \to s^{vi}\},\\
\sigma_2 &= (\{s, s', s''\}, s, \lambda, R_2),\\
R_2 &= \{sc \to s',\\
&\quad s'b \to s(b, go),\\
&\quad sc' \to s'',\\
&\quad s''b \to s(b', go)\}.
\end{aligned}
$$

If the rules are used in the parallel mode, then in the first cell we obtain 2^n copies of a, for some $n \geq 1$, in parallel with sending a number of copies of b to cell 2. After changing the state of σ_1 to s' and sending c to σ_2, the cell σ_1 waits two steps, while σ_2 changes its state from s to s' and then sends

back to σ_1 all copies of b. At the next step, in the first cell s' is changed to s'' (and all copies of b are removed), which at the next step sends out of the system copies of each a. If this is done by using the rule $s''a \to s''a'(a, out)$, then the work of the system is never finished, because of the rule $s'' \to s''$. If we use the rule $s''a \to s'''a'(a, out)(c', go)$, then the computation stops after one more step, when c' is used in cell σ_2 (because no copy of b is present in σ_2, no further rule can be used here, hence the first cell remains "locked" in state s'''). Thus, we obtain

$$N_{par,\beta}(\Pi_2) = \{2^n \mid n \geq 1\},\ \beta \in \{repl, one, spread\}.$$

(Each cell has only one successor, hence the three modes of communication are equivalent.)

Because we do not have rules involving the same pair of states (with the exception of $s''a \to s''a'(a, out), s'' \to s''$, and $s^{iv}a' \to s^{iv}a(a, out), s^{iv} \to s^{iv}$, but the use of rules $s'' \to s''$ and $s^{iv} \to s^{iv}$ changes nothing), the mode *max* coincides with the mode *par*, hence we get the same set of numbers as above also when using the maximal mode.

In the case of the minimal use of rules we have a more interesting behaviour and we obtain

$$N_{min,\beta}(\Pi_2) = \{n^2 \mid n \geq 2\},\ \beta \in \{repl, one, spread\}.$$

Indeed, any computation starts by producing a number $n \geq 2$ or copies of a in the first cell, while sending n copies of b to the second cell, by using the rules $sa \to saa(b, go)$ and $sa \to s'aa(b, go)(b, go)(c, go)$. During this time, the second cell waits in state s. After receiving the symbol c, the second cell changes its state to s' and then returns to s while sending a copy of b back to the first cell. During these two steps, the first cell waits in state s'.

In the presence of b, the first cell changes the state to s'', and in this state it sends out of the system one copy of a for each copy of a it has; at this time, the internal copies of a are primed. The computation can continue forever in state s'', because of the rule $s'' \to s''$, hence eventually the rule $s''a \to s'''a'(a, out)(c', go)$ must be used. If unprimed copies of a are still present, then the trap state s^{vi} is introduced by the rule $s'''a \to s^{vi}$, and the computation will never stop because of the rule $s^{vi} \to s^{vi}$. Thus, we have to introduce the state s''' while changing the last a into a' (and sending c' to the second cell).

In cell σ_2, the symbol c' entails the change of s into s'', which, in turn, sends to σ_1 a copy of b, primed. Now, a process as above takes place in σ_1, but controlled by states s^{iv}, s^v instead of s', s'': in the presence of b', the state s''' is changed to s^{iv}, which sends out a copy of a for each internal a', at the same time removing the prime; the process should stop by using the rule $s^{iv}a' \to s^v a(a, out)(c, go)$, otherwise the rule $s^{iv} \to s^{iv}$ can be used forever. Moreover, when using the rule which introduces the state s^v no primed a should be present, otherwise the trap state s^{vi} is introduced by the rule

$s^v a' \to s^{vi}$, and the computation never stops. The state s^v will return to s'' again in the presence of b.

The process is repeated as long as we have copies of b in the second cell, that is n times. For each b we send out of the system n copies of a, in total n^2. When no b is available, the computation stops, with the first cell in one of the states s''', s^v and the second cell in one of the states s', s''. We have sent outside n times n copies of a, in total n^2 copies. Again, no difference exists between the three modes of communication.

Note the dramatic difference between the results obtained in mode *min* and in mode *par*.

6.3.2 The Computational Power

We start by considering the minimal mode of using the rules, and this turns out to be computationally universal, a fact which makes natural the comparison with (length sets of) Chomsky families, in particular NRE. Then, we will consider the parallel and the maximal modes of using the rules, and this will make necessary the comparison with (length sets of) Lindenmayer families.

If we take into consideration the apparently weak ingredients of neural-like P systems, which can be considered finite state multiset (distributed) automata, then the fact that when using the mode *min* of applying the rules we obtain universality even in the non-cooperative case, is rather unexpected. (The same result holds true also when using cooperative rules, in all modes *min, par, max.*) In proving such results, we try to keep as reduced as possible both the number of cells and the maximal number of states used by the cells.

Theorem 6.3.1. *$NOnP_{2,5}(ncoo, min, \beta) = NRE$, for all $\beta \in \{repl, one, spread\}$.*

Proof. We start by considering a matrix grammar $G = (N, \{a\}, S, M, F)$ in the strong binary normal form, given with the standard notation, and we construct the neural-like system

$$\Pi = (O, \sigma_1, \sigma_2, syn, 2),$$

with the alphabet

$$\begin{aligned} O = {} & N_1 \cup N_2 \cup \{a, H, \#\} \\ & \cup \{X_{i,j} \mid X \in N_1, 1 \le i \le k, 0 \le j \le k\} \\ & \cup \{A_{i,j} \mid A \in N_2, 1 \le i \le k, 0 \le j \le k\}, \end{aligned}$$

and the following cells:

$$
\begin{aligned}
\sigma_1 &= (\{s, s', s_f, s_1, s_2\}, s, X_{init}A_{init}, R_1),\\
R_1 &= \{sX \to s'(X_{i,i}, go),\\
&\qquad s'A \to s(A_{i,i}, go) \mid m_i : (X \to \alpha, A \to x) \in M, 1 \le i \le k\}\\
&\quad \cup \{sX_{i,0} \to sY \mid m_i : (X \to Y, A \to x), 1 \le i \le k\}\\
&\quad \cup \{sX_{i,0} \to s_f \mid m_i : (X \to \lambda, A \to x), 1 \le i \le k\}\\
&\quad \cup \{s_f D \to s_f D \mid D \in N_2 \cup \{\#\}\}\\
&\quad \cup \{sX \to s_j Y(H, go),\\
&\qquad s_j B^{(j)} \to s_f \#,\\
&\qquad s_j H \to s \mid j = 1, 2,\ m_i : (X \to Y, B^{(j)} \to \#), i \in lab_j\}\\
&\quad \cup \{s \to s,\ s' \to s'\},\\
\sigma_2 &= (\{s, s', s''\}, s, \lambda, R_2),\\
R_2 &= \{sX_{i,j} \to s'X_{i,j-1} \mid X \in N_1, 1 \le i \le k, 2 \le j \le k\}\\
&\quad \cup \{s'A_{i,j} \to sA_{i,j-1} \mid A \in N_2, 1 \le i \le k, 2 \le j \le k\}\\
&\quad \cup \{sX_{i,1} \to s''(X_{i,0}, go) \mid X \in N_1, 1 \le i \le k\}\\
&\quad \cup \{s''A_{i,1} \to s(z, go)(y, out) \mid m_i : (X \to \alpha, A \to x), 1 \le i \le k,\\
&\qquad x = zy, z \in N_2^*, y \in a^*\}\\
&\quad \cup \{s''A_{i,j} \to s''\# \mid A \in N_2, 1 \le i \le k, 2 \le j \le k\}\\
&\quad \cup \{s'A_{i,1} \to s''\# \mid A \in N_2, 1 \le i \le k\}\\
&\quad \cup \{s''\# \to s''\#,\ sH \to s(H, go)\};
\end{aligned}
$$

moreover, we have the synapses

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

(We have written (z, go) instead of $(A_1, go)(A_2, go)$, with $A_1, A_2 \in N_2$, such that $z = A_1A_2$, and a similar short form was used for (y, out).)

As long as the states s, s' are present, the first cell can work at least by rules $s \to s, s' \to s'$, hence we have to remove these states. Assume that we use a rule $sX \to s'(X_{i,i}, go)$ and then a rule $s'A \to s(A_{j,j}, go)$, for some $1 \le i, j \le k$. The symbols $X_{i,i}, A_{j,j}$ are sent to the second cell, where, under the control of states s, s' of σ_2, one alternately decreases by one the second components of the subscripts. During this time, the first cell uses the rule $s \to s$ (no symbol from N_1 is present, so no rule of the form $sY \to s'(Y_{j,j}, go)$ can be used). The computation continues in this way until one of the symbols from cell 2 reaches a subscript with the second component being equal to 1.

We have three cases:

Case 1: If $i < j$, then in the second cell we have to use the rule $sX_{i,1} \to s''(X_{i,0}, go)$, and after that $s''A_{j,k} \to s''\#$, for some $k \ge 2$. The computation will never finish, because of the rule $s''\# \to s''\#$ of the second cell.

Case 2: If $i > j$, then in the second cell we will use the rule $s'A_{j,1} \to s''\#$, which again leads to an endless computation, because the rule $s''\# \to s''\#$ can be used forever.

Case 3: If $i = j$, then after using the rule $sX_{i,1} \to s''(X_{i,0}, go)$ we use the rule $s''A_{i,1} \to s(z, go)(y, out)$, which returns the second cell to state s. In cell 1 we can replace $X_{i,0}$ by Y, thus completing the simulation of the matrix $m_i : (X \to Y, A \to x)$, hence the computation can continue. If m_i was a terminal matrix, then we can use the rule $sX_{i,0} \to s_f$, and then the state s_f checks whether or not any non-terminal from N_2 is still present, a case in which the computation will continue forever.

Of course, instead of using these rules we can use the rule $s \to s$, but it changes nothing, hence eventually we have to use the above-mentioned rules.

In this way, we can correctly simulate the matrices of G without appearance checking rules.

Assume now that we start in cell 1 by using a rule $sX \to s_jY(H, go)$, for some $m_i : (X \to Y, B^{(j)} \to \#)$ with $i \in lab_j, j = 1, 2$. At the next step, if the symbol $B^{(j)}$ is present, then the rule $s_jB^{(j)} \to s_f\#$ is used, which introduces the trap symbol $\#$, and the computation continues forever by means of the rule $s_f\# \to s_f\#$. If no copy of $B^{(j)}$ is present, then the first cell waits one step. At this time, the second cell uses the rule $sH \to s(H, go)$ and sends back the object H. In the presence of this object, the first cell returns to state s. In this way, one correctly simulates the matrix m_i with the second rule used in the appearance checking mode.

Consequently, all derivations in G can be simulated in Π and, conversely, all halting computations in Π correspond to terminal derivations in G. One can easily see that $length(L(G)) = N_{min,\beta}(\Pi)$, for all $\beta \in \{repl, one, spread\}$ (because no cell has two successors, the three modes of communication are equivalent). □

From the previous construction we can easily see that if we start from a matrix grammar without appearance checking, then the states s, s', s_f of the first cell are sufficient, hence we have the following result (remember, however, that $NMAT^\lambda = NREG$).

Corollary 6.3.1. *$NMAT^\lambda \subseteq NOnP_{2,3}(ncoo, min, \beta)$, for all $\beta \in \{repl, one, spread\}$.*

At the cost of using two more cells, we can decrease the number of states used.

Theorem 6.3.2. *$NOnP_{4,4}(ncoo, min, \beta) = NRE$, for both $\beta \in \{one, spread\}$.*

Proof. We start again from a matrix grammar $G = (N, \{a\}, S, M, F)$ in the strong binary normal form, and we construct a neural-like system Π as indicated in Figure 6.5 (we do not present this system formally, because the representation in the figure contains all necessary information and can also be examined in an easier way from the point of view of the correctness of its functioning).

As usual, the matrices are of the forms $m_i : (X \to Y, A \to x)$, or $m_i : (X \to \lambda, A \to x)$, or $m_i : (X \to Y, B^{(j)} \to \#)$, $\#$ is a trap symbol, and H is a new symbol; D is again a generic symbol from N_2.

Let us examine the work of the system Π, especially because it has some features different from the system used in the previous proof.

In the first cell, we have to choose to pass to one of the states s' and s'' (as long as we use the rule $s \to s$ nothing is changed). In the presence of s' we send to one of cells 2 and 3 all objects, primed. In the presence of s'' we send to one of cells 2 and 3 all objects, double primed.

The fact that *all* objects are communicated is ensured by the state s''', which in the presence of any object $D \in N_2$ introduces the trap object $\#$, and then the rule $s'''\# \to s'''\#$ can be used forever; note that the state s''' is also introduced after simulating a terminal matrix of G, and if any $D \in N_2$ is present, then the computation will never stop.

Now, if any double primed object arrives in the second cell, then the trap object $\#$ is introduced and the computation will never stop. Similarly, if any primed object arrives in the third cell, then the computation will never stop. Therefore, in the case of starting with s', all objects arrive in cell 2. This cell, in collaboration with cell 4, simulates the matrices of G without appearance checking rules; this is performed in a similar way to the work of the system from the proof of Theorem 6.3.1. When a matrix $m_i, 1 \leq i \leq k$, is correctly simulated, then all objects are sent to cell 1, by means of the state s''' (the fact that all objects are communicated is ensured by the fact that the state s' of σ_4 will transform any remaining D from N_2 in $\#$ and then the rule $s'\# \to s'\#$ will be used forever).

When starting in cell 1 by introducing the state s'', all objects will be sent (double primed) to cell 3, and here we simulate a matrix $m_i : (X \to Y, B^{(j)} \to \#)$. That is, in the presence of state $s_j, j = 1, 2$, all objects D'' are sent back to cell 1, unprimed, with the exception of $B^{(j)}$.

Eventually, the rule $s_j X'' \to s'(Y, go)$ is used, completing the simulation of the matrix (and ensuring that all objects D were sent back to cell 1 and that the object $B^{(j)}$ is not present: otherwise, the rule $s'D'' \to s\#$ is used, for any remaining D, including $D = B^{(j)}$, and the computation will never halt). Thus, the process is iterated.

We conclude that we have $length(L(G)) = N_{min,\beta}(\Pi)$, for all $\beta \in \{one, spread\}$ (the fact that all primed or double primed objects should go to the same cell ensures the fact that even in the *spread* mode we have to work in the *one* mode, otherwise the computation will never stop). □

If we use cooperative rules, then we can further decrease both the number of cells and states. Moreover, we can characterize NRE for all modes *min, par, max* of processing the objects, and this completes the study of the cooperative case.

Theorem 6.3.3. $NOnP_{2,2}(coo, \alpha, \beta) = NRE$, *for all* $\alpha \in \{min, par, max\}$, $\beta \in \{repl, one, spread\}$.

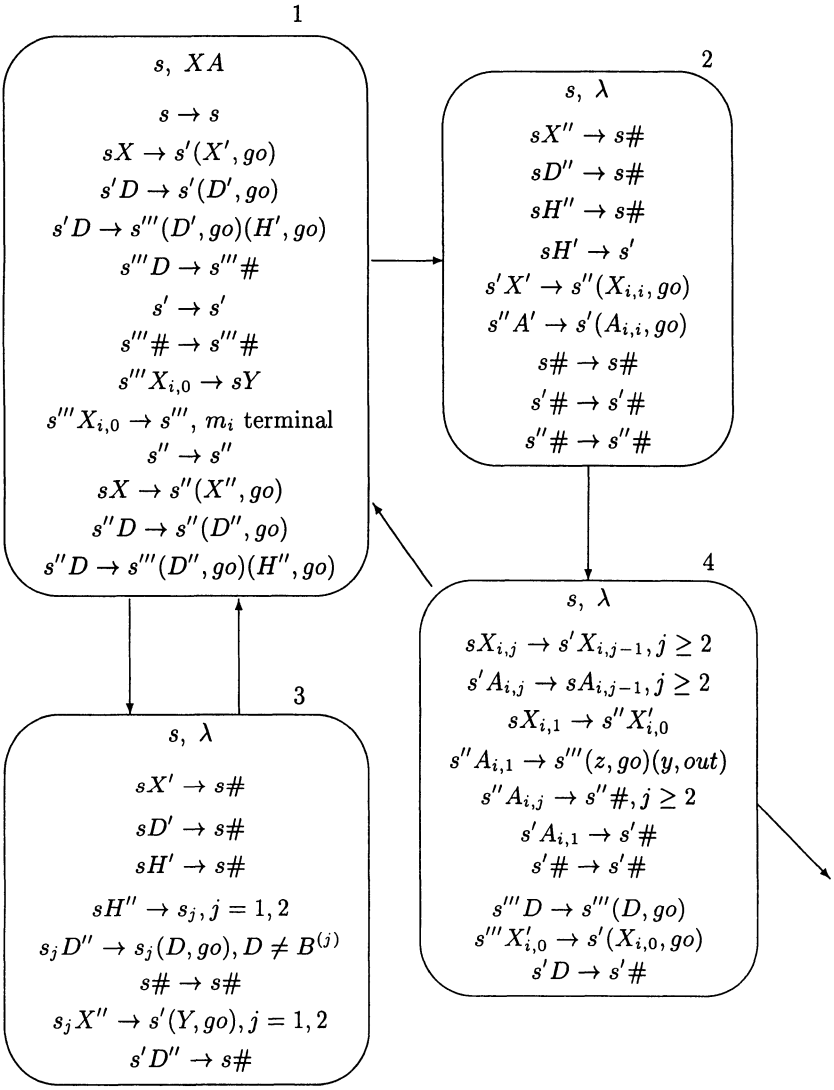


Fig. 6.5. The neural-like P system from the proof of Theorem 6.3.2

Proof. Starting from a matrix grammar $G = (N, \{a\}, S, M, F)$ in the strong binary normal form, given in the standard notation, we construct the tissue-like system

$$\Pi = (N_1 \cup N_2 \cup \{X_1, X_2 \mid X \in N_1\} \cup \{a, c, H, \#\}, \sigma_1, \sigma_2, \{(1,2), (2,1)\}, 1),$$

with the following cells:

$$\begin{aligned}
\sigma_1 &= (\{s, s'\}, s, cX_{init}A_{init}, R_1),\\
R_1 &= \{scXA \to scYz(y, out) \mid (X \to Y, A \to zy) \in M, z \in N_2^*, y \in a^*\}\\
&\quad \cup \{scXA \to sc(y, out) \mid (X \to \lambda, A \to y) \in M, y \in a^*\}\\
&\quad \cup \{scD \to sc\# \mid D \in N_2\}\\
&\quad \cup \{sc\# \to sc\#\}\\
&\quad \cup \{scX \to s'cY_j(H, go),\\
&\qquad s'cY_jB^{(j)} \to sc\#,\\
&\qquad s'cY_jH \to scY \mid (X \to Y, B^{(j)} \to \#) \in M, j = 1, 2\},\\
\sigma_2 &= (\{s\}, s, \lambda, \{sH \to s(H, go)\}).
\end{aligned}$$

With the experience of the previous proofs, the reader can easily check that we have the equality $length(L(G)) = N_{\gamma,\beta}(\Pi)$ for all $\gamma \in \{min, par, max\}$ (all rules involve the symbol c, which appears in only one copy and is never changed, hence even in the modes *par* and *max* the rules of the first cell are actually used in the *min* mode), and for all $\beta \in \{repl, one, spread\}$ (each cell has only one successor, hence the three modes of communication coincide). □

In the case of grammars without appearance checking, the second cell is not necessary and the same assertion is true for the state s', hence we have:

Corollary 6.3.2. *$NMAT^\lambda \subseteq NOnP_{1,1}(coo, \alpha, \beta)$, for all $\alpha \in \{min, par, max\}$, $\beta \in \{repl, one, spread\}$.*

In the case of the parallel mode of using the rules, the inclusion from Corollary 6.3.2 is proper, because we can obtain non-semilinear sets of numbers, even in the non-cooperative case, for systems of degree 1, for all modes of communication. Indeed, for

$$\Pi = (\{a\}, (\{s\}, s, a, \{sa \to saa, sa \to s(a, out)\}), \emptyset, 1),$$

we have $N_{par,\beta}(\Pi) = \{2^n \mid n \geq 0\}$, for all $\beta \in \{repl, one, spread\}$.

We do not know if the results in Theorems 6.3.1, 6.3.2, and 6.3.3 are optimal in the number of cells and states (Problem Q22).

Let us now consider the modes *par* and *max*.

The maximal mode of using the rules in a neural-like P system resembles the parallel mode of rewriting the strings in an L system, and this makes the following results expected (the challenge is only to find the smallest number of necessary cells).

Theorem 6.3.4. (i) $NE0L \subseteq NOnP_{1,2}(ncoo, max, \beta)$, *for all* $\beta \in \{repl, one, spread\}$.
(ii) $NET0L \subseteq NOnP_{1,3}(ncoo, max, \beta)$, *for all* $\beta \in \{repl, one, spread\}$.

Proof. Let $G = (V, T, w, R_1, R_2)$ be an ET0L system in the normal form, hence with only two tables. We construct the neural-like system

$$\Pi = (V, (\{s_1, s_2, s_3\}, s_1, w, R_1'), \emptyset, 1),$$

with the unique cell having the following rules:

$$\begin{aligned}
&s_1 a \to s_1 x, \\
&s_1 a \to s_2 x, \text{ for all } a \to x \in R_1, \\
&s_1 a \to s_3 a, \text{ for all } a \in V, \\
&s_2 a \to s_1 x, \text{ for all } a \to x \in R_2, \\
&s_3 a \to s_3(a, out), \text{ for all } a \in T, \\
&s_3 a \to s_3 a, \text{ for all } a \in V - T.
\end{aligned}$$

In the presence of the state s_1 one simulates the table R_1 (because of the maximal mode of using the rules, all objects are processed, exactly as in a derivation step with respect to G); at any moment, we can switch either to state s_2, which entails the simulation of table R_2, or to state s_3, which determines the end of the computation. In the presence of s_3 no further table is simulated, but the terminal symbols with respect to G are sent out of the system; if the derivation is not terminal, hence objects from $V - T$ are still present in the cell, then the computation continues forever by using rules of the form $s_3 a \to s_3 a$, for $a \in V - T$, hence no output is obtained. Consequently, $length(L(G)) = N_{max,\beta}(\Pi)$, for all $\beta \in \{repl, one, spread\}$, which proves assertion (ii).

If we have only one table, R_1, then the state s_2 is no longer necessary, hence two states suffice, and we get assertion (i). □

For the systems working in the *min* mode, we need additional cells (and states) in order to simulate E0L and ET0L systems.

Theorem 6.3.5. $NE0L \subseteq NOnP_{2,3}(ncoo, min, \beta)$, *for all* $\beta \in \{repl, one, spread\}$.

Proof. Let $G = (V, T, w, R)$ be an E0L system. We construct the neural-like system

$$\begin{aligned}
\Pi &= (V \cup \{F, H, \#\}, \sigma_1, \sigma_2, \{(1,2), (2,1)\}, 1), \\
\sigma_1 &= (\{s_1, s_2, s_3\}, s_1, w, R_1), \\
R_1 &= \{s_1 a \to s_1(x, go), \\
&\qquad s_1 a \to s_2(x, go)(H, go) \mid a \to x \in R\}
\end{aligned}$$

$$
\begin{aligned}
&\cup \{s_2 a \to s_2\# \mid a \in V\} \\
&\cup \{s_2\# \to s_2\#,\ s_1 \to s_1, \\
&\quad s_2 H \to s_1(F, go),\ s_2 H \to s_3\} \\
&\cup \{s_3 a \to s_3(a, out) \mid a \in T\} \\
&\cup \{s_3 a \to s_2\# \mid a \in V - T\}, \\
\sigma_2 &= (\{s, s', s''\}, s, \lambda, R_2), \\
R_2 &= \{sH \to s'\} \\
&\cup \{s'a \to s'(a, go), \\
&\quad s'a \to s''(a, go)(H, go), \\
&\quad s''a \to s\# \mid a \in V\} \\
&\cup \{s''F \to s,\ s\# \to s\#,\ s'\# \to s'\#,\ s''\# \to s''\#\}.
\end{aligned}
$$

By using rules of the form $s_1 a \to s_1(x, go)$, associated with $a \to x \in R$, we simulate a derivation step in G; when all objects are rewritten, we can use the rule $s_1 a \to s_2(x, go)(H, go)$, otherwise the trap-object $\#$ is introduced and the computation will never stop. If s_1 is not replaced by s_2, then the rule $s_1 \to s_1$ can be used forever. The second cell waits in state s as long as the object H is not received from the first cell. In the presence of H, the second cell changes its state to s', which sends back to cell σ_1 all the objects (this is again ensured by the trap object $\#$: if any object $a \in V$ is still present, then the rule $s''a \to s\#$ must be used). Now, in the presence of H, the first cell returns either from state s_2 to state s_1 (at the same time, F is sent to the second cell), and the process is iterated (when F is sent to σ_2, this cell returns to state s, waiting again for the symbol H), or to state s_3, which sends out all terminal symbols; if any object from $V - T$ is still present, then the computation never stops. Thus, we have the equality $length(L(G)) = N_{max,\beta}(\Pi)$, for all $\beta \in \{repl, one, spread\}$. □

In the case of ET0L systems we needed one more cell and one more state (but we do not know if this result can be improved).

Theorem 6.3.6. *$NET0L \subseteq NOnP_{3,4}(ncoo, min, \beta)$, for all $\beta \in \{repl, one, spread\}$.*

Proof. Let $G = (V, T, w, R_1, R_2)$ be an ET0L system in the normal form. We construct the neural-like system

$$\Pi = (O, \sigma_1, \sigma_2, \sigma_3, \{(1,2), (2,1), (1,3), (3,1)\}, 3),$$

with the alphabet

$$O = V \cup \{a', a'' \mid a \in V\} \cup \{F, H, H', H'', \#\},$$

and the following cells (h', h'' are the morphisms which replace each $a \in V$ by a', a'', respectively):

$$
\begin{aligned}
\sigma_1 &= (\{s_1, s_2, s_3\}, s_1, w, R_1'),\\
R_1' &= \{s_1 a \to s_1(h'(x), go),\\
&\qquad s_1 a \to s_2(h'(x), go)(H, go) \mid a \to x \in R_1\}\\
&\quad \cup \{s_2 a \to s_2\# \mid a \in V\}\\
&\quad \cup \{s_2\# \to s_2\#,\ s_1 \to s_1\}\\
&\quad \cup \{s_2 H \to s_1(F, go),\ s_2 H \to s_3\}\\
&\quad \cup \{s_3 a \to s_3(h''(x), go),\\
&\qquad s_3 a \to s_2(h''(x), go)(H, go) \mid a \to x \in R_1\},\\
\sigma_2 &= (\{s, s', s''\}, s, \lambda, R_2'),\\
R_2' &= \{sH' \to s', sH'' \to s\#\}\\
&\quad \cup \{s'a' \to s'(a, go),\\
&\qquad s'a' \to s''(a, go)(H, go),\\
&\qquad s''a' \to s\# \mid a \in V\}\\
&\quad \cup \{s\# \to s\#,\ s'\# \to s'\#,\\
&\qquad s''\# \to s''\#,\ s''F \to s\}\\
&\quad \cup \{sa'' \to s\# \mid a \in V\},\\
\sigma_3 &= (\{s, s', s'', s'''\}, s, \lambda, R_3'),\\
R_3' &= \{sH'' \to s'\}\\
&\quad \cup \{s'a'' \to s'(x, go),\\
&\qquad s'a'' \to s''(x, go)(H, go) \mid a \to x \in R_2\}\\
&\quad \cup \{s''a'' \to s\# \mid a \in V\}\\
&\quad \cup \{s\# \to s\#,\ s'\# \to s'\#,\ s''\# \to s''\#,\\
&\qquad s'''\# \to s'''\#,\ s''F \to s\}\\
&\quad \cup \{sa' \to s\# \mid a \in V\}\\
&\quad \cup \{sH' \to s\#,\ sH'' \to s'''\}\\
&\quad \cup \{s'''a'' \to s'''(a, out) \mid a \in T\}\\
&\quad \cup \{s'''a'' \to s\# \mid a \in V - T\}.
\end{aligned}
$$

The work of this system is a combination of the work of the systems from the proofs of Theorems 6.3.2 and 6.3.5: all the objects are sent either to cell σ_2, primed, or to cell σ_3, double primed, at the same time simulating the use of the table R_1 (if single primed objects arrive in cell σ_2 or double primed objects arrive in cell σ_1, then the computation will never stop). From σ_2 we return the objects to the first cell, so the simulation of R_1 can be iterated. In the third cell we simulate the use of the table R_2.

The trigger for starting the work of each cell (for introducing the states s' in σ_2 and σ_3, and for returning to state s_1 in σ_1) is again the object H (primed for σ_2 and double primed for σ_3). At any time, in cell σ_3 we can pass to the state s''', which determines the end of the simulation of the derivation. If all

objects are from T, then they are sent out of the cell and the computation stops, otherwise the computation continues forever and no output is obtained (after using the rule $s'''a'' \to a\#$, $a \in V - T$, we have to use forever the rule $s\# \to s\# \in R_3'$). Thus, we obtain the equality $length(L(G)) = N_{min,\beta}(\Pi)$, for $\beta \in \{one, spread\}$.

For $\beta = repl$ we remove the rules $sa'' \to s\#, a \in V$, and $sH'' \to s\#$ from σ_2, and the rules $sa' \to s\#, a \in V$, and $sH' \to s$ from σ_3. For the system Π' obtained in this way, we have $length(L(G)) = N_{min,repl}(\Pi')$ (the replication has no effect, as all single primed objects from σ_3 and all double primed objects from σ_2 are ignored; moreover, each cell receives all objects, hence none of them is lost). □

On the other hand, the converse of assertion (ii) from Theorem 6.3.4 is also true, even in the following more general form (and this settles the study of modes *par* and *max*: they do not lead to computational universality, as was the case with the mode *min*).

Theorem 6.3.7. *$NOnP_{*,*}(ncoo, \alpha, \beta) \subseteq NET0L$, for all $\alpha \in \{par, max\}$ and $\beta \in \{repl, one, spread\}$.*

Proof. Consider a neural-like P system $\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, i_o)$, with $\sigma_i = (Q_i, s_{i,0}, w_{i,0}, R_i)$, $1 \le i \le m$.

We construct an ET0L system G with the total alphabet

$$V = \{(s,i) \mid s \in Q_i, 1 \le i \le m\} \cup \{(a,i) \mid a \in E, 1 \le i \le m\} \cup \{\#\} \cup O,$$

the terminal alphabet O, and the axiom and the tables defined as follows.

Consider the morphisms $h_i : (O \cup Q_i)^* \to V^*$ by $h(\alpha) = (\alpha, i)$ for all $\alpha \in Q_i \cup O, 1 \le i \le m$. Then, the axiom of G is

$$w = h_1(s_{1,0}w_{1,0})h_2(s_{2,0}w_{2,0}) \ldots h_m(s_{m,0}w_{m,0}).$$

For each $i = 1, 2, \ldots, m$, and each $s, s' \in Q_i$, we consider the sets of rules of the following form:

$$\begin{aligned} R_i(s,s') = \{&(s,i) \to (s',i)\} \\ &\cup \{(a,i) \to h_i(x)f_\beta(u) \mid sa \to s'xu \in R_i, x \in O^*, u \in O_{go}^*\} \\ &\cup \{(q,i) \to \# \mid q \in Q_i - \{s\}\} \\ &\cup \{(b,i) \to (b,i) \mid \text{ there is no rule } sb \to s'z \text{ in } R_i\}, \end{aligned}$$

where $f_\beta(u), \beta \in \{repl, one, spread\}$, denotes a possible distribution of the symbols from u after the use of the rule $sa \to s'xu$, according to the mode β: if $\beta = repl$, then

$$f_{repl}(u) = h_{j_1}(u) \ldots h_{j_k}(u),$$

where $\{j_1, \ldots, j_k\} = succ(i)$; if $\beta = one$, then

$$f_{one}(u) = h_j(u),$$

where j is one of the descendants of cell i such that j is the same for all rules $(a,i) \to h_i(x)f_\beta(u)$ from $R_i(s,s')$; finally, if $\beta = spread$, then

$$f_{spread}(u) = h_{j_1}(u_1)\dots h_{j_k}(u_k),$$

where $\{j_1,\dots,j_k\} = succ(i)$ and $u_1,\dots,u_k$ is a partition of u. Note that for $\beta = one$ and $\beta = spread$ we have several sets of the form $R_i(s,s')$, one for each possible value of $f_\beta(u)$.

It is clear that such a set of rules, applied in the ET0L manner, corresponds to using the rules from R_i in the *max* manner, with at least one rule $sa \to s'xu$ effectively used.

In order to simulate the case when a cell does not work (no rule is applicable to its current multiset), we consider sets $R_i(s), s \in Q_i$, of rules of the following form:

$$\begin{aligned} R_i(s) = &\{(s,i) \to (s,i)\} \\ &\cup \{(q,i) \to \# \mid q \in Q_i - \{s\}\} \\ &\cup \{(a,i) \to (a,i) \mid a \in O, \text{ and no rule } sa \to s'z \text{ appears in } R_i\} \\ &\cup \{(a,i) \to \# \mid \text{there is a rule } sa \to s'z \text{ in } R_i\}. \end{aligned}$$

When constructing the sets $R_i(s,s')$ as above for $i = i_o$, besides symbols $b \in O$ and $(b, go) \in O_{go}$ in the right-hand side of rules $sa \to s'xu$ which lead to $(a,i) \to h_i(x)f_\beta(u)$, we can also have symbols $(b, out), b \in O$. All such symbols are replaced by b (hence they are terminal symbols with respect to the system G); therefore, in this case the rules from $R_i(s,s')$ are of the form $(a,i) \to h_i(x)f_\beta(u)z$, for $z \in O^*$.

Now, we perform the unions of sets of the form $R_i(s,s')$ or of the form $R_i(s'')$ as above; that is, for each i we choose one set of rules from those of the form $R_i(s,s')$ or of the form $R_i(s'')$, and we put together all their rules. To each of the resulting sets of rules we also add completion rules of the form $c \to c$, for all objects $c \in O \cup \{\#\}$. The obtained sets of rules are tables of the system G. From the way they are constructed, and from the manner of considering pair symbols (a,i) with $a \in O$ and i specifying the cell where a is present, one can see that using such tables exactly corresponds to transitions in the system Π. Conversely, each transition can be simulated by using a suitable table of the ET0L system G.

What remains is to stop the work of G in the same manner as in Π, namely only for halting computations, and in that case to remove all nonterminal symbols still present in the sentential form of G. This can be done by using "terminal" tables constructed as follows. For all m-tuples of states $s_1,\dots,s_m$ with $s_i \in Q_i, 1 \le i \le m$, consider the set

$$R(s_1,\dots,s_m) = \{(s_i,i) \to \lambda \mid 1 \le i \le m\}$$

$$\cup \{(a,i) \to \lambda \mid \text{no rule of the form } s_i a \to s'z \text{ is in } R_i, 1 \le i \le m\}$$
$$\cup \{(a,i) \to \# \mid \text{there are rules } s_i a \to s'z \text{ in } R_i, 1 \le i \le m\}$$
$$\cup \{a \to a \mid a \in O\}.$$

We add all rules of the form $c \to \#$ for all symbols c different from (s_i, i), a, and $(a,i), 1 \le i \le m$, for which we already have rules in the set $R(s_1, \ldots, s_m)$ (therefore, we also have the rule $\# \to \#$).

It is easy to see that such a table can be applied without introducing the trap symbol $\#$, which is never eliminated from the sentential forms of G, only when no rule is applicable in Π.

Consequently, we have the equality $length(L(G)) = N_{max,\beta}(\Pi)$, for all $\beta \in \{repl, one, spread\}$, hence the theorem is proved for the case of *max*.

The changes for the case of *par* are straightforward: in the sets $R_i(s, s')$ we have to consider only one rule of the form $(a,i) \to h_i(x) f_\beta(u)$, for a fixed $a \in O$ (that is, the set of rules is associated not only with s, s', but also with a precise rule $sa \to s'xu$ from R_i); the sets of rules $R_i(s)$ and $R(s_1, \ldots, s_m)$ remain unchanged. The parallel mode of using the rules in an ET0L system ensures the fact that when using such a rule $(a,i) \to y$ we process all occurrences of the symbol (a,i) in the current string. For the system Π' obtained in this way we have the equality $length(L(G)) = N_{par,\beta}(\Pi')$, for all $\beta \in \{repl, one, spread\}$, hence the theorem is also proved for the case of the mode *par*. □

Together with assertion (ii) from Theorem 6.3.4 we get the following characterization of $NET0L$, which precisely describes the power of the mode *max* in the non-cooperative case.

Corollary 6.3.3. $NOnP_{1,1}(ncoo, max, \beta) \subseteq NOnP_{1,2}(ncoo, max, \beta) \subseteq NOnP_{1,3}(ncoo, max, \beta) = NOnP_{m,r}(ncoo, max, \beta) = NOnP_{*,*}(ncoo, max, \beta) = NET0L$, *for all* $m \ge 1$, $r \ge 3$, *and* $\beta \in \{repl, one, spread\}$.

A more precise characterization of families $NOnP_{m,r}(ncoo, par, \beta)$, $\beta \in \{repl, one, spread\}$, remains to be found (but we already know that such systems only generate length sets of ET0L languages).

6.3.3 The Computational Efficiency

The architecture of neural-like P systems and their way of working (especially the fact that in the maximal mode of using the rules we can process all objects which may be processed in such a way that the same next state is obtained, irrespective of which rules are used, and the fact that in the replicative mode one can send the same objects to all successors of a cell) have an intrinsic computational efficiency. More precisely, problems related

to paths in a (directed) graph can be easily solved by such a system, by considering a system with the graph of synapses identical to the graph we deal with, constructing all paths in the graph with certain properties by making use of the maximal mode of applying the rules and of the replicative communication, and checking the existence of a path with a desired property.

We illustrate this power of neural-like systems with the Hamiltonian path problem (HPP) with given start and stop points. The problem asks whether or not in a given directed graph $\gamma = (V, U)$ (where $V = \{a_1, \ldots, a_m\}$ is the set of vertices, and $U \subseteq V \times V$ is the set of edges) there is a path starting in some vertex a_{in}, ending in some vertex a_{out}, and visiting all vertices exactly once. For simplicity, in what follows we assume that $a_{in} = a_1$ and $a_{out} = a_m$.

Having a graph $\gamma = (V, U)$ as above, we construct the system

$$\begin{aligned}
\Pi &= (O, \sigma_1, \ldots, \sigma_m, U, m),\\
O &= \{\langle z; k\rangle \mid z \in V^*, 0 \le |z| \le m, 0 \le k \le m\},\\
\sigma_1 &= (\{s\}, s, \langle\lambda; 0\rangle,\\
&\quad \{s\langle\lambda; 0\rangle \to s(\langle 1; 1\rangle, go)\}),\\
\sigma_i &= (\{s\}, s, \lambda,\\
&\quad \{s\langle z; k\rangle \to s(\langle zi; k+1\rangle, go) \mid z \in V^*,\\
&\quad 1 \le |z| \le m-2, |z|_i = 0, 1 \le k \le m-2\}),\\
&\quad \text{for each } i = 2, 3, \ldots, m-1, \text{ and}\\
\sigma_m &= (\{s\}, s, \lambda,\\
&\quad \{s\langle z; m-1\rangle \to s(\langle zm; m\rangle, out) \mid z \in V^*, |z| = m-1\}).
\end{aligned}$$

It is easy to see that $N_{max,repl}(\Pi) \neq \emptyset$ if and only if HPP has a solution for the graph γ: the paths in γ grow simultaneously in all cells of Π, because of the *max* mode of using the rules (each cell has only one state, hence all rules can be used at the same time). Moreover, the cell σ_m can work only after $m-1$ steps and a symbol is sent out of the system at the step m. Thus, it is enough to watch the system at step m and if any symbol is sent out, then HPP has a solution for γ, otherwise we know that such a solution does not exist. (Note that the symbol sent out describes a Hamiltonian path in γ.) We may say that HPP for γ was solved in m steps (hence in linear time with respect to the size of the graph).

The previous construction does not work for (non-cooperative) systems with the modes *min* and *par* of using the rules. For instance, for the graph indicated in Figure 6.6, one accumulates exponentially many partial paths in nodes i_r, j_r close to node 2, because the ancestors replicate the paths, but from each cell only one path is processed in each step.

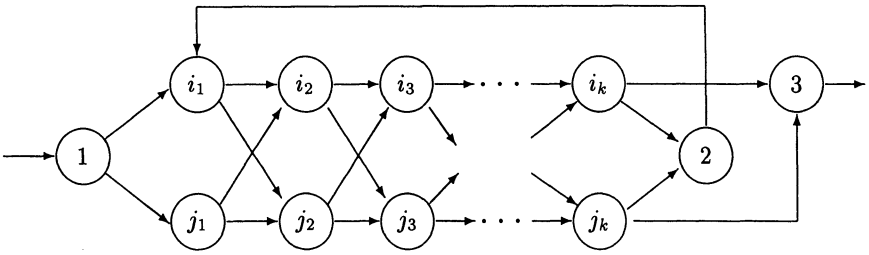


Fig. 6.6. A difficult graph

6.4 Bibliographical Notes

The extension from cell-like systems (hence with the membrane structure described by a tree) to networks of membranes (with the membranes placed in the nodes of an arbitrary graph) was first considered in [197] and in [201]. Splicing P systems of this type were then investigated in [69], where Theorem 6.1.1 can be found.

Section 6.2 is based on [181] (where, however, the result of a computation is the Parikh set of the multiset of objects present in the output membrane in the halting configuration – hence one computes vectors of natural numbers).

Neural-like networks of membranes were introduced and investigated in [157] (also for generating vectors of numbers).

The graphs underlying a membrane structure were considered in a more general topological framework in [75, 76].

7. Trading Space for Time

In Subsection 2.3.11, we mentioned that many practical problems are intractable for normal computers, and that the parallelism is a possible way to cope with this situation. One of the main features of membrane systems is their inherent parallelism. Can this parallelism be used for solving – in theory – hard problems in a feasible time? The answer is positive, and we will illustrate it for various classes of P systems (with enhanced parallelism), which will be shown to be able to solve **NP**-complete problems in polynomial (often, linear) time. In order to have a systematic framework for assessing the quality of such a solution to a given problem, we start by introducing some new complexity classes, which seem to be appropriate to the membrane computing area.

7.1 Complexity Classes for Membrane Systems

For membrane computing the class **NC**, presented at the end of Subsection 2.3.11 as the best known complexity class for parallel computing, is not very adequate – actually, no complexity class mentioned in Subsection 2.3.11 is adequate for membrane computing. The classes **P, NP**, as well other classes, such as **PSPACE, NPSPACE**, deal with sequential computing models, while membrane systems are inherently parallel. **NC** refers to computing models with a structure given in advance, while the membrane systems evolve during the computation; in some sense, the computation essentially means changing the computing device while solving the problem. Moreover, in membrane computing we have possibilities to exponentially increase the workspace, by operations such as cell division, string replication, membrane creation. All these are idealized operations, but they correspond to very precise real phenomena from biology. That is why the definition of **NC** does not cover our case, and that is why we propose new complexity classes here, related to the way the membrane systems work.

Let f be a function from $\mathbf{N}$ to $\mathbf{N}$, acceptable from the complexity theory point of view (see Subsection 2.3.11). Actually, here we will consider only the linear functions and the polynomial functions. Let X be a given class of membrane systems, and A be a decidability problem, with answers "yes" and "no". Denote by $A(n)$ the instance of size n of A.

We say that problem A belongs to $\mathbf{MC}_X(f)$ if a family of membrane systems $\Pi_A = (\Pi_A(1), \Pi_A(2), \ldots)$ of type X exists such that:

1. Π_A is a *uniform* family: there is a Turing machine which constructs $\Pi_A(n)$ in polynomial time starting from $A(n)$. As a direct consequence of the fact that $\Pi_A(n)$ is constructed in polynomial time we have the fact that $\Pi_A(n)$ has a *polynomial size*: the size of $\Pi_A(n)$ (for example, the total number of symbols necessary to encode $\Pi_A(n)$, that is the alphabet, the membrane structure, the initial configuration, and the sets of rules) is bounded by n^k, for some constant k depending on A.
2. Each $\Pi_A(n)$ is *confluent*: there is a constant $t_{A(n)}$ and a distinguished object *yes* such that $\Pi_A(n)$ always halts in less than $t_{A(n)}$ steps, and either it sends out the object *yes* exactly at time $t_{A(n)}$, or this object remains inside the system.
3. $\Pi_A(n)$ is *sound*: that is, it answers the instance of size n of problem A in the following way: if $\Pi_A(n)$ sends out the object *yes*, then the answer to $A(n)$ is "yes"; if $\Pi_A(n)$ halts without sending out the object *yes*, then the answer to $A(n)$ is "no".
4. Π_A is *f-efficient*: that is, $t_{A(n)} \leq f(n)$ for all $n \geq 1$.

Note that we have said nothing about determinism/non-determinism; the computation can be done in any way, non-determinism is allowed. However, because all computations halt (in at most $f(n)$ time units for an input of size n), and because of the confluence, either we send out the object *yes*, or this objects is not sent out, but in either case, because of the soundness, we have an answer to our problem.

Moreover, we have said nothing about the size of any configuration of systems $\Pi_A(n)$ other than the initial one. That is, during the computation, the size of the systems $\Pi_A(n)$ can increase arbitrarily. The increase itself is part of the computation, and it is done "for free" in terms of any resource other than time (the computer can increase exponentially during the computation).

The union of all classes $\mathbf{MC}_X(f)$ for f a linear function is denoted by $\mathbf{LMC}_X$, and the union of all classes $\mathbf{MC}_X(f)$ for f a polynomial is denoted by $\mathbf{PMC}_X$.

The way of defining the classes $\mathbf{LMC}_X$ has some problems: we say that a problem is in such a class if it is solved in *linear* time by a family of membrane systems which are constructed in *polynomial* time starting from the instances of the problem. Therefore, the total time is polynomial. However, we want to emphasize the time spent by the membrane system, and not the time spent in constructing/programming the system; constructing the system can be considered a sort of "pre-computation", so we will ignore this time when assessing the efficiency of (a class of) membrane systems.

Moreover, in the definition of classes $\mathbf{MC}_X(f)$ we have asked that the systems $\Pi_A(n)$ be constructed by a Turing machine, a fact that introduces two types of algorithms/times in the discussion: a sequential time for constructing the systems $\Pi_A(n)$, and a parallel time for computations in $\Pi_A(n)$. What

about also considering the construction of $\Pi_A(n)$ as done by a membrane system? Actually, there appears here a similar question about the reduction of a problem to another one.

For instance, in the following sections we will prove that there are **NP**-complete problems which belong to $\mathbf{LMC}_X$, for various classes X of membrane systems. Typically, the problems considered here are SAT (the satisfiability of propositional formulae given in the conjunctive normal form), and HPP (the existence of a Hamiltonian path in a directed graph). From the previous observation it follows then than $\mathbf{NP} \subseteq \mathbf{LMC}_X$: in a polynomial time, we can pass from any problem A from **NP** to SAT, then in a polynomial time we can construct a family of systems Π which solves SAT in linear time, hence Π also solves A *in linear time*, because the passing from A to SAT is included in the polynomial time for constructing Π.

Admittedly, this is a little like cheating, but again the reduction time is counted for a Turing machine; what about reductions done by means of membrane systems? Are there problems which can be reduced to SAT by means of membrane systems, in linear time? How large is their family? Then, what about the time necessary to construct the reduction systems themselves? The question can be iterated. Anyway, if the reader does not like them, then (s)he may ignore the classes $\mathbf{LMC}_X$, and consider only the classes $\mathbf{PMC}_X$, which are "fair": polynomial reduction, polynomial construction of the membrane system, polynomial time of work in the membrane system.

The theoretical study of the classes $\mathbf{LMC}_X, \mathbf{PMC}_X$, or of other possible similar classes, for various types X of P systems, is beyond the scope of this book, and it remains as a *research topic* (Problem Q23). (An important question: do the classes $\mathbf{LMC}_X$, $\mathbf{PMC}_X$ depend on X, or are they robust from this point of view – at least for the three types X of systems we will consider below?) Here we confine ourselves (mainly) to looking for P systems able to solve SAT and HPP as fast as possible, taking into account only the time spent by the P system itself.

7.2 Using Membrane Division

In Subsection 2.1.3 we mentioned that replication is one of the most important functions of a cell and that, in ideal circumstances, by division (mitosis) we can obtain 2^n cells in n steps, which looks very attractive from a computational point of view. On the other hand, we have seen that most of the reactions which take place in a cell are related to membranes, to various proteins, enzymes, and catalysts which are bound to membranes. In what follows, we will make use of these observations, both by directly involving the membranes in the "reactions", that is in the evoluton rules, and providing rules for membrane division.

By division we obtain two (or more) new membranes, hence we need "names" for them, in the form of labels. This is important also in order

to know which rules apply to the new membranes. Moreover, because the number of membranes can increase exponentially, it is possible to have several membranes with the same label. (The same assertions will be true in the next section, when dealing with membrane creation, starting from objects.) That is why in this chapter we allow the membranes to be labelled with elements from a set given in advance, not necessarily in a one-to-one manner.

In this section we also associate electrical charges with membranes, as usual, from the set $\{+, -, 0\}$, with 0 being the neutral polarization.

With these observations, we can now pass to define a (symbol–object) *P system with active membranes*, in the form of a construct

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

where:

(i) $m \geq 1$ (the initial degree of the system);
(ii) O is the alphabet of *objects*;
(iii) H is a finite set of *labels* for membranes;
(iv) μ is a *membrane structure*, consisting of m membranes, labelled (not necessarily in a one-to-one manner) with elements of H; all membranes in μ are supposed to be neutral;
(v) $w_1, \ldots, w_m$ are strings over O, describing the *multisets of objects* placed in the m regions of μ;
(vi) R is a finite set of *developmental rules*, of the following forms (we also give some explanations about the rules):
  (a) $[_h a \to v]_h^\alpha$,
  for $h \in H, \alpha \in \{+, -, 0\}, a \in O, v \in O^*$
  (object evolution rules, associated with membranes and depending on the label and the charge of the membranes, but not directly involving the membranes, in the sense that the membranes are neither taking part in the application of these rules nor are they modified by them);
  (b) $a[_h\ ]_h^{\alpha_1} \to [_h b]_h^{\alpha_2}$,
  for $h \in H, \alpha_1, \alpha_2 \in \{+, -, 0\}, a, b \in O$
  (communication rules; an object is introduced in the membrane, possibly modified during this process; also the polarization of the membrane can be modified, but not its label);
  (c) $[_h a\ ]_h^{\alpha_1} \to [_h\ ]_h^{\alpha_2} b$,
  for $h \in H, \alpha_1, \alpha_2 \in \{+, -, 0\}, a, b \in O$
  (communication rules; an object is sent out of the membrane, possibly modified during this process; also the polarization of the membrane can be modified, but not its label);
  (d) $[_h a\ ]_h^\alpha \to b$,
  for $h \in H, \alpha \in \{+, -, 0\}, a, b \in O$
  (dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);

(e) $[_h a\,]_h^{\alpha_1} \to [_h b\,]_h^{\alpha_2} [_h c\,]_h^{\alpha_3}$,
for $h \in H, \alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}, a, b, c \in O$
(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with the same label, possibly of different polarizations; the object specified in the rule is replaced in the two new membranes by possibly new objects; the operation is illustrated in Figure 7.1);

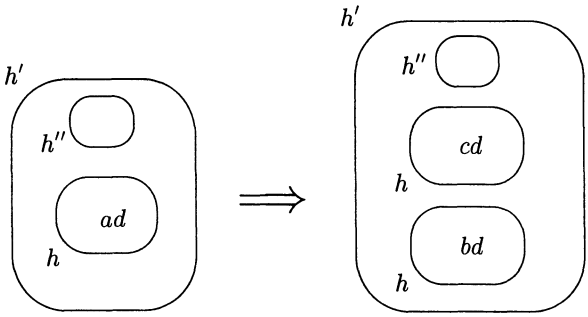


Fig. 7.1. Division of elementary membranes

(f) $[_{h_0}[_{h_1}\,]_{h_1}^{\alpha_1} \cdots [_{h_k}\,]_{h_k}^{\alpha_1}\,[_{h_{k+1}}\,]_{h_{k+1}}^{\alpha_2} \cdots [_{h_n}\,]_{h_n}^{\alpha_2}\,]_{h_0}^{\alpha_0}$
$\to [_{h_0}[_{h_1}\,]_{h_1}^{\alpha_3} \cdots [_{h_k}\,]_{h_k}^{\alpha_3}]_{h_0}^{\alpha_5}\,[_{h_0}[_{h_{k+1}}\,]_{h_{k+1}}^{\alpha_4} \cdots [_{h_n}\,]_{h_n}^{\alpha_4}]_{h_0}^{\alpha_6}$,
for $k \geq 1, n > k, h_i \in H, 0 \leq i \leq n$, and $\alpha_0, \ldots, \alpha_6 \in \{+, -, 0\}$ with $\{\alpha_1, \alpha_2\} = \{+, -\}$; if the membrane with the label h_0 contains other membranes than those with the labels $h_1, \ldots, h_n$ specified above, then they must have neutral charges in order to make this rule applicable; these membranes are duplicated and then are part of the contents of both new copies of the membrane h_0
(division of non-elementary membranes; this is possible only if a membrane contains two immediately lower membranes of opposite polarization, + and −; the membranes of opposite polarizations are separated in the two new membranes, but their polarization can change; Figure 7.2 illustrates this operation).

Note that in all rules of types (a)–(e) only one object is specified (that is, the objects do not directly interact) and that, with the exception of rules of type (a), single objects are always transformed into single objects (the two objects produced by a division rule of type (e) are placed in two different regions).

These rules are applied according to the following *principles*:

1. All rules are applied in parallel: in one step, the rules of type (a) are applied to all objects to which they can be applied, and all other rules

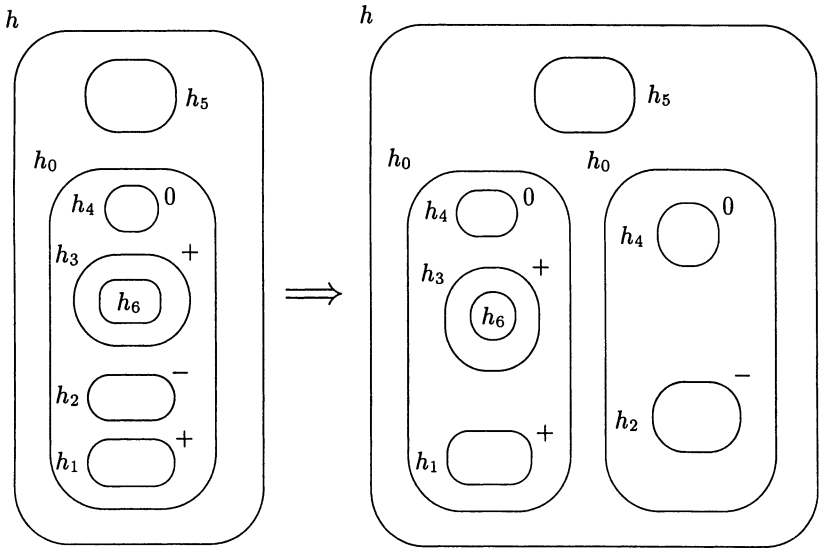


Fig. 7.2. Division of non-elementary membranes

are applied to all membranes to which they can be applied; an object can be used by only one rule, non-deterministically chosen (there is no priority relation among rules), but any object which can evolve by a rule of any form, should evolve.

2. If a membrane is dissolved, then all the objects from its region are left free in the surrounding region. The skin membrane is never dissolved.
3. All objects and membranes not specified in a rule and which do not evolve are passed unchanged to the next step. For instance, if a membrane with the label h is divided by a rule of type (e) which involves an object a, then all other objects from membrane h which do not evolve are introduced in each of the two resulting membranes h (this is the case of object d in Figure 7.1). Similarly, when dividing a membrane h_0 by means of a rule of type (f), the neutral membranes are reproduced in each of the two new membranes with the label h_0, unchanged if no rule is applied to them (in particular, the contents of these neutral membranes are reproduced unchanged in these copies, providing that no rule is applied to their objects) – this is the case of membrane h_4 in Figure 7.2.
4. If at the same time a membrane h is divided by a rule of type (e) and there are objects in this membrane which evolve by means of rules of type (a), then in the new copies of the membrane we introduce the result of the evolution; that is, we suppose that first the evolution rules of type (a) are used, changing the objects, and then the division is produced, so that in the two new membranes with label h we introduce copies of the changed objects. Of course, this process takes only one step. The same

assertions apply to the division by means of a rule of type (f): we always assume that the rules are applied "from bottom-up", in one step, but first the rules of the innermost region and then level by level until the region of the skin membrane.

5. The rules associated with a membrane h are used for all copies of this membrane, irrespective of whether or not the membrane is an initial one or it is obtained by division. At one step, a membrane h can be the subject of only one rule of types (b)–(f).
6. The skin membrane can never divide. As any other membrane, the skin membrane can be "electrically charged".

The membrane structure of the system at a given time, together with all multisets of objects associated with the regions of this membrane structure, is the *configuration* of the system at that time. The $(m+1)$-tuple $(\mu, w_1, \ldots, w_m)$ is the *initial configuration.* We can pass from one configuration to another one by using the rules from R according to the principles given above. We say that we have a (direct) *transition* among configurations. In the subsections which follow we will consider in detail two significant examples, of SAT and HPP, which will both prove the computational usefulness of membrane division and enlighten the previous definition.

A sequence of transitions which starts from the initial configuration is called a *computation* with respect to Π. A computation is *halting* if it cannot be continued: there is no rule which can be applied to objects and membranes in the last configuration.

During a computation, objects can leave the skin membrane (by means of rules of type (c)). The result of a halting computation is the number of objects which are sent out of the system during the computation. A non-halting computation provides no output. The set of all natural numbers computed in this way by a system Π is denoted by $N(\Pi)$.

Note that during a computation the number of membranes (hence the degree of the system) can increase and decrease but the labels of these membranes are always among the labels of membranes present in the initial configuration, because by division we only produce membranes with the same label as the label of the divided membrane. This restriction can be relaxed, allowing division rules of the form:

(e′) $[_{h_1} a\ ]^{\alpha_1}_{h_1} \rightarrow [_{h_2} b\ ]^{\alpha_2}_{h_2} [_{h_3} c\ ]^{\alpha_3}_{h_3}$,
for $h_1, h_2, h_3 \in H, \alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}, a, b, c \in O$, where h_1 is an elementary membrane; in reaction with an object, the membrane is divided into two membranes not necessarily with the same label; also the polarizations of the new membranes can be different from the polarization of the initial one; the object specified in the rule is replaced in the two new membranes by possibly new objects.

Other generalizations of rules are also possible. For instance, we can allow that non-elementary membranes are divided under the influence of objects, by rules of the following form:

(e″) $[_h a\ ]_h^{\alpha_1} \to [_h b\ ]_h^{\alpha_2} [_h c\ ]_h^{\alpha_3}$,
for $h \in H, \alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}, a, b, c \in O$, where h is not necessarily an elementary membrane; all objects and all membranes from the initial membranes h will be replicated and will appear in the two new copies of this membrane – with the exception of object a, which may be replaced by different objects in the two new membranes.

The previous two extensions can be coupled, allowing the division of non-elementary membranes under the influence of an object, with the two new membranes allowed to have new labels.

Another generalization which is natural from a mathematical point of view – it also corresponds to the multiplication of cells by budding – is to divide a membrane not necessarily into two membranes, but into a larger number, bounded by a constant d given in advance. Clearly, the contents of the former membrane are also replicated in d copies and placed inside the d new membranes. Systems using such a general form of division rules are said to have d-division (hence the basic case is called 2-division).

Membrane division is mainly introduced in order to enhance the parallelism of P systems in such a way as to essentially speedup their computations. Still, from a mathematical point of view we have also to ask how powerful such systems are in comparison with Turing machines. As we will immediately see, in order to obtain computational universality we do not need rules of all six forms: membrane division is not needed. Actually, membrane dissolving is also not needed. The "explanation" of this result lies in the fact that the membranes are part of rules; on the one hand, this makes available the membrane polarizations, on the other hand, the membranes can act as catalysts, in fact, as sorts of tri-stable catalysts, because of their polarizations; from Chapter 3 we know that bi-stable – hence furthermore tri-stable – catalysts are very powerful.

Let us denote by $NOP_m(activ, (a, b, c))$ the family of sets $N(\Pi)$, computed by P systems with active membranes of degree at most $m \geq 1$, using rules of types (a), (b), and (c). (Note that in the case of systems which do not use membrane division rules, the degree is equal to the number of membranes in the initial configuration; when using membrane division, we have to consider the maximal degree reached during all possible computations.)

Theorem 7.2.1. $NOP_3(activ, (a, b, c)) = NRE$.

Proof. Let $L \subseteq a^*$ be a recursively enumerable language and let us consider the language $L' = \{a^{n-1} \mid a^n \in L\}$. This is also a recursively enumerable language. Let us consider a matrix grammar with appearance checking, $G = (N, \{a\}, S, M, F)$, in the strong binary normal form, given in the standard

notation, such that $L(G) = L'$. Each matrix of the form $(X \to \lambda,\ A \to x)$, $X \in N_1, A \in N_2, x \in T^*$, is replaced by $(X \to f, A \to x)$, where f is a new symbol. Also, we replace each rule $A \to x$ appearing in matrices of types 2 and 4 of G and having $|x| < 2$ with $A \to xd^j, j = 1, 2$, such that $|xd^j| = 2$.

We construct the system with active membranes Π as follows:

$$
\begin{aligned}
\Pi &= (O, H, \mu, w_1, w_2, w_3, R),\\
O &= N_1 \cup N_2 \cup \{X', X'', X''', X_i \mid X \in N_1, 1 \le i \le k\}\\
&\quad \cup \{A_i, A_1' \mid A \in N_2, 1 \le i \le k\} \cup \{\bar{\alpha} \mid \alpha \in N_2 \cup \{a, d\}\}\\
&\quad \cup \{X_i^{(1)}, X_i^{(2)} \mid X \in N_1, k+1 \le i \le n\} \cup \{a, d, f, f'\},\\
H &= \{1, 2, 3\},\\
\mu &= [_1[_2\ ]_2[_3\ ]_3]_1,\\
w_1 &= X_{init}A_{init},\\
w_2 &= w_3 = \emptyset,
\end{aligned}
$$

and the set R contains the following rules:

For each matrix $m_i : (X \to Y, A \to \alpha_1\alpha_2)$ with $Y \in N_1 \cup \{f\}$, $\alpha_1, \alpha_2 \in N_2 \cup \{a, d\}, 1 \le i \le k$, we introduce the rules:

$X[_2\ ]_2^0 \to [_2X_1]_2^-$,
$[_2X_1 \to X_1]_2^-$,
$A[_2\ ]_2^- \to [_2A_1']_2^+$,
$[_2X_1 \to X_2]_2^+$,
$[_2A_1' \to A_1]_2^+$,
$[_2X_j \to X_{j+1}]_2^+,\ 2 \le j \le k-1$,
$[_2A_j \to A_{j+1}]_2^+,\ 1 \le j \le k-1$,
$[_2X_j]_2^+ \to [_2\ ]_2^0Y',\ 1 \le j \le k$,
$[_2A_j \to \bar{\alpha}_1\bar{\alpha}_2]_2^0,\ 1 \le j \le k$,
$[_2\bar{\alpha}_j]_2^0 \to [_2\ ]_2^0\alpha_j,\ j = 1, 2$,
$[_2A_j \to \#]_2^0,\ 1 \le j \le k$,
$[_2\# \to \#]_2^p,\ p \in \{+, -, 0\}$,
$[_1\# \to \#]_1^p,\ p \in \{+, -, 0\}$,
$[_1Y' \to Y'']_1^0$,
$[_1Y'' \to Y''']_1^0$,
$[_1Y''' \to Y]_1^0$,
$[_1f']_1^0 \to [_1\ ]_1^+a$,
$[_1a]_1^p \to [_1\ ]_1^pa,\ p \in \{+, 0\}$,
$[_1A \to \#]_1^+$,
$[_2X_k \to \#]_2^+$.

For each matrix $m_i' : (X \to Y, B^{(1)} \to \#), k+1 \le i \le n$, we introduce the rules:

$X[_3\ ]_3^0 \to [_3 X_i^{(1)}]_3^-$,
$B^{(1)}[_3\ ]_3^- \to [_3\#]_3^0$,
$[_3\# \to \#]_3^0$,
$[_3 X_i^{(1)} \to Y]_3^-$,
$[_3 Y]_3^- \to [_3\ ]_3^0 Y$.

For each matrix $m_i' : (X \to Y, B^{(2)} \to \#), k+1 \le i \le n$, of type 3, we introduce the rules:

$X[_3\ ]_3^0 \to [_3 X_i^{(2)}]_3^+$,
$B^{(2)}[_3\ ]_3^+ \to [_3\#]_3^0$,
$[_3\# \to \#]_3^0$,
$[_3 X_i^{(2)} \to Y]_3^+$,
$[_3 Y]_3^+ \to [_3\ ]_3^0 Y$.

This system works as follows.

Assume that at a given moment in membrane 1 we have a multiset Xw, for $X \in N_1$ and $w \in (N_2 \cup \{a\})^*$. Initially, we have $X_{init}A_{init}$.

As long as objects X from N_1 are present, the computation is not finished, hence we have to remove all non-terminals of N_1 from membrane 1; this is done by simulating a derivation in G, in the following way.

To simulate a matrix m_i of type 2 or 4, the unique copy of X from the skin region enters membrane 2, by using the rule $X[_2\ ]_2^0 \to [_2 X_1]_2^-$. In this way, membrane 2 gets a negative charge. The rule $[_2 X_1 \to X_1]_2^-$ can be used forever. The way to avoid this is to send also an object A into membrane 2, and this can be done by using the rule $A[_2\ ]_2^- \to [_2 A_1']_2^+$, which changes the polarity of membrane 2 to positive. This allows both X_1 and A_1' to change into X_2 and A_1, respectively. Now the subscripts of both X_i and A_{i-1} are incremented step by step without changing the polarity of the membrane. At any time we can replace X_i with Y' and send it out of membrane 2. When the object Y' comes out, the polarity of membrane 2 is reset to 0, and this allows the use of the rule $[_2 A_i \to \alpha_1\alpha_2]_2^0$. This rule is used simultaneously with $[_1 Y' \to Y'']_1^0$. In the next two steps, the objects α_1, α_2 exit membrane 2, and, simultaneously, Y'' becomes Y''' and then Y. In this way we complete the simulation of the matrix m_i.

Note that both X and A have the same subscript and that the rule $[_2 X_i]_2^+ \to [_2\ ]_2^0 Y'$ is present only for i corresponding to matrices m_i with $X \to Y$ being its first rule. If instead of the rule $[_2 X_i]_2^+ \to [_2\ ]_2^0 Y'$ we use the rule $[_2 X_i \to X_{i+1}]_2^+$ and we continue to increase the subscript of X, then we have two cases. If we reach X_k, then the trap object $\#$ is introduced and the computation never stops. If there is another matrix $m_j, j > i$, with the same first rule $X \to Y$, then we can use the rule $[_2 X_j]_2^+ \to [_2\ ]_2^0 Y'$. If the matrix m_j is of the form $m_j : (X \to Y, A \to \beta_1\beta_2)$, then at the next step we can also use the rule $[_2 A_j \to \bar{\beta}_1\bar{\beta}_2]_2^0$, which is correct with respect to G. If the matrix m_j is of the form $m_j : (X \to Y, B \to u)$, with $B \ne A$, hence we do not have

B_j in membrane 2 but A_j, then we have to use the rule $[_2 A_j \to \#]_2^0$ and the computation will never stop. Therefore, a matrix of type 2 or 4 is correctly simulated.

If the matrix was a terminal one, then in the skin membrane we get the symbol f'. This symbol sends out of the system a copy of a, and also changes the polarity of the skin membrane. If the skin membrane with a positive charge contains any symbol A from N_2, then the computation never halts (due to the introduction of the object $\#$). All copies of a are sent out of the system by using the rule $[_1 a]_1^p \to [_1\ ]_1^p a, p \in \{+, 0\}$.

Consider now a matrix $m_i : (X \to Y, B^{(j)} \to \#)$ of type 3; assume that we have $j = 1$. We start simulating it by using the rule $X[_3\ ]_3^0 \to [_3 X_i^{(1)}]_3^-$, instead of the rule $X[_2\ ]_2^0 \to [_2 X_1]_2^-$. Thus, we send $X_i^{(1)}$ to membrane 3 and the membrane gets a negative charge. Without changing the polarity of this membrane, $X_i^{(1)}$ is replaced with Y. At the same time, if any copy of $B^{(1)}$ is present in the skin membrane, then the trap object $\#$ is introduced into membrane 1 and also the polarity of membrane 3 will be changed to neutral. This will make possible the use of the rule $[_3 \# \to \#]_3^0$, and the computation will never stop. If the object $B^{(1)}$ is not present, then the object Y is sent out of membrane 3, and the polarity of this membrane returns to neutral. The simulation of matrix m_i is complete.

In the same way one simulates matrices $m_i : (X \to Y, B^{(2)} \to \#)$, with the difference that this time membrane 3 becomes positively polarized and the object $X_i^{(1)}$ is replaced with $X_i^{(2)}$.

These operations can be iterated, hence any derivation in G can be simulated by a computation in Π and, conversely, the halting computations in Π correspond to correct derivations in G. Thus, $N(\Pi) = length(L(G))$. □

7.2.1 Solving SAT in Linear Time

The SAT problem (satisfiability of propositional formulae in the conjunctive normal form) is probably the best known **NP**-complete problem. It asks whether or not for a given formula in the conjunctive normal form there is a truth–assignment of variables such that the formula assumes the value *true*. A propositional formula C is in the conjunctive normal form if it is a conjunction of disjunctions, that is

$$C = C_1 \wedge C_2 \wedge \ldots \wedge C_m,$$

where each $C_i, 1 \leq i \leq m$, is a *clause* of the form

$$C_i = y_{i,1} \vee y_{i,2} \vee \ldots \vee y_{i,p_i},$$

with each literal y_j being either a propositional variable, x_s, or its negation, $\neg x_s$. (Thus, we use the variables $x_1, x_2, \ldots$ and the three connectives $\vee, \wedge, \neg$: *or*, *and*, *negation*.)

For example, let us consider the propositional formula

$$C = (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2).$$

We have two variables, x_1, x_2, and two clauses. It is easy to see that it is satisfiable: any of the following truth–assignments makes the formula *true*:

$$(x_1 = \mathit{true}, x_2 = \mathit{false}), \quad (x_1 = \mathit{false}, x_2 = \mathit{true}).$$

Theorem 7.2.2. *The* SAT *problem can be solved in linear time with respect to the number of variables and the number of clauses by a P system with active membranes using rules of the forms* $(a), (b), (c), (e)$ *(with 2-division only).*

Proof. Let us consider a propositional formula $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, with $C_i = y_{i,1} \vee \ldots \vee y_{i,p_i}$, for some $m \geq 1, p_i \geq 1$, and $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}$, for each $1 \leq i \leq m, 1 \leq j \leq p_i$.

We construct the system

$$\Pi = (O, H, \mu, w_1, w_2, R),$$

where

$$\begin{aligned} O &= \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 0 \leq i \leq m\} \\ &\quad \cup \{c_i \mid 1 \leq i \leq m+1\} \cup \{d_i \mid 0 \leq i \leq n\} \cup \{yes\}, \\ H &= \{1, 2\}, \\ \mu &= [_1[_2\ ]_2^0]_1^0, \\ w_1 &= \lambda, \\ w_2 &= a_1 a_2 \ldots a_n d_0, \end{aligned}$$

and the set R contains the following rules (we also give explanations about the role of these rules in the computation meant to solve SAT):

1. $[_2 a_i]_2^0 \to [_2 t_i]_2^0 [_2 f_i]_2^0$, $1 \leq i \leq n$.
   The objects a_i correspond to variables $x_i, 1 \leq i \leq n$. By using a rule as above, for i non-deterministically chosen, we produce the truth values *true* and *false* assigned to variable x_i, placed in two separate copies of membrane 2. Note that the charge remains the same for both membranes, namely neutral, hence the process can continue. In this way, in n steps we assign truth values to all variables, hence we get all 2^n truth–assignments, placed in 2^n separate copies of membrane 2. In turn, these 2^n copies of membrane 2 are placed in membrane 1 – the system always has only two levels of membranes. Note that in spite of the fact that in each step the object a_i is non-deterministically chosen, after n steps we get the same result, irrespective of which objects were used in each step.
2. $[_2 d_k \to d_{k+1}]_2^0$, $0 \leq k \leq n-2$.

3. $[_2 d_{n-1} \to d_n c_1]_2^0$.
4. $[_2 d_n]_2^0 \to [_2\ ]_2^+ d_n$.
   The objects d_i are counters. Initially, d_0 is placed in membrane 2. By division (when using rules of type 1), we introduce copies of the counter in each new membrane. In each step, in each membrane with label 2, we pass from d_k to d_{k+1}, thus "counting to n". In step n we introduce both d_n and c_1; the former objects will exit the membrane (at step $n+1$), changing its polarization to positive, the latter one will be used at the subsequent steps as shown below. Note that at step n we have also completed the generation of all truth–assignments.
5. $[_2 t_i \to r_{h_{i,1}} \dots r_{h_{i,j_i}}]_2^+$, $1 \le i \le n$, and the clauses $C_{h_{i,1}}, \dots, C_{h_{i,j_i}}$ contain the literal x_i.
6. $[_2 f_i \to r_{h_{i,1}} \dots r_{h_{i,j_i}}]_2^+$, $1 \le i \le n$, and the clauses $C_{h_{i,1}}, \dots, C_{h_{i,j_i}}$ contain the literal $\neg x_i$.
   In one step, in parallel in all 2^n membranes with label 2 (and polarization +), we look for the clauses satisfied by the truth–assignments from each membrane. This is also done in parallel for all truth values present in each membrane (the rules of type 5, 6 are used in parallel, they do not actively involve the membranes). If, after completing this step, there is at least one membrane with label 2 which contains all symbols $r_1, r_2, \dots, r_m$, this means that the truth–assignment from that membrane satisfies all clauses, hence it satisfies formula C. Otherwise (if in no membrane we get all objects $r_1, r_2, \dots, r_m$), the formula C is not satisfiable. Note that the satisfiability was already solved, in $n+2$ steps, making essential use of the parallelism, but we still have to "read" the answer, and send a suitable signal out of the system. This will be done by the following rules.
7. $[_2 r_1]_2^+ \to [_2\ ]_2^- r_1$.
   At the same time for all 2^n membranes with label 2, we check whether or not r_1 is present in each membrane. If this is the case, then r_1 is sent out of the membrane where it is present, changing in this way the polarization of that membrane, to negative. The membranes which do not contain the object r_1 remain positively charged and they will no longer evolve, as no further rule can be applied to them.
8. $[_2 c_i \to c_{i+1}]_2^-$, $1 \le i \le m$.
9. $[_2 r_k \to r_{k-1}]_2^-$, $2 \le k \le m$.
10. $r_1 [_2\ ]_2^- \to [_2 r_0]_2^+$.
    In the copies of membrane 2 with a negative polarization, hence only those where we have found r_1 in the previous step, we perform two operations in parallel: we count the number of satisfied clauses, by means of increasing the subscript of c_k, and we decrease the subscripts of all objects r_j from that membrane. Thus, if the second clause was satisfied by the truth–assignment from a membrane, that is r_2 was present in a membrane, then r_2 becomes r_1 – hence the rule of type 7 can be applied again. Note the important fact that passing from r_2 to r_1 is possible

only in membranes where we already had r_1, hence we check whether the second clause is satisfied only after knowing that the first clause was satisfied. At the same time, the object r_1 from the skin membrane returns to membranes with label 2, changed to r_0 (which will never evolve again), and returning the polarization of the membrane to positive (this makes possible the use of rules of type 7).

Note also another important fact: we have in the skin membrane a number of copies of r_1 equal to the number of membranes with negative charge; a rule of type 7 is of general type (b), which involves the membrane; thus, because of this fact and because of the parallelism, each membrane which previously contained an object r_1 will now contain an object r_0.

The rules of types 7, 8, 9, 10 are applied as many times as possible (in one step rules of type 7, in the next one rules of types 8, 9, and 10, and then we repeat the cycle). Clearly, if a membrane does not contain an object r_i, then that membrane will stop evolving at the time when r_i is supposed to reach the subscript 0. In this way, after $2m$ steps we can find whether there is a membrane which contains all objects $r_1, r_2, \ldots, r_m$. The membranes with this property, and only they, will get the object c_{m+1}. Note that at that moment the last object r_1, originating in r_m, enters the corresponding membrane, and changes its polarization to positive.

11. $[_2 c_{m+1}]_2^+ \to [_2\ ]_2^+ yes$.
12. $[_1 yes]_1^0 \to [_1\ ]_1^0 yes$.

The object c_{m+1} exits the membrane that has contained a truth-assignment that has satisfied formula C, producing the object yes. This object is then sent to the environment, telling us that the formula is satisfiable, and the computation stops. This is the $(n+2m+4)$th step of the computation. If formula C is not satisfiable, then the computation stops earlier and we get no output, because no membrane was able to produce the object c_{m+1}, hence the object yes.

The satisfiability of formula C was decided in linear time, and this concludes the proof. □

Let us denote by D the class of P systems with active membranes, using rules of types (a), (b), (c), (e), with 2-division.

Although the following result is a corollary of Theorem 7.2.2, we formulate it as a theorem, in order to stress its importance.

Theorem 7.2.3. $\mathtt{SAT} \in \mathbf{LMC}_D$.

Proof. We have to check the four conditions from the definition of the class $\mathbf{LMC}_D$ for the previous family of membrane systems. Consider m fixed, and denote by $\Pi_{\mathtt{SAT}}(n)$ the system constructed for an instance $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ of SAT which involves n variables. From the proof of the previous theorem it is clear that the system $\Pi_{\mathtt{SAT}}(n)$ always halts; specifically, it either

halts after $t_{\mathtt{SAT}(n)} = n + 2m + 4$ steps and sends out the object *yes*, or halts earlier but in that case no object is sent out. Thus, the system is confluent, and linearly efficient. From the explanations given together with the rules, we also know that the system is sound: the instance $\mathtt{SAT}(n)$ is satisfiable if and only if the system $\Pi_{\mathtt{SAT}}(n)$ sends the object *yes* to the environment.

It remains to prove that the family $\Pi_{\mathtt{SAT}}(n)$ is uniform, that is it can be constructed in polynomial time starting from $\mathtt{SAT}(n)$. A detailed, formal construction of a Turing machine which reads an instance of SAT and produces a P system as in the previous proof is totally cumbersome, although straightforward in principle. Here are some hints for such a construction.

We codify all substripts in a given base.

Suppose that the SAT instance is given in a standard normal form, that is without any clause containing a literal twice, with the clauses numbered from 1 to m, and with the variables numbered from 1 to n (hence, without gaps in numberings). Thus, the longest subscript will have $\max(n, m)$ digits even when the codification is made in base 1. This observation gives precise upper bounds for the length of rules of our systems (the rules involve objects associated with variables and with clauses, membranes of two types only, polarizations of three types, as well as additional symbols, such as $\rightarrow$ and the parentheses used for indicating the membranes). Note that we have in total $4n + 2m + 4$ rules, but the rules are clustered in types.

Suppose also that the instance we consider is non-trivial, in the sense that there is no clause C_i which contains both x_j and $\neg x_j$; a clause which contains both a variable and its negation is always true, hence it can be removed without changing the satisfiability of C. This implies the fact that each clause contains at most n literals (one further literal would repeat a variable).

Consider, for example, the sequence of rules of type 1; the same considerations hold for all types. We have the sequence

$$[_2 a_1]_2^0 \rightarrow [_2 t_1]_2^0 [_2 f_1]_2^0, [_2 a_2]_2^0 \rightarrow [_2 t_2]_2^0 [_2 f_2]_2^0, \ldots, [_2 a_n]_2^0 \rightarrow [_2 t_n]_2^0 [_2 f_n]_2^0.$$

(We have used a comma to separate the rules.) Such a sequence can be constructed as follows: write $[_2 a_1]_2^0 \rightarrow [_2 t_1]_2^0 [_2 f_1]_2^0$, on the tape of the Turing machine. This takes a constant time, proportional to the number of symbols we have to write. Then, copy this sequence of symbols to the right of the comma, at the same time increasing the subscripts of a, t, f by one. Continue in this way, copying the string of symbols between the last two commas to the right of the last comma, at the same time increasing the subscripts of a, t, f. When the subscripts become equal to n, stop. In order to copy a string of length k to the right of it, with a marker in between the two copies, we have to move the head of the Turing machine back and forth for each symbol of the string, carrying it (mark it, go to the right, write the symbol, return to the marked symbol, unmark it and continue with the symbol placed to the left of it, and so on and so forth) at most a distance k. This means a number of

steps of the order of k^2. (The copying procedure can be more efficient if at the same time we carry r symbols, for a given r greater than 1, but still the time is quadratic.) The longest rule in our sequence has the length of the order of n (actually, something like $3n + q$, depending on the way of codifying the auxiliary symbols $a, t, f, [,], \rightarrow, +, -, 1$). Therefore, copying one rule takes a time of the order of n^2. We have n rules, hence the total time is of the order of n^3.

Similar reasoning holds for all rules of types 1, 2, 8, 9 (where we have a sequence of rules), while for rules of types 3, 4, 7, 10, 11, 12 the polynomial bound of steps is ensured by the fact that we have a single rule of each type.

The case of rules of types 5, 6 is slightly different, because also the clauses are involved. For each rule of the form $[_2 t_i \rightarrow r_{h_{i,1}} \ldots r_{h_{i,j_i}}]_2^+$, $1 \leq i \leq n$, we also have to parse each clause of our formula from left to right. Because we deal with formulae which are non-trivial, the length of C is bounded, and we have at most n literals in each clause. Therefore, the string we have to parse when writing a rule as mentioned above is polynomially bounded with respect to m and n. Then, writing the whole sequence of rules is polynomially bounded with respect to the length of one rule, the longest one. In total, a polynomial time is needed for constructing all rules of type 5. The same assertion holds for rules of type 6.

Consequently, when m is fixed, we can construct all rules of Π in a time which is bounded by a polynomial with respect to n. If we also consider m as a variable, then the polynomial will depend on both n and m.

Most of the previous operations – especially, copying strings – can be performed much faster if we use multi-tape Turing machines, but we are only interested in the fact that the total time is polynomial, without taking care of the degree or the coefficients of the specific polynomial which bounds the total time.

Clearly, also the other elements of $\Pi_{\mathtt{SAT}}(n)$ – the alphabet, the set of labels, the initial membrane structure, and the contents of the two initial regions – can be constructed in a polynomial time with respect to n (and m). We conclude that the family of membrane systems we use is uniform, and this ends the proof of the fact that $\mathtt{SAT} \in \mathbf{LMC}_D$. □

7.2.2 Solving the Hamiltonian Path Problem

Another well-known **NP**-complete problem is HPP, the Hamiltonian path problem, which asks whether or not a given graph $\gamma = (N, E)$ (here we consider directed graphs) contains a Hamiltonian path, that is a path which visits all nodes from γ exactly once – we do not specify the first and the last node of a path, each node from N can be the first or the last one.

According to the definition of the class $\mathbf{LMC}_D$ – see again the discussion from the first section of this chapter – from the fact that SAT belongs to this class it follows that all problems from **NP** – HPP included – are in $\mathbf{LMC}_D$. However, we are interested here in *direct* solutions to HPP; that is, solutions

obtained by means of membrane systems which are constructed starting directly from an instance of HPP, thus avoiding the reduction to SAT.

First, we will prove that HPP can be solved in an easy way if we can divide membranes into more than two new membranes.

Theorem 7.2.4. *The directed* HPP *can be solved in linear time with respect to the number of nodes of the graph by P systems using rules of the forms* $(a), (b), (c)$, *and* (e), *with d-division.*

Proof. Let $\gamma = (N, E)$ be a directed graph with $n \geq 2$ nodes, $N = \{1, 2, \ldots, n\}$. We construct the system

$$\Pi = (O, \{1, 2\}, \mu, \lambda, dd_0, R),$$

where

$$\begin{aligned} O &= \{a_i, b_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 0 \leq i \leq n+1\} \\ &\quad \cup \{c_i \mid 0 \leq i \leq n+1\} \cup \{d_i \mid 0 \leq i \leq 2n\} \cup \{yes, d\}, \\ \mu &= [_1[_2\ ]_2^0]_1^0, \end{aligned}$$

while the set R contains the following rules (we also give explanations about the use of these rules during the computations):

1. $[_2 d]_2^0 \to [_2 a_1]_2^0 \cdots [_2 a_n]_2^0$.
   We create n membranes, all of them labelled with 2, and each of them containing one of the objects $a_1, \ldots, a_n$ which correspond to the nodes of γ. The idea is that we want to check whether or not a Hamiltonian path exists which starts in the ith node of the graph, for all i.
2. $[_2 d_k \to d_{k+1}]_2^0$, $0 \leq k \leq 2n-2$.
3. $[_2 d_{2n-1} \to d_{2n} c_1]_2^0$.
4. $[_2 d_{2n}]_1^0 \to [_1\ ]_2^+ d_{2n}$.
   We count to $2n$, which is the number of steps necessary in order to produce all paths of length n in the graph. When the counting is finished, the polarization of all membranes with label 2 becomes positive (one more step), and we pass to the next phase of the computation.
5. $[_2 a_i \to r_i b_i]_2^0$, $1 \leq i \leq n$.
6. $[_2 b_i]_2^0 \to [_2 a_{j_1}]_2^0 \cdots [_2 a_{j_k}]_2^0$, $1 \leq i \leq n, 1 \leq j_h \leq n$ for all $1 \leq h \leq k$, such that $(i, j_1), \ldots, (i, j_k)$ are edges from E.
   We create all paths in γ, starting from all nodes. Prolonging the paths with one node takes two steps, one when we use the rules of type 5 (which indicates by means of r_i the last node that was visited), and one when we use the rules of type 6 (which create as many new membranes as many descendants has the last node that was visited). Thus, in $2n$ steps we generate all paths of length n – which is the length of any Hamiltonian path in γ, providing that one exists. Remember that the rule of type 4

changes the polarization of the membrane where it is applied, hence after using that rule we cannot continue to use rules of types 5 and 6. However, in the step when we use the rule of type 4 we also use rules of type 5, for all existing objects a_i, and in this way we can have n objects r_i in each membrane 2.

7. $[_2 r_1]_2^+ \to [_2\ ]_2^- r_1$.
8. $r_1 [_2\ ]_2^- \to [_2 r_0]_2^+$.
9. $[_2 r_i \to r_{i-1}]_2^-$, $1 \leq i \leq n$.
10. $[_2 c_i \to c_{i+1}]_2^-$, $1 \leq i \leq n$.
11. $[_2 c_{n+1}]_2^+ \to [_2\ ]_2^+ yes$.
12. $[_1 yes]_1^0 \to [_1\ ]_1^0 yes$.

After changing the polarization of membranes with label 2 to positive, we have all candidate paths generated, and it only remains to check whether or not any of them is Hamiltonian. This is the case if and only if there is at least one membrane that contains all objects $r_1, r_2, \ldots, r_n$. Checking whether or not such a membrane exists is done exactly as in the proof of Theorem 7.2.2, by means of rules of types 7–10. In the affirmative case, the object c_{n+1} is produced, and it will send outside the corresponding membrane 2 the object *yes* which will immediately exit the system. This happens after $4n + 3$ steps.

From the previous explanations, it is now clear that we get the object *yes* if and only if the graph γ contains at least one Hamiltonian path (therefore, the family of membrane systems we have constructed is sound, confluent, and linearly efficient). When starting from a graph with n nodes, we obtain a system with $6n + 6$ rules, clustered in classes of similar forms, with the longest rule in a class being polynomially bounded with respect to n. Rules of type 1 and 6 are the longest, as they involve variables corresponding to several nodes of the graph. However, there is only one rule of type 1, hence its construction is clear. In the case of rules of type 6 we have a sequence, but we can construct it as explained in the proof of Theorem 7.2.3. The details are left to the reader; we only conclude that the family of P membrane systems constructed above for solving HPP is uniform. □

We do not know a direct way of solving HPP in linear time by using membrane systems with 2-division, but we can solve HPP in such a framework in polynomial (quadratic) time.

Indeed, consider a graph $\gamma = (N, E)$, with $n \geq 2$ nodes, denoted by $1, \ldots, n$, and construct the system $\Pi = (O, H, \mu, w_0, w_1, R)$, with the following components:

$$\begin{aligned} O = &\{a_i, p_i, r_i \mid 1 \leq i \leq n\} \cup \{f_i, q_i \mid 1 \leq i \leq n-1\} \\ &\cup \{c_i \mid 0 \leq i \leq n+1\} \cup \{d_i \mid 0 \leq i \leq n^2\} \\ &\cup \{p_{i,j_1 \ldots j_k} \mid 1 \leq i \leq n, 1 \leq j_s \leq n, (i, j_s) \in E, 1 \leq s \leq k\} \cup \{b, yes\}, \end{aligned}$$

$$H = \{1, 2\},$$
$$\mu = [_1[_2\ ]_2^0]_1^0,$$
$$w_1 = f_1,$$
$$w_2 = q_1,$$

and the set R contains the following rules:

1. $[_2 q_i]_2^0 \to [_2 a_i]_2^0 [_2 q_{i+1}]_2^0$, $1 \le i \le n-2$.
2. $[_2 q_{n-1}]_2^0 \to [_2 a_{n-1}]_2^0 [_2 a_n]_2^0$.
   We start by dividing the central membrane n times, in each of the copies introducing one object a_i, corresponding to the n nodes of the graph. The idea is to start generating paths in γ from each of these nodes, in search of a Hamiltonian path. This takes $n-1$ steps.
3. $[_1 f_i \to f_{i+1}]_1^0$, $1 \le i \le n-2$.
4. $[_1 f_{n-1} \to b^n]_1^0$.
   In parallel with the use of rules of type 1, in the skin membrane we count to $n-1$ by means of objects f_i. In the last step, that with number $n-1$, the counter f_{n-1} introduces n copies of the object b.
5. $b[_2\ ]_2^0 \to [_2 d_0]_2^+$.
   In the next step, that with number n, the n copies of b enter the n membranes with label 2, change the polarization of these membranes to positive, and are replaced by d_0. Thus, after this step we have the configuration
   $$[_1[_2 a_1 d_0]_2^+ [_2 a_2 d_0]_2^+ \dots [_2 a_n d_0]_2^+]_1^0.$$
   The next phase will generate paths in γ starting from each node $1, 2, \dots, n$. For each node i, assume the nodes $j_1, \dots, j_k$ with $(i, j_s) \in E$, $i \ne j_s$ (the nodes of γ accessible from node i), are ordered such that $j_1 < j_2 < \dots < j_k$.
6. $[_2 a_i \to r_i p_i]_2^+$, $1 \le i \le n$.
7. $[_2 p_i]_2^+ \to [_2 a_{j_1}]_2^+ [_2 p_{i,j_1}]_2^+$, for $(i, j_1) \in E$,
   $[_2 p_{i,j_1 \dots j_s}]_2^+ \to [_2 a_{j_{s+1}}]_2^+ [_2 p_{i,j_1 \dots j_{s+1}}]_2^+$, for all $1 \le s \le k-3$,
   $[_2 p_{i,j_1 \dots j_{k-2}}]_2^+ \to [_2 a_{j_{k-1}}]_2^+ [_2 a_{j_k}]_2^+$, for all $1 \le i \le n$.
   By 2-division, we create as many membranes with label 2 as there are descendants of the last node from a path. The polarization of membranes 2 always remains positive, hence the process can be continued for each new object a_s. Because of the way the contents of membranes are replicated when dividing them, the previously introduced objects r_u are reproduced in the new membranes, hence the paths are continued step by step. We have n nodes in the graph, hence each of them can have at most $n-1$ descendants (we ignore the edges of the form (i, i), if such edges exist in γ). This means that the above process continues at most $n(n-1)+1$ steps.
8. $[_2 d_i \to d_{i+1}]_2^+$, for $0 \le i \le n^2 - 2$,
   $[_2 d_{n^2-1} \to d_{n^2} c_1]_2^+$,

$[_2 d_{n^2}]_2^+ \to [_2\ ]_2^- d_{n^2}$.
In parallel with the previous process of generating paths in the graph, we count to n^2 by means of the objects d_i. At step n^2 we introduce the objects d_{n^2} and c_1. The former object exits membrane 2 where it is produced and changes its polarization to negative.

We now have a configuration the same as that obtained in the proof of Theorem 7.2.4 after applying the rules of type 6: in a number of membranes with label 2 and negative polarization, we have paths in γ encoded by means of objects r_i, as well as the counter c_1. Using rules of types 7–12 from the proof of Theorem 7.2.4, in $2n+2$ steps we can check whether or not there is any path which is Hamiltonian. In the affirmative case, in the step n^2+3n+4 we send the object *yes* out of the system, otherwise the computation stops earlier without sending out any object. Thus, we have solved HPP in polynomial time.

7.2.3 Using Cooperative Rules

Working with non-cooperative rules is natural from a mathematical point of view, but from a biochemical point of view this is not only unnecessary, but also unrealistic: most of the chemical reactions involve two or more chemical compounds (and also produce two or more compounds). "Reactions" of the form $a \to bc$ correspond to breaking a molecule a into two smaller molecules, b and c, but many reactions are of the form $ab \to cd$, where a and b interact in producing c and d.

We know from the previous chapters that the cooperative rules are very useful concerning the computing power of membrane systems: even systems of simple forms are computationally universal when using cooperative rules. Here we are not interested in the computing capacity (we know from Theorem 7.2.1 that systems with active membranes using only non-cooperative rules are universal), but in the computing efficiency. Not entirely surprisingly, we find that by using cooperative rules of type (a), as well as (non-cooperative) rules of the other types (b)–(e), we can solve many **NP**-complete problems in a rather uniform manner, that is by systems which are very similar to each other.

The general structure (and functioning) of these systems is rather similar in principle to the structure of the systems from Theorems 7.2.2 and 7.2.4, but much more uniform in the cooperative case:

1. We always start with two membranes, and the central one is always divided into an exponential number of copies.
2. In a central "parallel engine" one generates, making use of the membrane division, a "data pool" of an exponential size; due to the parallelism, however, this takes only a linear time. In parallel with this process, a "timer" is simultaneously ticking, in general for synchronization reasons.

3. After finishing the generation of the "data pool", one checks whether or not any solution to the problem exists; this is the step where we use cooperative rules.
4. A message is sent out of the system at a precise moment telling whether or not the problem has a solution. In all cases, the last two steps are done in a constant number of time units, again making use of the parallelism.

Figure 7.3 pictorially suggests the general shape of the systems which follow.

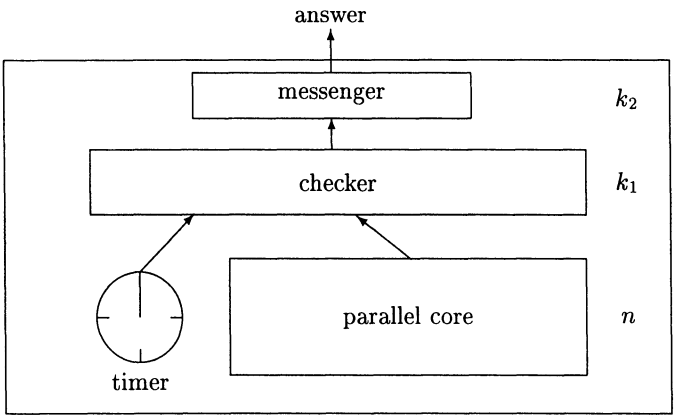


Fig. 7.3. The shape of P systems solving **NP**-complete problems

Many other decidability problems can be approached in the same way, but here we just illustrate the idea with five problems from logic and five from graph theory. We approach SAT also in this framework, obtaining the solution in a time which is shorter than when using non-cooperative rules. As in the case of the proofs of Theorems 7.2.2 and 7.2.4, we use systems which have rules of forms (a)–(e) (but not of all types) and never rules of type (f), trying always to be as "economical" as possible from this point of view. For instance, in the case of the problems from logic, we do not use rules for the membrane dissolution. Moreover, and this is an important difference from the case of non-cooperative rules, we never use membrane polarizations (for uniformity, we may assume that the membranes are always neutral, but we do not explicitly indicate the neutral polarization).

Because in all cases we have the initial membrane structure $[_1[_2\ ]_2]_1$, and an empty multiset in the skin membrane, we no longer specify these items, but only give the other components of the systems, the set O of objects, the initial multiset w_2, and the set of evolution rules. For the formulation of the problems, we closely follow [73], from where we also borrow the codes identifying the problems: [LOs] in the case of logic and [GTs] in the case of graph theory.

PROPOSITIONAL LOGIC

[LO1] **Satisfiability** (SAT)

INSTANCE: A collection C of clauses $C_i = y_{i,1} \vee \ldots \vee y_{i,p_i}$, $1 \leq i \leq m$, for some $m \geq 1, p_i \geq 1$, and $y_{i,j} \in \{x_k, \neg x_k \mid 1 \leq k \leq n\}, n \geq 1$, for each $1 \leq i \leq m, 1 \leq j \leq p_i$.

QUESTION: Is there a satisfying truth–assignment for C?

P SYSTEM:

$$
\begin{aligned}
O &= \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{c_i \mid 1 \leq i \leq n+1\} \\
&\quad \cup \{k_i \mid 1 \leq i \leq m\} \cup \{yes\}, \\
w_2 &= c_1 a_1 \ldots a_n, \\
R &= \{[_2 a_i]_2 \to [_2 t_i]_2 [_2 f_i]_2 \mid 1 \leq i \leq n\} \ (\textit{parallel engine}) \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \leq i \leq n\} \ (\textit{timer}) \\
&\quad \cup \{[_2 c_{n+1} t_i \to k_1 t_i]_2 \mid x_i \text{ appears in } C_1, 1 \leq i \leq n\} \\
&\quad \cup \{[_2 c_{n+1} f_i \to k_1 f_i]_2 \mid \neg x_i \text{ appears in } C_1, 1 \leq i \leq n\} \\
&\quad \cup \{[_2 k_j t_i \to k_{j+1} t_i]_2 \mid x_i \text{ appears in } C_{j+1}, 1 \leq i \leq n, 1 \leq j \leq m-1\} \\
&\quad \cup \{[_2 k_j f_i \to k_{j+1} f_i]_2 \mid \neg x_i \text{ appears in } C_{j+1}, 1 \leq i \leq n, \\
&\qquad 1 \leq j \leq m-1\} \ (\textit{truth-checking rules}) \\
&\quad \cup \{[_2 k_m]_2 \to [_2\ ]_2 yes, \\
&\qquad [_1 yes]_1 \to [_1\ ]_1 yes\} \ (\textit{messenger rules}).
\end{aligned}
$$

In n steps, we generate all 2^n truths–assignments of the n variables; at the same time, the counter c_i arrives at c_{n+1} and starts checking the truth of clauses; if a clause C_j is valid, then the object k_j is introduced. Checking all clauses takes m steps. If in any copy of membrane 2 we can obtain the object k_m, then the "message" *yes* is sent to membrane 1 and from here out of the system. This means that the set C of clauses is satisfiable if and only if at step $n + m + 2$ we get the object *yes* outside the system.

Note that for each clause we have as many checking rules as there are many literals in the clause.

[LO2] **3-Satisfiability (3-SAT)**

INSTANCE: As in the case of SAT, with each clause containing exactly three literals (that is, $p_i = 3$ for all $1 \leq i \leq m$).

QUESTION: Is there a satisfying truth–assignment for C?

P SYSTEM: As above, but with only three truth-checking rules for each clause.

[LO3] **Not-all-equal 3SAT**

INSTANCE: As in the case of 3-SAT.

QUESTION: Is there a truth–assignment such that each clause in C has at least one true literal and at least one false literal?

P SYSTEM: As above, but with the following truth-checking rules:

$$\{[_2 c_{n+1} t_{i_1} t_{i_2} \to k_1 t_{i_1} t_{i_2}]_2 \mid x_{i_1}, \neg x_{i_2} \text{ appear in } C_1, 1 \le i_1, i_2 \le n\}$$
$$\cup \{[_2 c_{n+1} t_{i_1} f_{i_2} \to k_1 t_{i_1} f_{i_2}]_2 \mid x_{i_1}, x_{i_2} \text{ or } \neg x_{i_1}, \neg x_{i_2} \text{ appear in } C_1, 1 \le i_1, i_2 \le n\}$$
$$\cup \{[_2 c_{n+1} f_{i_1} f_{i_2} \to k_1 f_{i_1} f_{i_2}]_2 \mid \neg x_{i_1}, x_{i_2} \text{ appear in } C_1,\ 1 \le i_1, i_2 \le n\}$$
$$\cup \{[_2 k_j t_{i_1} t_{i_2} \to k_{j+1} t_{i_1} t_{i_2}]_2 \mid x_{i_1}, \neg x_{i_2} \text{ appear in } C_{j+1}, 1 \le i_1, i_2 \le n, 1 \le j \le m-1\}$$
$$\cup \{[_2 k_j t_{i_1} f_{i_2} \to k_{j+1} t_{i_1} f_{i_2}]_2 \mid x_{i_1}, x_{i_2} \text{ or } \neg x_{i_1}, \neg x_{i_2} \text{ appear in } C_{j+1}, 1 \le i_1, i_2 \le n, 1 \le j \le m-1\}$$
$$\cup \{[_2 k_j f_{i_1} f_{i_2} \to k_{j+1} f_{i_1} f_{i_2}]_2 \mid \neg x_{i_1}, x_{i_2} \text{ appear in } C_{j+1}, 1 \le i_1, i_2 \le n, 1 \le j \le m-1\}.$$

For each clause we have three rules in each membrane 2.

The previous rules contain three objects in their left side, but it is easy to modify them in such a way as to have only two objects in each case. For instance, instead of a rule $[_2 k_j t_{i_1} t_{i_2} \to k_{j+1} t_{i_1} t_{i_2}]_2$ we can use the rules

$$[_2 k_j t_{i_1} \to \langle k_j t_{i_1} \rangle]_2,$$
$$[_2 \langle k_j t_{i_1} \rangle t_{i_2} \to k_{j+1} t_{i_1} t_{i_2}]_2,$$

where $\langle k_j t_{i_1} \rangle$ is a new object. Instead of m steps, we now need $2m$ steps for checking the truth values of the m clauses (hence the answer is obtained at the step $n + 2m + 2$).

In the same way, with a linear slowdown, we can change all cooperative rules from the systems which follow, so we will not state this observation again for each case.

[LO4] **One-in-three 3SAT**

INSTANCE: As in the case of 3-SAT.

QUESTION: Is there a truth–assignment such that each clause in C has exactly one true literal?

P SYSTEM: Exactly as above, but with the following truth-checking rules:

$$\{[_2 c_{n+1} t_{i_1} t_{i_2} t_{i_3} \to k_1 t_{i_1} t_{i_2} t_{i_3}]_2 \mid x_{i_1}, \neg x_{i_2}, \neg x_{i_3} \text{ appear in } C_1, 1 \le i_1, i_2, i_3 \le n\}$$
$$\cup \{[_2 c_{n+1} t_{i_1} t_{i_2} f_{i_3} \to k_1 t_{i_1} t_{i_2} f_{i_3}]_2 \mid x_{i_1}, \neg x_{i_2}, x_{i_3} \text{ or } \neg x_{i_1}, \neg x_{i_2}, \neg x_{i_3} \text{ appear in } C_1,\ 1 \le i_1, i_2, i_3 \le n\}$$
$$\cup \{[_2 c_{n+1} t_{i_1} f_{i_2} f_{i_3} \to k_1 t_{i_1} f_{i_2} f_{i_3}]_2 \mid x_{i_1}, x_{i_2}, x_{i_3} \text{ or } \neg x_{i_1}, x_{i_2}, \neg x_{i_3} \text{ appear in } C_1,\ 1 \le i_1, i_2, i_3 \le n\}$$
$$\cup \{[_2 c_{n+1} f_{i_1} f_{i_2} f_{i_3} \to k_1 f_{i_1} f_{i_2} f_{i_3}]_2 \mid \neg x_{i_1}, x_{i_2}, x_{i_3} \text{ appear in } C_1, 1 \le i_1, i_2, i_3 \le n\}$$

$$\cup \{[_2 k_j t_{i_1} t_{i_2} t_{i_3} \to k_{j+1} t_{i_1} t_{i_2} t_{i_3}]_2 \mid x_{i_1}, \neg x_{i_2}, \neg x_{i_3} \text{ appear in } C_{j+1},$$
$$1 \le i_1, i_2, i_3 \le n\}$$
$$\cup \{[_2 k_j t_{i_1} t_{i_2} f_{i_3} \to k_{j+1} t_{i_1} t_{i_2} f_{i_3}]_2 \mid x_{i_1}, \neg x_{i_2}, x_{i_3} \text{ or } \neg x_{i_1}, \neg x_{i_2}, \neg x_{i_3}$$
$$\text{appear in } C_{j+1},\ 1 \le i_1, i_2, i_3 \le n, 1 \le j \le m-1\}$$
$$\cup \{[_2 k_j t_{i_1} f_{i_2} f_{i_3} \to k_{j+1} t_{i_1} f_{i_2} f_{i_3}]_2 \mid x_{i_1}, x_{i_2}, x_{i_3} \text{ or } \neg x_{i_1}, x_{i_2}, \neg x_{i_3}$$
$$\text{appear in } C_{j+1},\ 1 \le i_1, i_2, i_3 \le n, 1 \le j \le m-1\}$$
$$\cup \{[_2 k_j f_{i_1} f_{i_2} f_{i_3} \to k_{j+1} f_{i_1} f_{i_2} f_{i_3}]_2 \mid \neg x_{i_1}, x_{i_2}, x_{i_3}$$
$$\text{appear in } C_{j+1},\ 1 \le i_1, i_2, i_3 \le n, 1 \le j \le m-1\}.$$

For each clause we have exactly one rule in each membrane 2 (we will have three rules if we reduce to two the number of objects which cooperate in each rule).

[LO5] **Minimum 2-satisfiability**

INSTANCE: A set of n variables, a collection C of m clauses such that each clause contains exactly two literals ($p_i = 2$ for all $1 \le i \le m$), and a positive integer $k \le m$.

QUESTION: Is there a truth–assignment that simultaneously satisfies at least k clauses in C?

P SYSTEM:

$$\begin{aligned}
O &= \{a_i, t_i, t'_i, f_i, f'_i \mid 1 \le i \le n\} \\
&\quad \cup \{c_i \mid 1 \le i \le n+3\} \\
&\quad \cup \{d_i \mid 1 \le i \le m\} \cup \{d, yes\}, \\
w_2 &= c_1 a_1 \ldots a_n, \\
R &= \{[_2 a_i]_2 \to [_2 t'_i]_2 [_2 f'_i]_2 \mid 1 \le i \le n\} \\
&\quad \cup \{[_2 t'_i \to t_i^m]_2, \\
&\qquad [_2 f'_i \to f_i^m]_2 \mid 1 \le i \le n\} \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \le i \le n, \text{ and } i = n+2\} \\
&\quad \cup \{[_2 c_{n+1} \to c_{n+2} d_1 d_2 \ldots d_m]_2\} \\
&\quad \cup \{[_2 d_j t_i \to d t_i]_2 \mid x_i \text{ appears in } C_{j+1}, 1 \le i \le n, 1 \le j \le m\} \\
&\quad \cup \{[_2 d_j f_i \to d f_i]_2 \mid \neg x_i \text{ appears in } C_{j+1}, 1 \le i \le n, 1 \le j \le m\} \\
&\quad \cup \{[_2 c_{n+3} d^k \to yes]_2, \\
&\qquad [_2 yes]_2 \to [_2\ ]_2 yes, \\
&\qquad [_1 yes]_1 \to [_1\ ]_1 yes\}.
\end{aligned}$$

The answer is obtained in $n+5$ steps ($n+k+4$ steps if we "reduce" the rule $[_2 c_{n+3} d^k \to yes]_2$ to rules with only two cooperating objects), because the checking of the truth of the m clauses is done in parallel for all of them, in only one step, due to the presence of m copies of each truth value t_i, f_i

and m objects $d_1, \ldots, d_m$ in each membrane 2. (This trick, with checking all clauses in parallel, in only one step, can also be applied for SAT, but at the cost of using a rule of the form $[_2 c_{n+3} d^m \to yes]_2$, which checks whether or not *all* clauses are satisfied; if we replace this rule with m rules with only two objects in their left-hand sides, then we return to a system similar to that considered above for SAT.) The fact that each clause contains only two literals is "visible" only in what concerns the number of checking rules, not in the shape of the system.

In the case of logic, we have chosen the first problems from the beginning of the respective section from [73], but for graph theory we have to select decision problems; moreover, we present the problems in the order of their difficulty/similarity.

GRAPH THEORY

[GT1] **Vertex cover**

INSTANCE: A directed graph $\gamma = (V, E)$ with n vertices, $v_1, \ldots, v_n$, and a positive integer $k \leq card(V)$.

QUESTION: Is there a subset $V' \subseteq V$ with $card(V') \leq k$ such that for all $(v_i, v_j) \in E, 1 \leq i, j \leq n$, at least one of v_i, v_j is in V'?

P SYSTEM:

$$\begin{aligned}
O &= \{v_i, v_i', v_i'', v_i''' \mid 1 \leq i \leq n+1\} \\
&\quad \cup \{c_i \mid 1 \leq i \leq n+3\} \cup \{c, d, yes\}, \\
w_2 &= c_1 v_1 \ldots v_n v_{n+1}, \\
R &= \{[_2 v_i]_2 \to [_2 v_i''']_2 [_2 v_i'']_2 \mid 1 \leq i \leq n+1\} \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \leq i \leq n+2\} \\
&\quad \cup \{[_2 v_i''' \to v_i' c]_2 \mid 1 \leq i \leq n+1\} \\
&\quad \cup \{[_2 c_{n+3} v_i'' v_j'' \to d]_2 \mid (v_i, v_j) \in E, 1 \leq i, j \leq n\} \\
&\quad \cup \{[_2 c_{n+3} c^{k+1} \to d]_2, \\
&\qquad [_2 d]_2 \to d, \;\; d[_2\ ]_2 \to [_2 yes]_2, \\
&\qquad [_2 yes]_2 \to [_2\ ]_2 yes, \;\; [_1 yes]_1 \to [_1\ ]_1 yes\}.
\end{aligned}$$

In $n+1$ steps we generate all possible partitions of $V \cup \{v_{n+1}\}$ in two subsets, by triple priming and double priming the objects; the tripled primed objects correspond to the set V' we look for. In step $n+2$ the triple primed objects introduce a single primed version as well as the object c, used for checking whether or not $card(V') \leq k$. This is done at step $n+3$ by the rule $[_2 c_{n+2} c^{k+1} \to d]_2$. Because we have considered an object v_{n+1}, plus its primed versions, which does not correspond to a vertex in V, we can have $k = card(V)$, hence $V' = V$; if also v_{n+1} is put in V', then we get $k+1$ copies of c; if v_{n+1} is not in V', then the rule $[_2 c_{n+3} c^{k+1} \to d]_2$ is not applicable. The object d will dissolve the copy of membrane 2 where it appears. The rules

$[_2 c_{n+3} v_i'' v_j'' \to d]_2$ are present for all edges in E (hence not for $i, j = n+1$) and any rule of this form can be applied if and only if the negation of the property in the problem is fulfilled. In such a case, the membrane 2 which corresponds to that subset V' is dissolved. Note that at least one membrane is dissolved, for instance for $V' = V \cup \{v_{n+1}\}$ (because in this case we get $card(V') \geq k+1$), hence at least one copy of d will be produced in membrane 1. If any membrane survives, then the rule $d[_2\ ]_2 \to [_2 yes]_2$ can be applied. This means that a satisfactory set V' exists and the corresponding message is sent out of the system (at step $n+7$).

[GT58] **k-Closure**

INSTANCE: The same as above.

QUESTION: Is there a subset $V' \subseteq V$ with $card(V') \leq k$ such that for all $(v_i, v_j) \in E, 1 \leq i, j \leq n$, either $v_i \in V'$ or $v_j \notin V'$?

P SYSTEM: Exactly the same as for Vertex cover, with the checking rules $[_2 c_{n+3} v_i'' v_j'' \to d]_2$ replaced by

$$\{[_2 c_{n+3} v_i'' v_j' \to d]_2,$$
$$[_2 c_{n+3} v_i' v_j'' \to d]_2 \mid (v_i, v_j) \in E, 1 \leq i, j \leq n\}.$$

[GT20] **Independent set**

INSTANCE: The same as above (but we may assume that $k < card(V)$, because the case when $k = card(V)$ corresponds to $E = \emptyset$, which is trivial).

QUESTION: Is there a subset $V' \subseteq V$ with $card(V') \geq k$ such that no two vertices in V' are joined by an edge in E?

P SYSTEM: Very similar to that for Vertex cover, but we give it in full detail just to let the reader observe the similarities.

$$\begin{aligned}
O &= \{v_i, v_i', v_i'', v_i''' \mid 1 \leq i \leq n\} \\
&\quad \cup \{c_i \mid 1 \leq i \leq n+2\} \cup \{c, d, yes\}, \\
w_2 &= c_1 v_1 \ldots v_n, \\
R &= \{[_2 v_i]_2 \to [_2 v_i']_2 [_2 v_i''']_2 \mid 1 \leq i \leq n\} \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \leq i \leq n+1\} \\
&\quad \cup \{[_2 v_i''' \to v_i'' c]_2 \mid 1 \leq i \leq n\} \\
&\quad \cup \{[_2 c_{n+2} v_i' v_j' \to d]_2 \mid (v_i, v_j) \in E, 1 \leq i, j \leq n\} \\
&\quad \cup \{[_2 c_{n+2} c^{n-k+1} \to d]_2, \\
&\qquad [_2 d]_2 \to d,\ d[_2\ ]_2 \to [_2 yes]_2, \\
&\qquad [_2 yes]_2 \to [_2\ ]_2 yes,\ [_1 yes]_1 \to [_1\ ]_1 yes\}.
\end{aligned}$$

Note that this time we need to check that we have *at least* k selected vertices, hence we dissolve a membrane 2 only when less than k vertices are marked with a prime.

[GT4] **Graph 3-colourability**

INSTANCE: The same as above.

QUESTION: Does there exist a function $f : V \longrightarrow \{1,2,3\}$ such that $f(v_i) \neq f(v_j)$ whenever $(v_i, v_j) \in E$?

P SYSTEM:

$$
\begin{aligned}
O &= \{v_i^\alpha \mid 1 \leq i \leq n, \alpha \in \{(1),(2),(3),(23)\} \\
&\quad \cup \{c_i \mid 1 \leq i \leq 2n+2\} \cup \{d, yes\}, \\
w_2 &= c_1 v_1 \ldots v_n, \\
R &= \{[_2 v_i]_2 \to [_2 v_i^{(1)}]_2 [_2 v_i^{(23)}]_2, \\
&\qquad [_2 v_i^{(23)}]_2 \to [_2 v_i^{(2)}]_2 [_2 v_i^{(3)}]_2 \mid 1 \leq i \leq n\} \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \leq i \leq 2n+1\} \\
&\quad \cup \{[_2 c_{2n+1} v_i^{(r)} v_j^{(r)} \to d]_2 \mid (v_i, v_j) \in E, 1 \leq i, j \leq n, 1 \leq r \leq 3\} \\
&\quad \cup \{[_2 d]_2 \to d,\ d[_2\ ]_2 \to [_2 yes]_2, \\
&\qquad [_2 yes]_2 \to [_2\ ]_2 yes,\ [_1 yes]_1 \to [_1\ ]_1 yes\}.
\end{aligned}
$$

We leave to the reader the task to see how this system works (and, also, to see the differences and the similarities with the previous systems). We only mention that we dissolve a membrane 2 at least for constant functions f, hence we always produce an object d – but if all membranes 2 are dissolved, then d has no membrane to enter, hence we cannot also produce the object yes.

[GT6] **Monochromatic triangle**

INSTANCE: A graph $\gamma = (V, E)$ with n vertices and m edges ($E = \{e_1, \ldots, e_m\}$).

QUESTION: Is there a partition of E into two disjoint sets E_1, E_2 such that neither $\gamma_1 = (V, E_1)$ nor $\gamma_2 = (V, E_2)$ contains a triangle?

P SYSTEM: Again very similar to the previous ones, but this time we have to find a subset E' of E, such that, simultaneously, $card(E') \geq 1$ and $card(E') \leq m - 1$.

$$
\begin{aligned}
O &= \{e_i, e_i', e_i'', \bar{e}_i', \bar{e}_i'' \mid 1 \leq i \leq n\} \\
&\quad \cup \{c_i \mid 1 \leq i \leq m+2\} \cup \{d, g, h, yes\}, \\
w_2 &= c_1 e_1 \ldots e_m, \\
R &= \{[_2 e_i]_2 \to [_2 \bar{e}_i']_2 [_2 \bar{e}_i'']_2 \mid 1 \leq i \leq m\} \\
&\quad \cup \{[_2 c_i \to c_{i+1}]_2 \mid 1 \leq i \leq m+1\} \\
&\quad \cup \{[_2 \bar{e}_i' \to e_i' g]_2, \\
&\qquad [_0 \bar{e}_i'' \to e_i'' h]_0 \mid 1 \leq i \leq m\} \\
&\quad \cup \{[_2 c_{m+2} g^m \to d]_2,
\end{aligned}
$$

$$
\begin{aligned}
& [_2 c_{m+2} h^m \to d]_2\} \\
\cup\ & \{[_2 c_{m+2} e_i' e_j' e_k' \to d]_2, \\
& [_2 c_{m+2} e_i'' e_j'' e_k'' \to d]_2 \mid 1 \le i, j, k \le m, \\
& e_i = (v_{s_1}, v_{s_2}), e_j = (v_{s_2}, v_{s_3}), e_j = (v_{s_3}, v_{s_1}), 1 \le s_1, s_2, s_3 \le n\} \\
\cup\ & \{[_2 d]_2 \to d,\ d[_2\ ]_2 \to [_2 yes]_2, \\
& [_2 yes]_2 \to [_2\ ]_2 yes,\ [_1 yes]_1 \to [_1\ ]_1 yes\}.
\end{aligned}
$$

It is probably a matter of routine to find other decidability problems which can be solved with the same "partially universal" type of membrane systems with cooperative rules (of type (a)) and membrane division, hence we do not persist in this direction here.

7.2.4 Is Membrane Division Necessary?

Not for universality, we have seen this in Theorem 7.2.1, and not necessarily for obtaining polynomial solutions to **NP**-complete problems, because we can use other ways for generating an exponential space, as we will see in the next sections. However, in the precise framework of systems with active membranes, it seems that the answer is positive: systems without membrane division *which evolve in a deterministic way* (in each step there is at most one possible transition) cannot solve **NP**-complete problems in polynomial time – unless $\mathbf{P} = \mathbf{NP}$, which is not at all expected. More precisely, the following result is true.

Theorem 7.2.5. *A deterministic P system with active membranes but without membrane division can be simulated by a deterministic Turing machine with a polynomial slowdown.*

Proof. Consider a system $\Pi = (O, H, \mu, w_1, \ldots, w_m, R)$ without membrane division such that any computation in Π stops in t steps. Such a system can easily be reduced to an equivalent system Π' which stops in exactly t steps: we only have to add a "counter" symbol which evolves for exactly t steps. Let us denote by A, B, and C, respectively, the number of membranes, the number of symbols, and the maximal length of a rule (the number of symbols necessary to write a rule, both its left and right sides, the membranes, and the polarizations of membranes involved in the rule) of the system Π. Denote $D = \max(C, B + 2)$.

We build a deterministic Turing machine M with multiple tapes, which simulates any computation of length t in Π' in a number of steps of the order of $A \cdot B \cdot D \cdot t \cdot \log(A \cdot B \cdot C^t)$.

To simulate Π' with M, we have to keep track of the multiplicity of each object in each membrane. For instance, consider the application of a rule $[_i a \to bc]_i^\alpha$: all copies of the object a present in membrane i are substituted with two objects, b and c. We can simulate this rule by adding to the multiplicity of objects b and c in membrane i the number of copies of a and then

by setting the last quantity to zero. In other words, the application of a rule in Π' corresponds to the modification of the number of copies of the objects involved in the rule of the specific membrane (and eventually, as we will see, in modifying the multiplicity of all objects of a membrane, if a membrane placed immediately inside it is dissolved). The modification of the multiplicities is done by adding the multiplicity of the object from the left side of the rule to each multiplicity of the objects from the right side of the rule.

For the previous reasons, the Turing machine M has $2 \cdot A \cdot B + 3$ tapes:

- $A \cdot B$ "main" tapes, to keep track of the multiplicities of each object in each membrane (A membranes and B symbols) after every step,
- $A \cdot B$ "support" tapes, used to make partial sums,
- one "polarity" tape, used to keep track of the polarity of each membrane,
- one "structure" tape, used to keep track of the membrane structure,
- one output tape.

Every computation step of Π' can be simulated by means of two macro steps of M:

1. We simulate the application of a rule on each object in each membrane by modifying the quantity written on the tapes containing the partial sums. For example, consider a rule $[_i a \to bc]_i^\alpha$, and let the strings written on the tapes corresponding to the symbols a, b, and c in membrane i contain the multiplicities 100, 200, and 250. To simulate this rule, we have to put the value 0 in the first string, 300 in the second one, and 350 in the third one. We do not write these quantities on the main tapes, because in order to simulate the application of another rule $[_i b \to de]_i^\alpha$ we have to know that the quantity of symbols b when the computation step started was 200. For this reason we need support tapes. The rule to be applied (only one rule per symbol can be applied, because we consider deterministic systems) is chosen according to the polarity of the membrane; the polarity can be found in the "polarity" tape.

2. When the application of the rules has been executed on all objects, we copy the multiplicity of each object from the support tapes to the main tapes.

We have to repeat these two steps for t times. To simulate a single step of computation of Π', we have to simulate the application of at most $A \cdot B$ different rules from R (one rule for each object). The number of sums to be executed for each rule in each membrane depends on the type of the rule.

Consider the four types of rules in R:

1. Rules of type (a): $[_i a \to x]_i^\alpha$, where $a \in O$ and $x \in O^*, |x| \leq C - 1$. To simulate such a rule, we have to execute at most C sums.
2. Rules of type (b): $a[_i\ ]_i^\alpha \to [_i b]_i^\beta$, where $a, b \in O$. To simulate such a rule, we have to sum 1 to the multiplicity of the object b in membrane i and to sum -1 to the multiplicity of the object a in the membrane placed immediately outside membrane i (remember that at each computation

step only one object can enter a rule of type (b), (c), or (d) in each membrane). If needed, we change the charge of the membrane in the "polarity" tape.

3. Rules of type (c): $[_i a]_i^\alpha \to [_i\ ]_i^\beta b$, where $a, b \in O$. To simulate such a rule, we have to sum 1 to the multiplicity of the object b in the membrane placed immediately outside membrane i and to sum -1 to the multiplicity of the object a in membrane i. If needed, we change the charge of the membrane in the "polarity" tape. If i is the skin membrane, then we have to write the symbol a on the output tape.
4. Rules of type (d): $[_k[_i a]_i^\alpha]_k^\beta \to [_k b]_k^\gamma$, where $a, b \in O$. To simulate such a rule, we have to sum 1 to the multiplicity of the object b in the membrane placed immediately outside membrane i, and to sum -1 to the multiplicity of the object a in membrane i. Then we have to sum the multiplicity of each object from membrane i to the multiplicity of the same object in membrane k. Thus, we have to execute $B+2$ sums. After these sums, we modify the membrane structure in the "structure" tape and, if necessary, we also change the polarization of membrane k.

It is easy to see that the most expensive rules (in terms of time needed to simulate them) are those of type (a) and (d).

To simulate a single step of computation of Π' we need at most $A \cdot B \cdot D$ sums. The time required by each sum depends, of course, on the number of digits of the numbers to sum. Without loss of generality, we can consider a membrane system with a maximum number of initial objects equal to $A \cdot B$ (that is, every membrane contains at most one occurrence of each object in O): given a system with an arbitrary finite number of occurrences of each object in each membrane, we can easily build an equivalent system with exactly one occurrence of each object in each membrane, which requires a bounded number of steps to reach the same configuration as the starting system. Thus, after one step, the total quantity of symbols present in the system will be less than $A \cdot B \cdot C$. After the second step, this quantity will be less than $A \cdot B \cdot C^2$. After t steps, we get no more than $A \cdot B \cdot C^t$ objects. This means that each sum will be made with numbers involving less than $\log(A \cdot B \cdot D^t)$ digits. The time required by a deterministic Turing machine to perform a sum is linear in the number of digits of the numbers involved, thus every sum requires a time which is of the order of t. This means that the time needed to execute the sums for each object from each membrane is of the order of $A \cdot B \cdot D \cdot t$, which is again of the order of t. After the execution of all sums, we have to copy the results from the support tapes to the main tapes; the time needed is again of the order of t. Therefore, the total time requested by M to simulate a single step of Π' is of the order of t.

Consequently, in order to simulate t steps of Π' by the Turing machine M we need a total time of the order of t^2; more rigorously, this number is in $O(A \cdot B \cdot D \cdot t \cdot \log(A \cdot B \cdot C^t))$. □

If the only rules which can change the number of membranes are those which dissolve membranes, then the number of membranes remains the same or at most decreases during a computation, but if also rules for dividing membranes are used, then the number of membranes can increase exponentially. Thus, after t steps we can obtain $A \cdot 2^t$ membranes, and the simulation of such a system by a Turing machine will require the handling of an exponential number of strings, hence the total time will be in $O(A \cdot B \cdot D \cdot t \cdot 2^t \cdot \log(A \cdot B \cdot C^t \cdot 2^t))$, and the previous theorem is no longer obtained for this case.

Very important in the previous proof was the fact that the system we started with was a deterministic one (also, non-cooperative). On the other hand, when we defined the classes $\mathbf{LMC}_X, \mathbf{PMC}_X$, we did not pay much attention to non-determinism – provided that the systems are confluent, in the precise sense defined at the beginning of this chapter. These observations raise the natural question whether or not we can find **NP**-complete problems which are in $\mathbf{PMC}_X$ for classes X of P systems with active membranes which do not use membrane division, hence using only rules of the forms (a), (b), (c), (d) (Problem Q24). The problem is non-trivial, because by means of rules of type (a) we already have the possibility to create exponentially many objects in linear time: consider rules of the form $[_i a \to aa]_i^\alpha$. However, the objects created in this way are placed in the same membrane (or in the bounded number of membranes of the initial configuration of the system), which seems to be a difficulty in using them efficiently. Structuring the multisets of objects in a better manner, for instance by dividing membranes *or* by creating new membranes, produces the extra power sufficient for solving **NP**-complete problems in polynomial time. For membrane division this was illustrated above; the case of creating membranes will be considered in the following section.

7.3 Using Membrane Creation

Membranes are created continuously in biology, for instance in the process of vesicle mediated transport. Also, because one of the roles of membranes is to keep the molecules of a compartment close to each other, in order to facilitate their reactions, when a compartment becomes too large, it often happens that new membranes appear inside it, more or less spontaneously or during biological evolution. Membranes can also be created in a laboratory – see [128] and its bibliography.

Here we consider membrane creation in a way somewhat related to the second case mentioned above: that is, new membranes are produced under the influence of the existing objects. Of course, we abstract the operation, using it in an idealized way, mainly as a tool for better structuring the objects of a given membrane system.

Membrane creation can be introduced both for systems with symbol–objects and for systems with string–objects. We consider the case of symbol–objects first.

In this framework, a rule for membrane creation is of the form $a \to [_i v]_i$, where a is an object, v is a string representing a multiset of objects, and i is a label from a given list of labels for possible membranes. When such a rule is applied in a region j, then the object a is replaced with a new membrane, with the label i and the contents as indicated by v. The fact that we indicate the label i of the new membrane is important, because we know in this way what is the set of rules which can be applied in this membrane.

Together with such rules, we can have the usual rules: cooperative, catalytic, or non-cooperative, with or without further ingredients, such as a priority relation, or membrane thickness control by means of actions δ, τ. Catalysts, priorities, and actions δ, τ can be added also to membrane creation rules – but in what follows we will not need this extension, neither for universality, nor for obtaining polynomial solutions to **NP**-complete problems.

More formally, a P system with membrane creation as we investigate here is a construct $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_n)$, where O is the alphabet of objects, μ is a membrane structure of degree m, with the membranes labelled with $i_1, \ldots, i_m \in \{1, 2, \ldots, n\}$, w_j is the multiset of objects present in region $i_j, 1 \leq j \leq m$, of μ, and $R_1, \ldots, R_n$ are finite sets of rules; there are n possible types of membranes, with labels $1, 2, \ldots, n$, and rules from R_i are associated with membranes with label i.

The rules can be of the forms $u \to v$, or $u \to v\delta$, or $u \to v\tau$, with $u \in O^*$ and $v \in (O \times \{here, out, in\})^*$, or of the form $a \to [_i v]_i$, for $a \in O, v \in O^*, 1 \leq i \leq n$. They are used as usual, in the non-deterministic maximally parallel manner, with the only new detail being that the membranes created at a given step cannot be used at the same step for sending objects to them from the enveloping region; the new membranes become available for communication from the next step on. The result of a computation can be defined both in the internal mode, considering an output membrane, or in the external mode; however, there is a difficulty with the internal mode, because we have to make sure that the output membrane is unique in the halting configuration. This means that we cannot create membranes with the label of the output membrane, or to accept only computations where we end with only one membrane with that label. Because this seems to be a little bit clumsy, and because when solving decidability problems we look for the message *yes* outside the system, we choose here the external output, in the form of the number of objects sent out during a halting computation. We denote by $N(\Pi)$ the set of numbers computed in this way by a system Π, without any restriction on the number of membranes ever appearing during a computation; we also denote by $N_k(\Pi)$ the set of numbers computed by Π by computations whose configurations do not contain more than k membranes.

Clearly, $N_k(\Pi) \subseteq N(\Pi)$ for all $k \geq 1$, and $N_k(\Pi) = \emptyset$ for $k < m$, where m is the number of membranes from the initial configuration.

Let us examine an **example**, both illustrating the definition and proving the usefulness of membrane creation. Consider the system

$$\Pi = (\{a, a', b, c\}, [_1[_2\ ]_2]_1, \lambda, a'c, R_1, R_2, R_3),$$

with the following sets of rules:

$$\begin{aligned} R_1 &= \{a \to a', a' \to (a, in),\ c \to [_3c]_3,\ b \to (a, out)\}, \\ R_2 &= \{a' \to a,\ a \to a^2,\ c \to c,\ c \to c\delta\}, \\ R_3 &= \{a \to b,\ b \to b^3,\ c \to c,\ c \to \delta\}. \end{aligned}$$

(As usual, the indication *here* is omitted.) Initially, we have only two membranes, with only two objects, both of them in membrane 2: a copy of a' and one of c. We produce 2^n copies of a, for some $n \geq 0$, in $n+1$ steps; in the first step we pass from a' to a, in the last one the membrane is dissolved, and 2^n copies of a and one copy of c are left free in membrane 1. In the next step, each a is replaced by a' and, simultaneously, a copy of membrane 3 is produced. All objects are sent to this membrane, where first we replace a by b, then we multiply the number of objects b by 3, repeatedly. At any moment, membrane 3 is dissolved. At the next step, all copies of b are sent out of the system.

Consequently, $N(\Pi) = \{2^n 3^m \mid n, m \geq 0\}$. Because we always have at most two membranes in our system, we have $N(\Pi) = N_2(\Pi)$.

We denote by $NOP_{n_1,n_2,n_3}(ncoo, in, mcre)$ the family of sets of numbers $N(\Pi)$ generated by P systems with membrane creation, using non-cooperative rules and target commands of the forms *here, out, in*, having at most n_1 membranes in the initial configuration, using at most n_2 membranes in any configuration, of at most n_3 types. If any of the parameters n_1, n_2, n_3 is not bounded in advance, then we replace it with $*$. Instead of *ncoo, in* we can use other parameters, as suitable; similarly, further features can be added, such as pri, δ, τ, etc.

If Π has n_1 initial membranes, and it uses at any time at most n_2 membranes, of n_3 types in total, then (n_1, n_2, n_3) is called the *profile* of Π. Note that if Π has the profile (n_1, n_2, n_3), then $N_{n_2}(\Pi) = N(\Pi)$.

Obviously, a system Π of degree m without membrane creation is also a system with membrane creation, of profile (m, m, m), hence inclusions of the form $NOP_m(\ldots) \subseteq NOP_{m,m,m}(\ldots, mcre)$ follow from the definitions (actually, we have also to pass from the internal mode of defining the result of a computation to the external one, but this can be easily done, as already explained in Section 3.7; see the proof of Lemma 3.7.1). Such inclusions imply the universality of all classes of systems with membrane creation for which the corresponding families $NOP_m(\ldots)$ equal NRE. This observation can be strengthened as follows.

Lemma 7.3.1. *Given a P system of degree m, without membrane creation, of any type, one can construct an equivalent P system with membrane creation, with the profile* $(1, m, m)$.

Proof. According to Lemma 3.7.1, it is sufficient to start from a system with external output. Then, the idea is that given a system $\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m)$, with external output, and with the membrane structure μ of *height* h (there are h levels of membranes, counting both the external membrane and the elementary membranes), we can construct an equivalent system Π', with only one initial membrane, which in the first phase of a computation constructs the membrane structure μ and the multisets $w_1, \ldots, w_m$, by making use of the membrane creation. This phase takes $2h$ steps. After completing it, the system obtained is exactly Π, hence the equivalence follows.

Here are a few details of this construction. The initial configuration of Π' is $[_1 d_1^{(0)}]_1$ (note that 1 is assumed to be the label of the skin membrane of Π and that the membranes of Π are labelled in a one-to-one manner). To each set $R_i, 1 \leq i \leq m$, we add the following rules:

$$\begin{aligned}
&d_i^{(j)} \to c_{k_1}^{(j+1)} \ldots c_{k_i}^{(j+1)} e_i^{(j+1)}, \ 0 \leq j \leq 2h-1, \\
&e_i^{(j)} \to e_i^{(j+1)}, \ 1 \leq j \leq 2h-1, \\
&e_i^{(2h-1)} \to w_i,
\end{aligned}$$

where $k_1, \ldots, k_i$ are the labels of membranes placed inside the membrane with label i (if there is no such membrane, then the rule becomes $d_i^{(j)} \to e_i^{(j+1)}$). Moreover, if $up(i)$ is the membrane placed immediately above membrane i, then in $R_{up(i)}$ we also add the rules

$$c_i^{(j)} \to [_i d_i^{(j+1)}]_i, \ 1 \leq j \leq 2h-1.$$

The objects $d_i^{(j)}$ introduce the membrane creating objects c and the counters e; all these objects have superscripts indicating the step of the computation, so that the process lasts exactly as needed, namely $2h$ steps, while both the branching and the creation of membranes are accomplished – in the last step one also introduces the multisets $w_i, \ldots, w_m$ in the corresponding membranes.

The system obtained is equivalent to the initial one, and this is true independently of any other ingredients used by Π, as the initial sets of rules $R_1, \ldots, R_m$ were just augmented with the above-mentioned rules, which are non-cooperating, and use no further feature. □

By combining this result with the characterizations of NRE from Theorems 3.4.3, 3.4.7, 3.6.1, etc., we can obtain universality results for systems with membrane creation with rather low profiles. We mention only two such results.

Corollary 7.3.1. $NOP_{1,2,2}(cat, in, pri, mcre) = NOP_{1,4,4}(cat, in, \delta, \tau, mcre) = NRE$.

Before considering the usefulness of membrane creation from the point of view of the computational efficiency, let us mention one interesting result, giving a characterization of the length sets of ET0L languages. Note that in Theorem 3.4.4 we have $NET0L \subseteq NOP_1(ncoo, pri)$ while in Theorem 3.4.2 we have $NOP_*(ncoo, tar, \delta) \subseteq NET0L$, but we do not know whether the converse inclusions also hold. The only characterization of $NET0L$ given in the previous chapters appears in Corollary 6.3.3, but in a completely different framework (using neural-like systems, where membranes also have associated states, essentially used in the proofs).

Theorem 7.3.1. $NOP_{2,2,3}(ncoo, in, \delta, mcre) = NET0L$.

Proof. (i) $NET0L \subseteq NOP_{2,2,3}(ncoo, in, \delta, mcre)$.

Consider an ET0L system with two tables, $G = (V, T, w, R_1, R_2)$. From the way the two-table normal form for ET0L systems is found – see the proof of Theorem II.1.3 from [218] – we find that after each use of the table R_1 we either use the table R_1 again or we use the table R_2, but after each use of the table R_2 we always use the table R_1; at the first step of a derivation, we use the table R_1. Bearing in mind this observation, we construct a membrane system as follows.

Denote $V' = \{a' \mid a \in V\}, V'' = \{a'' \mid a \in V\}$ and define the morphism h by $h(a) = a', a \in V$. Then, let us consider the system

$$\Pi = (V \cup V' \cup V'' \cup \{c\}, [_1[_2]_2]_1, \lambda, cw, R_1', R_2', R_3'),$$

with the folowing sets of rules:

$$\begin{aligned}
R_1' &= \{a \to h(v) \mid a \to v \in R_2\} \\
&\quad \cup \{a' \to (a, in) \mid a \in V\} \\
&\quad \cup \{a'' \to (a, out) \mid a \in T\} \\
&\quad \cup \{a'' \to a'' \mid a \in V - T\} \\
&\quad \cup \{c \to [_2c]_2,\ c \to [_3c]_3\}, \\
R_2' &= R_1 \cup \{c \to c,\ c \to c\delta\}, \\
R_3' &= \{a \to a'' \mid a \in V\} \\
&\quad \cup \{c \to \delta\}.
\end{aligned}$$

This system works as follows. In the initial configuration we have only two membranes, $\mu = [_1[_2\ ]_2]_1$. The table R_1 is simulated in membrane 2 any number of times; when the rule $c \to c\delta$ is used, the membrane is dissolved and its contents are left free in membrane 1. We simulate here the use of the table R_2 (all symbols are primed during this simulation) and, at the same time, either a membrane 2 or a membrane 3 is created by c. At the next step,

all objects are sent to the newly created membranes (without primes). If the created membrane was 2, then the process can be iterated. In this way, we can simulate any derivation in G, and this is correct because of the parallel mode of using the rules in the ET0L system G and the membrane system Π and because of the way of using the tables of G (after using R_2 we always pass to using R_1 at least once).

If we introduce the membrane with label 3, then all objects sent to it are doubly primed and at the same time the membrane is dissolved. In the skin membrane, the double primed symbols either are sent out – provided that they are from T – or can evolve for ever – in the case that they are from $V-T$. In this way, we check whether or not the derivation in G was terminal; in the latter case, the computation in Π never stops.

Consequently, $N(\Pi) = length(L(G))$. Because at each step there are at most two membranes present in our system, we obtain $N(\Pi) = N_2(\Pi) \in NOP_{2,2,3}(ncoo, in, \delta, mcre)$.

(ii) $NOP_{2,2,3}(nCoo, in, \delta, mcre) \subseteq NET0L$.

Actually, we will prove the general inclusion $NOP_{n_1,n_2,n_3}(ncoo,\ in, \delta,\ mcre) \subseteq NET0L$, for arbitrary n_1, n_2, and n_3. The proof will be a strengthening of the proof of Theorem 3.4.2, also taking into account the membrane creation. Essential to this proof will be the fact that at any moment in a computation we have at most n_2 membranes in the system.

Let us consider a P system $\Pi = (O, \mu, w_1, \ldots, w_{n_1}, R_1, \ldots, R_{n_3})$, for some $n_1 \geq 1$ and $n_3 \geq 1$. Our goal is to construct an ET0L system G such that $length(L(G)) = N_{n_2}(\Pi)$, for some given $n_2 \geq 1$. To this end, the following notations are useful.

In order to distinguish the copies of the same membrane which may be simultaneously present in a configuration, we will label them with pairs (i, j) of integers, such that $1 \leq i \leq n_3$ identifies the type of the membrane and $1 \leq j \leq n_2$ identifies the copy of the membrane of type i. The skin membrane will always be labelled with $(1, 1)$ and only one copy of this membrane is present.

Consider all symbols $e_{i,j}^{i',j'}$, for $1 \leq i, i' \leq n_3$ and $1 \leq j, j' \leq n_2$, as well as the symbol $e_{1,1}^{\infty,\infty}$. These symbols identify each membrane by the pair (i, j) and the membrane directly containing membrane (i, j); in the case of the skin membrane, we consider the outside region, identified by (∞, ∞), as the covering "membrane".

We denote by E the set of all symbols of this form.

Consider now the language F over E consisting of all strings u such that $|u| \leq m$, associated with all possible membrane structures consisting of at most m membranes, labelled with (i, j) as above, such that the labels are distinct (if two copies of a membrane of the same type i are present, then one is labelled with (i, j) and the other one with (i, j'), for $j \neq j'$). For instance, for the membrane structure described by the string

$$[_{(1,1)}[_{(2,1)}\ ]_{(2,1)}[_{(3,2)}\ ]_{(3,2)}[_{(4,3)}[_{(2,2)}\ ]_{(2,2)}]_{(4,3)}]_{(1,1)}$$

a possible string of this type is

$$e_{2,1}^{1,1} e_{1,1}^{\infty,\infty} e_{4,3}^{1,1} e_{3,2}^{1,1} e_{2,2}^{4,3}.$$

Note that any permutation of the same string in F determines the same membrane structure and, indeed, such a string precisely identifies a membrane structure, because the subscripts and the superscripts of symbols in E correspond to edges in the tree associated with the membrane structure. For each $u \in F$ we denote by $\overline{u}$ the class of all permutations of u; let $\overline{F}$ be the set of all classes $\overline{u}$, for $u \in F$.

Note also that not all strings in E^* of length at most n_2 are in F; for instance, if $e_{i,j}^{i',j'}$ appear in a string, and $(i',j') \neq (\infty,\infty)$, then also $e_{i',j'}^{i'',j''}$ should appear, for some i'', j''.

Consider also the sets

$$\begin{aligned} C &= \{c_{i,j} \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2\}, \\ C' &= \{c'_{i,j} \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2\}, \\ D &= \{d_{i,j} \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2\}, \\ D' &= \{d'_{i,j} \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2\}. \end{aligned}$$

The elements of C and D identify all possible membranes in a membrane structure. When some $c_{i,j}$ (resp., $d_{i,j}$) are replaced by $c'_{i,j}$ (resp., $d'_{i,j}$), this will indicate the fact that a membrane with the label (i,j) was created (resp., dissolved).

Let $u \in F$ describe a membrane structure μ_u and let $v \in D'^*$ be a string such that for each $d'_{i,j}$ appearing in v, the symbol $e_{i,j}^{i',j'}$ appears in u (we say that *v identifies a substructure of u*). Let us interpret the string v as identifying membranes of μ_u which are dissolved. The membrane structure obtained from the membrane structure described by u by dissolving the membranes identified by v is described by a string z in F; we denote the class $\overline{z}$ by $f(u,v)$ (that is, f is an operator which associates with any membrane structure and with any set of dissolved membranes from this structure a description of the resulting membrane structure).

Consider now the objects appearing in the regions of Π. If an object $a \in V$ is present in the region of a membrane labelled with (i,j), then we also mark the object with (i,j), and write $a^{(i,j)}$.

For $u \in F$, $v \in D'^*$ such that v identifies a substructure of u, and for (i,j) such that there is a symbol $e_{i,j}^{i',j'}$ in u, we denote by $g(i,j;u,v)$ the pair (i'',j'') of integers which identify the region of the membrane structure identified by $f(u,v)$ where the objects from membrane (i,j) of u will be placed after dissolving the membranes identified by v. Clearly, $g(i,j;u,\lambda) = (i,j)$.

Note that by knowing u and v we know precisely the place of all objects from the membrane structure described by u after dissolving the membranes indicated by v.

Let u_0 be the string in F which identifies the initial membrane structure, μ, of Π, let w_0 be the concatenation of the strings describing the initial multisets of Π, after replacing each object a by $a^{(i,j)}$, according to the region (i,j) where a is placed, and let z_C, z_D be the concatenation of all symbols from sets C, D, respectively.

We now proceed to define the ET0L system G we look for.

Its total alphabet is

$$\begin{aligned} &T \cup C \cup C' \cup D \cup D' \cup \{\langle \overline{u} \rangle \mid u \in F\} \cup \{X, \#\} \\ \cup\ &\{a^{(i,j)} \mid a \in V, 1 \leq i \leq n_3, 1 \leq j \leq n_2\}, \end{aligned}$$

the terminal alphabet is

$$T = \{a \in O \mid \text{there is a rule } b \to u \in R_1 \text{ such that } |u|_{(a,out)} \geq 1\}$$

(X and $\#$ are new symbols; $\#$ is a trap symbol, which can never be removed), and the axiom is

$$X w_0 \langle \overline{u_0} \rangle z_C z_D,$$

while the set of tables is constructed as follows (for each table we specify only the rules of interest for our proof, but not the completion rules for symbols α for which no rule of the form $\alpha \to x$ is already given; that is, rules of the form $\alpha \to \alpha$ should be added to all tables when necessary):

1. For each $u \in F$ we consider the following table:

$$\begin{aligned} R_u = \ &\{X \to X', X' \to \#, \langle \overline{u} \rangle \to \langle \overline{u} \rangle\} \\ \cup\ &\{\langle \overline{u'} \rangle \to \# \mid \overline{u'} \in \overline{F} - \{\overline{u}\}\} \\ \cup\ &\{\alpha \to \# \mid \text{ for all } \alpha \text{ in } D' \cup C'\} \\ \cup\ &\{a^{(i,j)} \to v' \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2, \\ &\quad a \to v \in R_i, e_{i,j}^{i'j'} \text{ appears in } u \text{ for some } i', j', \\ &\quad \text{and } v' \text{ is obtained from } v \text{ in the following way:} \\ &\qquad \text{if } (b, here) \text{ appears in } v, \text{ then we put } b^{(i,j)} \text{ in } v', \\ &\qquad \text{if } (b, out) \text{ appears in } v, \text{ then we put } b^{(i',j')} \text{ in } v', \\ &\qquad \text{if } (b, in) \text{ appears in } v, \text{ then we put } b^{(i'',j'')} \text{ in } v' \\ &\qquad\quad \text{for some } (i'', j'') \text{ associated with a membrane} \\ &\qquad\quad \text{which is directly inside membrane } (i,j), \\ &\quad \text{but if } (b, out) \text{ appears in } v \text{ and we have} \\ &\quad (i,j) = (1,1), \text{ then we put } b \text{ in } v'\} \\ \cup\ &\{a^{(i,j)} \to v', \\ &\quad d_{i,j} \to d'_{i,j} \mid 1 \leq i \leq n_3, 1 \leq j \leq n_2, \\ &\quad a \to v\delta \in R_i, e_{i,j}^{i'j'} \text{ appears in } u \text{ for some } i', j', \end{aligned}$$

and v' is obtained from v in the same way as above}
$\cup \{a^{(i,j)} \to v^{(k,l)} e^{i,j}_{k,l},$
$c_{k,l} \to c'_{k,l} \mid 1 \le i \le n_3, 1 \le j \le n_2,$
$a \to [_k v]_k \in R_i$, for some (k,l) which does not appear as a superscript in u and $v^{(k,l)}$ is obtained by replacing each symbol b which appears in v by the symbol $b^{(k,l)}\}$.

2. For each $u \in F, z \in D'^*$, and $y \in C'^*$ such that $f(u,z)y \in F$ we consider the following table (u indicates existing membranes, z indicates membranes of u which were dissolved at the previous step, and y indicates membranes which were created, hence they are added to membranes of u after removing the membranes indicated by z):

$$\begin{aligned} R_{u,z,y} = \{&X' \to X, X \to \#, \langle\overline{u}\rangle \to \langle\overline{f(u,z)y}\rangle\} \\ &\cup \{\langle\overline{u'}\rangle \to \# \mid \overline{u'} \in \overline{F} - \{\overline{u}\}\} \\ &\cup \{a^{(i,j)} \to a^{g(i,j;u,z)} \mid 1 \le i \le n_3, 1 \le j \le n_2, a \in V, \\ &\quad \text{for each } (i,j) \text{ which appears as a subscript in } u\} \\ &\cup \{d'_{i,j} \to d_{i,j}, d_{i,j} \to \# \mid \text{for } d'_{i,j} \text{ appearing in } z\} \\ &\cup \{d'_{i,j} \to \# \mid \text{for } d'_{i,j} \text{ not appearing in } z\} \\ &\cup \{c'_{i,j} \to c_{i,j}, c_{i,j} \to \# \mid \text{for } c'_{i,j} \text{ appearing in } y\} \\ &\cup \{c'_{i,j} \to \# \mid \text{for } c'_{i,j} \text{ not appearing in } y\}. \end{aligned}$$

3. For each $u \in F$ we consider the following table:

$$\begin{aligned} R'_u = \{&X \to \lambda, X' \to \#, \langle\overline{u}\rangle \to \lambda\} \\ &\cup \{\langle\overline{u'}\rangle \to \# \mid \overline{u'} \in \overline{F} - \{\overline{u}\}\} \\ &\cup \{\alpha \to \# \mid \text{ for all } \alpha \text{ in } C' \cup D'\} \\ &\cup \{a^{(i,j)} \to a \mid (i,j) \text{ appears as a subscript in } u \\ &\quad \text{and no rule } a \to v \text{ exists in } R_i\} \\ &\cup \{a^{(i,j)} \to \# \mid (i,j) \text{ appears as a subscript in } u \\ &\quad \text{and there is a rule } a \to v \text{ in } R_i\} \\ &\cup \{d_{i,j} \to \lambda, c_{i,j} \to \lambda \mid 1 \le i \le n_3, 1 \le j \le n_2\}. \end{aligned}$$

This ET0L system works as follows.

The non-terminal symbols $\langle\overline{u}\rangle$, for $u \in F$, precisely identify the membrane structure. The symbols in D and C are associated with all possible copies of membranes, while the symbols in D' and C' indicate the membranes which are deleted, respectively created, at a given step of a computation. Initially, all symbols from D and C are present in the axiom of G, together with a description of the initial membrane structure and of the initial multiset. It is also present in the control symbol X.

In the presence of X only tables of the form R_u, R'_u can be used (if a rule $X' \to \#$ is used, then the derivation will never be a terminal one, because $\#$ can never be eliminated).

The tables of the form R_u simulate the transitions in the system Π. Because each object a present in the region of a membrane (i,j) is present in G in the form $a^{(i,j)}$, we know precisely the rules which can be applied to this object, namely those from R_i. The application of such rules is simulated in such a way as to take care of the communication commands *here, out, in*, the dissolving actions, and the membrane creation. Note that the string u contains the necessary information for correctly handling the indication *in*: we know which of the membranes are placed inside the membrane where we work, hence we can choose one of them, non-deterministically. The membranes which are dissolved are identified with symbols from D' and the membranes which are created are identified with symbols from C'.

At the next step, we have to use a table of the type $R_{u,z,y}$, for $u \in F$ identifying the existing membranes, $z \in D'^*$ identifying the dissolved membranes, and $y \in C'^*$ identifying the created membranes. This table will check whether or not the strings z, y correctly indicate the dissolved and the created membranes (in the negative case, the symbol $\#$ will be introduced) and it also computes the new string from F which identifies the current membrane structure, and the new superscripts (i', j') of objects. This is done by means of the operators f and g defined at the beginning of the proof.

Because X' is again replaced by X, we can iterate this procedure, hence any computation in Π can be simulated by a derivation in G.

At any moment, we can use a "terminal" table R'_u, for some $u \in F$. It erases both the description u of the membrane structure and the symbol X. We also check whether or not we have reached a halting configuration in Π: if any further rule can be applied in Π to the multisets corresponding to the obtained sentential form of G, then the trap symbol $\#$ is introduced. All non-terminals in $D \cup C$ are removed.

If the derivation is terminal, that is no occurrence of $\#$ is present in the string x obtained in this way, then $|x|$ is exactly the number computed by Π: each object $a \in V$ which is sent out of the skin membrane during the computation in Π is introduced by a table of type R_u in the form a, and the only rules which process such a symbol are the completion rules $a \to a$; no other symbol is sent out by Π, hence no other symbol appears in the final string generated by G.

In conclusion, $length(L(G)) = N_2(\Pi)$, and this completes the proof. □

It is easy to see that the previous result, even with the stronger inclusion $NOP_{n_1,n_2,n_3}(ncoo, in, \delta, mcre) \subseteq NET0L$, also holds for the indication *in* replaced with *tar*. However, it does not hold if we add a priority relation (and still using two membranes at a time), or if we also add the action τ (and we use at least four membranes at a time), because in those cases we get universality. This observation is rather interesting in view of looking for "the

borderline between universality and non-universality", between the combinations of features sufficient for obtaining universality and the combinations which are not sufficient.

We do not persist with this direction here, but pass to addressing **NP**-complete problems by means of P systems with membrane creation.

7.3.1 Solving SAT

Let us first consider an easy way to obtain the solution to SAT, which, however, is not completely satisfactory from the point of view of the criteria discussed at the beginning of this chapter – specifically, the systems we obtain will not be of a polynomial size with respect to the size of the instance of SAT.

Let $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ be a propositional formula with each clause $C_i, 1 \leq i \leq m$, of the form $C_i = y_{i,1} \vee y_{i,2} \vee \ldots \vee y_{i,p_i}$, where each literal $y_{i,j}$ is either a propositional variable, x_s, or its negation, $\neg x_s$.

Let us consider the P system with the alphabet of objects

$$O = \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{b, yes\} \cup \{\langle D \rangle \mid D \subseteq \{1, 2, \ldots, m\}\}$$

(note that the symbols of the form $\langle D \rangle$ are considered symbol–objects, although D is a set of numbers, any subset of $\{1, 2, \ldots, m\}$), having initially only one membrane, $[_1\ ]_1$, with label 1 and containing the symbol–object a_1. The following sets of rules are possible – implicitly, in this way we give the set H of possible labels for the corresponding membranes:

$$\begin{aligned}
R_1 = \{&a_1 \to t_1 f_1,\\
&t_1 \to [_{t_1} a_2]_{t_1},\\
&f_1 \to [_{f_1} a_2]_{f_1},\\
&\langle\{1, 2, \ldots, m\}\rangle \to (yes, out)\},\\
R_{t_i} = \{&a_{i+1} \to t_{i+1} f_{i+1},\\
&t_{i+1} \to [_{t_{i+1}} a_{i+2}]_{t_{i+1}},\\
&f_{i+1} \to [_{f_{i+1}} a_{i+2}]_{f_{i+1}}\}\\
&\cup \{\langle D \rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } x_i\}\rangle, out)\},\\
R_{f_i} = \{&a_{i+1} \to t_{i+1} f_{i+1},\\
&t_{i+1} \to [_{t_{i+1}} a_{i+2}]_{t_{i+1}},\\
&f_{i+1} \to [_{f_{i+1}} a_{i+2}]_{f_{i+1}}\}\\
&\cup \{\langle D \rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } \neg x_i\}\rangle, out)\},\\
&\text{for all } i = 1, 2, \ldots, n-2,\\
R_{t_{n-1}} = \{&a_n \to t_n f_n,\\
&t_n \to [_{t_n} b]_{t_n},\\
&f_n \to [_{f_n} b]_{f_n}\}
\end{aligned}$$

$$
\begin{aligned}
&\cup \{\langle D\rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } x_{n-1}\}\rangle, out)\},\\
R_{f_{n-1}} &= \{a_n \to t_n f_n,\\
&\quad t_n \to [_{t_n} b]_{t_n},\\
&\quad f_n \to [_{f_n} b]_{f_n}\}\\
&\cup \{\langle D\rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } \neg x_{n-1}\}\rangle, out)\},\\
R_{t_n} &= \{b \to (\langle\{j \mid \text{clause } C_j \text{ contains } x_n\}\rangle, out)\},\\
R_{f_n} &= \{b \to (\langle\{j \mid \text{clause } C_j \text{ contains } \neg x_n\}\rangle, out)\}.
\end{aligned}
$$

In each membrane, starting with the initial one, we introduce two objects, t_i and f_i, which will create two new membranes, with the labels t_i, f_i, respectively, and the same contents, the object a_{i+1}. In this way, a membrane structure will grow, by creating in each step two membranes in each elementary membrane, labelled with the truth values for variables. Creating one more level of membranes takes two steps. Thus, after $2n$ steps, we get a membrane structure with $n+1$ levels (we also count the skin membrane, the one labelled with 1), described by a complete binary tree such that the paths from leaves to the root correspond to the 2^n truth-assignments for the variables $x_1, \ldots, x_n$. After completing this operation, in the elementary membranes, those labelled either with t_n or with f_n, we have the object b, which will send to the upper membrane an object of the form $\langle D\rangle$, where D is the set of subscripts of clauses which are satisfied by the truth value which labels the corresponding membrane. The objects $\langle D\rangle$ will go towards the skin membrane, at each step collecting the subscripts of clauses satisfied by the truth value which labels the membrane where we are. In this way, after n steps, we arrive in the skin membrane with 2^n objects of the form $\langle D\rangle$, possibly with multiple copies of some of them. However, only the copies of the object $\langle\{1, 2, \ldots, m\}\rangle$ can send the message *yes* out of the system. If this happens, then we know that C is satisfiable, otherwise we know that C is not satisfiable.

In $3n+1$ steps we have the solution to our instance of SAT, but there is a problem here: there are exponentially many rules of the forms $\langle D\rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } x_i\}\rangle, out)$ and $\langle D\rangle \to (\langle D \cup \{j \mid \text{clause } C_j \text{ contains } \neg x_i\}\rangle, out)$. We can handle this point in various ways: (a) by claiming that the form of rules is very uniform, and strings of the form $xcxd$, which approximately codify such rules, can be recognized by a Turing machine in polynomial time, hence we also accept such a family of P systems as being uniform; (b) by observing that not all rules written when giving the system are effectively used; actually, in some sense, the rules themselves can be created only when needed, hence passsing from a level of the membrane structure to the upper level by increasing the "size" of objects $\langle D\rangle$ received from the lower levels; the objects coming from the lower levels together with the label of the membrane where we work precisely identify a *unique* rule to be used at that moment, in that place of the membrane structure; (c) by

looking for a different construction, which keeps the size of the system under control.

From a "practical" point of view, the two strategies (a) and (b) can be satisfactory enough, but not from a theoretical point of view. However, we do not know (Problem Q25) whether or not we can construct a uniform family of P systems (in the sense of the first section of this chapter), with membrane creation of the type considered above, hence without any further feature such as priorities, or membrane thickness control, for solving SAT in linear time.

Such a solution can be obtained if we can make use of the addressing by commands in_j and actions δ, τ, too.

Let us denote by C the class of P systems with membrane creation using non-cooperative rules, and the previously mentioned features, in_j, δ, τ.

Theorem 7.3.2. SAT $\in$ **LMC**$_C$.

Proof. Consider again an instance $C = C_1 \wedge C_2 \wedge \ldots \wedge C_m$, of SAT, with $C_i = y_{i,1} \vee y_{i,2} \vee \ldots \vee y_{i,p_i}$, and $y_{i,j} \in \{x_s, \neg x_s \mid 1 \leq s \leq n\}$, $1 \leq j \leq p_i, 1 \leq i \leq m$. We construct a P system which is similar to the system considered above, but checking the satisfiability of the formula C in a different way.

Specifically, we consider the system Π with the alphabet of objects

$$\begin{aligned} O = &\{a_i, a_i', t_i, f_i \mid 1 \leq i \leq n\} \cup \{b, b', d, d', yes\} \\ &\cup \{c_i, \bar{c}_i, c_i', c_i'' \mid 1 \leq i \leq m\} \cup \{b_i \mid 0 \leq i \leq m\}, \end{aligned}$$

one initial membrane with label 1, $[_1\ ]_1$, where we have the object a_1, and the following types of membranes – given directly by means of their associated sets of rules:

$$\begin{aligned} R_1 : a_1 &\to t_1 f_1, \\ &t_1 \to [_{t_1} a_2 c_{j_1}' \ldots c_{j_k}']_{t_1}, \text{ with } C_{j_1}, \ldots, C_{j_k} \text{ being} \\ &\text{the clauses of } C \text{ which contain the literal } x_1, \\ &f_1 \to [_{f_1} a_2 c_{l_1}' \ldots c_{l_r}']_{f_1}, \text{ with } C_{l_1}, \ldots, C_{l_r} \text{ being} \\ &\text{the clauses of } C \text{ which contain the literal } \neg x_1, \\ &b_m \to (yes, out), \\ R_{t_1} = R_{f_1} : a_2 &\to t_2 f_2, \\ &t_2 \to [_{t_2} a_3 \bar{c}_{j_1} \ldots \bar{c}_{j_k}]_{t_2}, \text{ with } C_{j_1}, \ldots, C_{j_k} \text{ being} \\ &\text{the clauses of } C \text{ which contain the literal } x_2, \\ &f_2 \to [_{f_2} a_3 \bar{c}_{l_1} \ldots \bar{c}_{l_r}]_{f_2}, \text{ with } C_{l_1}, \ldots, C_{l_r} \text{ being} \\ &\text{the clauses of } C \text{ which contain the literal } \neg x_2, \\ &b_m \to (b_m, out), \\ &c_j' \to c_j'', \\ &c_j'' \to c_j, \end{aligned}$$

$$
\begin{aligned}
& c_j \to (c'_j, in_{t_2})(c'_j, in_{f_2}), \text{ for } 1 \le j \le m, \\
R_{t_i} = R_{f_i} : & a_{i+1} \to a'_{i+1}, \\
& a'_{i+1} \to t_{i+1} f_{i+1}, \\
& t_{i+1} \to [_{t_{i+1}} a_{i+2} \bar{c}_{j_1} \ldots \bar{c}_{j_k}]_{t_{i+1}}, \text{ with } C_{j_1}, \ldots, C_{j_k} \text{ being} \\
& \text{the clauses of } C \text{ which contain the literal } x_{i+1}, \\
& f_{i+1} \to [_{f_{i+1}} a_{i+2} \bar{c}_{l_1} \ldots \bar{c}_{l_r}]_{f_{i+1}}, \text{ with } C_{l_1}, \ldots, C_{l_r} \text{ being} \\
& \text{the clauses of } C \text{ which contain the literal } \neg x_{i+1}, \\
& b_m \to (b_m, out), \\
& \bar{c}_j \to c'_j, \\
& c'_j \to c''_j, \\
& c''_j \to c_j, \\
& c_j \to (c'_j, in_{t_{i+1}})(c'_j, in_{f_{i+1}}), \text{ for } 1 \le j \le m, \\
& \text{for all } i = 2, 3, \ldots, n-2, \\
R_{t_{n-1}} = R_{f_{n-1}} : & a_n \to a'_n, \\
& a'_n \to t_n f_n, \\
& t_n \to [_{t_n} b \bar{c}_{j_1} \ldots \bar{c}_{j_k}]_{t_n}, \text{ with } C_{j_1}, \ldots, C_{j_k} \text{ being} \\
& \text{the clauses of } C \text{ which contain the literal } x_n, \\
& f_n \to [_{f_n} b \bar{c}_{l_1} \ldots \bar{c}_{l_r}]_{f_n}, \text{ with } C_{l_1}, \ldots, C_{l_r} \text{ being} \\
& \text{the clauses of } C \text{ which contain the literal } \neg x_n, \\
& b_m \to (b_m, out), \\
& \bar{c}_j \to c'_j, \\
& c'_j \to c''_j, \\
& c''_j \to c_j, \\
& c_j \to (c'_j, in_{t_n})(c'_j, in_{f_n}), \text{ for } 1 \le j \le m, \\
R_{t_n} = R_{f_n} : & b \to [_2 b']_2, \\
& \bar{c}_j \to c'_j, \\
& c'_j \to (c_j, in_1), \text{ for } 1 \le j \le m, \\
& d \to d' \tau, \\
& b_m \to (b_m, out), \\
R_2 : & b' \to b_0 d, \\
& c_1 \to d\tau, \\
& b_i \to b_{i+1} \delta, \text{ for all } 1 \le i \le m-1, \\
& c_j \to c_{j-1}, \text{ for all } 2 \le i \le m, \\
& b_m \to (b_m, out).
\end{aligned}
$$

The generation of truth–assignments takes place in a way similar to that in the construction discussed before the theorem, with the creation of a mem-

brane structure corresponding to a complete binary tree ending with 2^n leaf nodes – the membranes from level n, labelled with t_n or f_n, have one further membrane inside, with label 2. The difference from the first construction is that at each step we also produce all objects $c_j, 1 \leq j \leq m$, such that the associated clauses C_j are satisfied by the label of the membrane just created. These objects are then replicated and introduced in the two new membranes which are created in the immediately lower level, then replicated again in the next membranes, and so on until arriving in the elementary membranes, those labelled with 2.

Let us examine this process in some detail, on the one hand in order to check whether it works in the appropriate way, on the other hand in order to evaluate the number of steps a computation has.

We start from $[_1 a_1]_1$ and we get $[_1 t_1 f_1]_1$. Each of the new objects t_1, f_1 creates a membrane with its label:

$$[_1 [_{t_1} a_2 c'_{j_1} \ldots c'_{j_k}]_{t_1} [_{f_1} a_2 c'_{l_1} \ldots c'_{l_r}]_{f_1}]_1 .$$

The mentioned clauses are those satisfied by t_1 (as a notation for "$x_1 = true$") and f_1 (as a notation for "$x_1 = false$"), respectively. In the new membranes we use at the same time the rules $a_2 \to t_2 f_2$ and $c'_j \to c''_j$, for all j, hence we obtain

$$[_1 [_{t_1} t_2 f_2 c''_{j_1} \ldots c''_{j_k}]_{t_1} [_{f_1} t_2 f_2 c''_{l_1} \ldots c''_{l_r}]_{f_1}]_1 .$$

Now, we create new membranes from t_2 and f_2, and we erase the primes:

$$\begin{aligned} &[_1 [_{t_1} [_{t_2} a_3 \bar{c}_{p_1} \ldots \bar{c}_{p_u}]_{t_2} [_{f_2} a_3 \bar{c}_{q_1} \ldots \bar{c}_{q_v}]_{f_2} c_{j_1} \ldots c_{j_k}]_{t_1} \\ &\quad [_{f_1} [_{t_2} a_3 \bar{c}_{p_1} \ldots \bar{c}_{p_u}]_{t_2} [_{f_2} a_3 \bar{c}_{q_1} \ldots \bar{c}_{q_v}]_{f_2} c_{l_1} \ldots c_{l_r}]_{f_1}]_1 . \end{aligned}$$

In the membranes with label t_2 we have the objects $\bar{c}_j$ for the clauses satisfied by t_2, and in the membranes with label f_2 we have the objects $\bar{c}_j$ for the clauses satisfied by f_2. At the next step, all a_3 become a'_3, the barred objects from the newly created membranes become primed, while the objects c_j from membranes created at the previous step, those with labels t_1, f_1, are replicated and enter the new membranes (in primed forms). That is, we get the following configuration of the system:

$$\begin{aligned} &[_1 [_{t_1} [_{t_2} a'_3 c'_{p_1} \ldots c'_{p_u} c'_{j_1} \ldots c'_{j_k}]_{t_2} [_{f_2} a'_3 c'_{q_1} \ldots c'_{q_v} c'_{j_1} \ldots c'_{j_k}]_{f_2}]_{t_1} \\ &\quad [_{f_1} [_{t_2} a'_3 c'_{p_1} \ldots c'_{p_u} c'_{l_1} \ldots c'_{l_r}]_{t_2} [_{f_2} a'_3 c'_{q_1} \ldots c'_{q_v} c'_{l_1} \ldots c'_{l_r}]_{f_2}]_{f_1}]_1 . \end{aligned}$$

The process can be iterated: at the next step we apply the rule $a'_3 \to t_3 f_3$ at the same time as $c'_j \to c''_j$; after that we create new membranes from the objects t_3, f_3 (containing a'_4 and barred objects c_j), and at the same time c''_j become c_j, at the next step a'_4 loses the prime, the barred objects c_j replace the bar with a prime, and the objects c_j from outside the last membranes are replicated and introduced inside, primed.

In each of the membranes with labels $1, t_1$, and f_1 we have worked two steps; after that we work three steps at each level. Three steps are also used in membranes with labels t_{n-1}, f_{n-1}, ending with the creation of membranes with labels t_n, f_n, containing the object b (and barred objects c_j, corresponding to the clauses satisfied by t_n and f_n).

Assume that we are in a membrane in this last level. In one step, the bars of objects c_j from our membrane are replaced by primes, at the same time the objects c_j from the previous level enter the membrane, while b creates a new membrane, by means of the rule $b \to [_2 b']_2$. In the next step, all objects c_j enter this membrane with label 2 and b' introduces the objects b_0 and d.

Thus, in the inner membranes we have collected all clauses from the path starting in that membrane and ending in the root of the tree describing the membrane structure. We have to see whether or not at least one of these sets of clauses equals the set of all clauses; that is, whether or not there is a membrane with label 2 containing all objects $c_1, \ldots, c_m$. This is done by the interplay of the counter b_i and the objects c_j, making essential use of the actions δ, τ. In each step, we increase the subscript of the counter by one, by means of rules $b_i \to b_{i+1}\delta$, and, in parallel, we decrease the subscript of all objects c_j by one. If we do not also have at least a copy of c_1, then the membrane is dissolved, and the object d will be released in membrane t_n or f_n containing membrane 2; in membranes t_n, f_n, the object d becomes d' and makes the membrane impermeable, hence no object can leave it. If we have an object c_1, then the membrane with label 2 remains of thickness 1, and the process continues. If we have at least one copy of each $c_1, c_2, \ldots, c_m$, then we reach the stage where we produce the object b_m (and all objects c_j were replaced by d). The object b_m can leave the membrane (note that it has normal thickness), can go up in the system until the skin region, and then sends the message *yes* out of the system.

In membranes t_n, f_n we work two steps, the counting in membrane 2 takes m steps, leaving the membrane by b_m means one step, reaching the skin region takes a further n steps, and leaving the system takes one furher step. In total, $4n + m + 5$ steps. Thus, if at this step we have the object *yes* outside the system, then formula C is satisfiable, otherwise it is not satisfiable. (Note that the system stops earlier if C is not satisfiable.)

Now, it is easy to see that this system is confluent, sound, linearly efficient, and also uniform: for each type of rules we have a number of rules which is polynomially bouded in n and m, hence we can construct these rules by a Turing machine in the same way as suggested in the proof of Theorem 7.2.3. We conclude that $\mathtt{SAT} \subseteq \mathbf{LMC}_C$. □

7.3.2 Solving HPP

There is an easy solution also for HPP (but not very satisfactory, from the point of view of the size of the system obtained), using membrane creation but not other "programming tools", and one more elaborate solution, by

means of a polynomially large system, also using addressing in_j and actions δ, τ.

We first give the easy solution.

Let $\gamma = (N, E)$ be a directed graph with n nodes, and construct the system Π with the alphabet of objects

$$\begin{aligned} O = \{a, yes\} &\cup \{a_i^{(t)}, \bar{a}_i^{(t)} \mid 1 \le i \le n, 1 \le t \le n\} \\ &\cup \{\langle D \rangle \mid D \subseteq \{1, 2, \ldots, n\}\}, \end{aligned}$$

one initial membrane $[_0\ ]_0$ containing the object a, and the following sets of rules – they also indicate the list of possible membranes:

$$\begin{aligned} R_0 = \{&a \to \bar{a}_1^{(1)} \bar{a}_2^{(1)} \ldots \bar{a}_n^{(1)}, \\ &\langle \{1, 2, \ldots, n\} \rangle \to (yes, out)\} \\ \cup \{&\bar{a}_i^{(1)} \to [_i a_i^{(1)}]_i \mid 1 \le i \le n\}, \\ R_i = \{&a_i^{(t)} \to \bar{a}_{j_1}^{(t+1)} \ldots \bar{a}_{j_s}^{(t+1)} \mid 1 \le t \le n-1, (i, j_k) \in E \\ &\text{for all } j = 1, 2, \ldots, s\} \\ \cup \{&\bar{a}_j^{(t)} \to [_j a_j^{(t)}]_j \mid 1 \le t \le n\} \\ \cup \{&a_i^{(n)} \to (\langle \{i\} \rangle, out)\} \\ \cup \{&\langle D \rangle \to (\langle D \cup \{i\} \rangle, out) \mid D \subseteq \{1, 2, \ldots, n\}\}, \text{ for } i = 1, 2, \ldots, n. \end{aligned}$$

In $2n$ steps we generate a membrane structure with the skin membrane labelled with 0, with n internal layers of membranes, labelled with the subscripts of nodes of γ, and such that the membranes placed directly inside a membrane with label i are labelled with those numbers j such that (i, j) is an edge in the graph. In this way, all paths in γ of length at most n are encoded by the paths in the tree which describes the membrane structure. During the creation of membranes we also count to n by means of the superscripts of the objects a_j, so that we know when we have obtained all paths of length n. In that moment we switch to checking whether or not at least one of these paths contains all nodes $1, 2, \ldots, n$. This is done by going from the elementary membranes towards the skin membrane, accumulating the labels of membranes by means of the objects of the form $\langle D \rangle$. This takes a further n steps. If we reach the skin membrane with the object $\langle \{1, 2, \ldots, n\} \rangle$, then we can send out the object *yes*. In $3n + 1$ steps we know whether or not γ contains a Hamiltonian path, and only in such a case do we produce the object *yes*, otherwise the computation stops earlier.

The discussion we had after presenting the first solution for SAT, at the beginning of the previous subsection, can be repeated here. Similarly, if we are allowed to use more powerful features, such as in_j, δ, and τ, then a P system Π can be constructed working in linear time, and also having a polynomial size. The idea is the same as that used in the proof of Theorem 7.3.2. That is, when creating a new membrane, going deep in the membrane structure, we

also introduce the node associated with that membrane (as we have done with the clauses satisfied by a truth value in the previous proof). Then, all these nodes must be passed to all membranes created inside the current membrane. To this end, we can use rules of the form $c_i \to (c_i, in_{j_1})(c_i, in_{j_2}) \dots (c_i, in_{j_s})$, for all descendants $j_1, j_2, \dots, j_s$ of node i. In this way, we accumulate in each membrane all labels of all membranes placed above it, that is all nodes visited by the path corresponding to the membranes placed above a given membrane. Then, when reaching the objects $a_i^{(n)}$, we have the nodes of all candidate paths of length n present in the elementary membranes, namely those from level n. We have to check whether or not at least one membrane of this type contains all nodes, and this can be done in exactly the same way as in the proof of Theorem 7.3.2 where we have checked whether or not any membrane with label 2 contains all clauses. Therefore, we can solve HPP in linear time by means of a system Π which fulfils all conditions for having a direct proof of the following theorem.

Theorem 7.3.3. HPP $\in$ **LMC**$_C$.

The details of this construction are left to the reader.

7.3.3 The Case of String–Objects

The membrane creation possibility can be introduced in a natural way also in P systems working with string–objects: consider rules of the form $a \to [_i v]_i$; when rewriting a string $w_1 a w_2$ by such a rule we get $[_i w_1 v w_2]_i$, that is a new membrane with label i containing the string $w_1 v w_2$.

The proof of Lemma 7.3.1 cannot be directly extended to this case, because we cannot enforce the creation of several membranes at the same level (unless we use more powerful rules than of the form $a \to [_i v]_i$, for instance, for creating two or more membranes at the same time, or for replicating strings), so we prove a universality result for these systems in a direct manner.

We denote by $[E]LSP_{n_1,n_2,n_3}(rw, in, \delta, mcre)$ the family of languages generated by [extended] rewriting P systems with membrane creation, using the communication commands *here, out, in*, the membrane dissolving action, and of profiles componentwise smaller than (n_1, n_2, n_3).

Theorem 7.3.4. $ELSP_{1,3,4}(rw, in, \delta, mcre) = RE$.

Proof. Let $G = (N, T, S, M, F)$ be a matrix grammar with appearance checking in the strong binary normal form, and construct the system $\Pi = (V, T, [_1\ ]_1, M_1, R_1, R_2, R_3, R_4)$, where

$$\begin{aligned} V &= N_1 \cup N_2 \cup T \cup \{X_{i,j} \mid X \in N_1, 1 \le i \le k, 0 \le j \le k\} \\ &\quad \cup \{A_{i,j} \mid A \in N_2, 1 \le i, j \le k\} \cup \{C, f, \#\}, \\ M_1 &= \{X_{init} A_{init},\ C\}, \end{aligned}$$

and with the following sets of rules:

R_1: $X \to [_2 X_{i,i}]_2$, for $m_i : (X \to \alpha, A \to x) \in M, 1 \le i \le k$,
$C \to (C, in)$,
$A_{i,j} \to \#$, for $1 \le i, j \le k$,
$A_{i,j} \to (A_{i,j-1}, in)$, for $1 \le i \le k, 2 \le j \le k$,
$A_{i,1} \to [_4 x]_4$, for $m_i : (X \to \alpha, A \to x) \in M, 1 \le i \le k$,
$f \to (\lambda, out)$,
$X \to [_{2+j} Y]_{2+j}$, for $m_i : (X \to Y, B^{(j)} \to \#) \in M, i \in lab_j, j = 1, 2$;
R_2: $A \to [_3 A_{i,i}]_3$, for $1 \le i \le k$,
$C \to C\delta$;
R_3: $X_{i,j} \to (X_{i,j-1}, out)$, for $1 \le i \le k, 2 \le j \le k$,
$X_{i,1} \to X_{i,0}\delta$, for $1 \le i \le k$,
$B^{(1)} \to \#$,
$C \to C$,
$C \to C\delta$;
R_4: $B^{(2)} \to \#$,
$C \to C\delta$,
$X_{i,0} \to \alpha$, for $m_i : (X \to \alpha, A \to x) \in M, 1 \le i \le k$,
$X_{i,j} \to \#$, for $1 \le i, j \le k$.

This system works as follows. Initially, we have two strings, $X_{init}A_{init}$ and C, in the unique initial membrane, with label 1. If we start by using a rule of the form $X \to [_2 X_{i,i}]_2$, then we will simulate a matrix $m_i : (X \to \alpha, A \to x) \in M$, $1 \le i \le k$, in the following way. The membrane with label 2 is created. At the next step, the rule $C \to (C, in)$ sends the symbol C into this membrane, and simultaneously in membrane 2 we create a membrane with label 3, containing a string of the form $X_{i,i}w_1A_{j,j}w_2$. At the next step, the rule $C \to C\delta$ dissolves membrane 2 and the symbol C returns to the external membrane.

In membrane 3 we decrease the second component of the subscript of X and the string is sent to membrane 1; simultaneously, C is sent to the inner membrane. In membrane 3 we can use the rule $C \to C$ an arbitrary number of times, or the rule $C \to C\delta$, which dissolves this membrane. If we dissolve membrane 3 while the string from membrane 1 contains a symbol $A_{j,k}$ with $k \ge 2$, then the only rule which can be applied here is $A_{j,k} \to \#$, and the trap symbol $\#$ is introduced; $\#$ can never be removed, hence no terminal string is obtained. Therefore, we must not use the rule $C \to C\delta$, or use it at the right time, as specified below.

Assume that membrane 3 exists (we use the rule $C \to C$ inside it), hence in membrane 1 we can use the rule $A_{j,k} \to (A_{j,k-1}, in)$. The string is sent to membrane 3, where again the second component of the subscript of X is decreased by one, and the string is sent out. This process is iterated, thus alternately decreasing the second components of the subscripts of X and A.

We distinguish three cases:

Case 1: If $i < j$, hence the rule $X_{i,1} \to X_{i,0}\delta$ is used in membrane 3 while the string contains a symbol $A_{j,k}$ with $k \geq 2$, then the rule $A_{j,k} \to \#$ must be used in membrane 1, and no terminal string is obtained.

Case 2: If $i > j$, hence we use the rule $A_{j,1} \to [_4 x]_4$ in membrane 1, having a string of the form $X_{i,k}w_1xw_2$ with $k \geq 1$ in membrane 4, then the only rule from this membrane which can be applied to this string is $X_{j,k} \to \#$, and again no terminal string will be obtained.

Case 3: If $i = j$, then when dissolving membrane 3, in membrane 1 we have a string of the form $X_{i,0}w_1A_{i,1}w_2$. We use the rule $A_{i,1} \to [_4 x]_4$, and the string $X_{i,0}w_1xw_2$ is introduced in membrane 4. At the next step, the string–symbol C also arrives in membrane 4, and we use the rule $X_{i,0} \to \alpha$. At the next step we dissolve membrane 4 by the rule $C \to C\delta$, hence we return to membrane 1 the strings αw_1xw_2 and C, the first one being the string obtained by a correct simulation of the matrix $m_i : (X \to \alpha, A \to x) \in M, 1 \leq i \leq k$.

If $\alpha \in N_1$, then the processs can be continued; if $\alpha = f$, then the rule $f \to (\lambda, out)$ can be used and the string is sent out of the system. If the string is terminal, then it is accepted in $L(\Pi)$, otherwise it is "lost". (The auxiliary string C remains in the system and cannot be rewritten, because no inner membrane exists.)

Assume now that we start in the unique membrane 1 by using a rule $X \to [_{2+j} Y]_{2+j}$, for some $j = 1, 2$, and $m_i : (X \to Y, B^{(j)} \to \#), i \in lab_j$. This produces a new membrane containing a string Yw. At the next step, the string C is sent to the newly generated membrane. In membrane $2 + j$ we check whether $B^{(j)}$ is present in the string (in the positive case the trap symbol $\#$ is introduced). If no occurrence of $B^{(j)}$ appears in the string Yw, then it cannot be rewritten, and waits unchanged. In membrane 3 we can use the rule $C \to C$ for a while, but eventually the rule $C \to C\delta$ must be used, otherwise we get no output. In membrane 4 we have to use the rule $C \to C\delta$ immediately. In both cases, the membranes are dissolved, hence the string Yw is returned to membrane 1. In this way, the correct use of the matrix m_i was simulated, with its second rule used in the appearance checking mode.

The process can continue. It is clear that in this way all derivations in G can be simulated in Π, and that, if a terminal string is sent out of the system Π, then it can also be generated by the grammar G. That is, $L(G) = L(\Pi)$. Because we start from a unique initial membrane, at each step (of a correct computation) we have at most three membranes in the system, out of four possible membranes, and the profile of the system Π is $(1, 3, 4)$, hence the proof is complete. □

It is a *research topic* to consider P systems with string–objects and membrane creation for solving **NP**-complete problems (Problem Q26).

7.4 Using String Replication

In Subsection 5.2.4 we considered rewriting membrane systems using rules of the form $a \to u_1||u_2||\dots||u_n$, which were applied to a string w_1aw_2 in such a way that n strings are simultaneosly obtained, namely $w_1u_1w_2$, $w_1u_2w_2, \dots,$ $w_1u_nw_2$. By making sure that the rules can be repeatedly applied, in this way we can generate exponentially many strings in linear time, hence again we have good premisses for obtaining polynomial solutions to **NP**-complete problems. We will confirm this expectation by again considering the standard problems, SAT and HPP, which can be solved in a rather easy way in this framework.

Let us denote by R the class of P systems with string–objects using replicated rewriting rules with the replication factor equal to 2 (and target indications *here, out, in*).

Theorem 7.4.1. SAT $\in$ **LMC**$_R$.

Proof. Starting from a propositional formula C with m clauses $C_j, 1 \le j \le m$, involving n variables $x_i, 1 \le i \le n$, we construct the system Π, of degree $m+2$, with replicated rewriting, as follows:

$$\begin{aligned}
\Pi &= (V, \mu, \{dt_1', df_1'\}, \emptyset, \dots, \emptyset, R_0, R_1, \dots, R_m, R_{m+1}),\\
V &= \{t_i, t_i', f_i, f_i' \mid 1 \le i \le n\} \cup \{d, yes\},\\
\mu &= [_{m+1}[_m \cdots [_2[_1[_0\ ]_0]_1]_2 \cdots]_m]_{m+1},\\
R_0 &= \{t_i' \to (t_it_{i+1}', here)||(t_if_{i+1}', here),\\
&\qquad f_i' \to (f_it_{i+1}', here)||(f_if_{i+1}', here) \mid 1 \le i \le n-1\}\\
&\quad \cup \{t_n' \to (t_n, out),\\
&\qquad f_n' \to (f_n, out)\},\\
R_j &= \{t_i \to (t_i, out) \mid \text{if } x_i \text{ appears in clause } C_j, 1 \le i \le n\}\\
&\quad \cup \{f_i \to (f_i, out) \mid \text{if } \neg x_i \text{ appears in clause } C_j, 1 \le i \le n\},\\
&\qquad \text{for all } j = 1, 2, \dots, m,\\
R_{m+1} &= \{d \to (yes, out)\}.
\end{aligned}$$

In the central membrane 0 we generate all 2^n truth–assignments for the variables $x_1, x_2, \dots, x_n$, in the form of strings $ag_1g_2\dots g_n$, with $g_i \in \{t_i, f_i\}$, $1 \le i \le n$. This takes n steps. Only when this operation is complete, hence t_n' or f_n' is present at the end of all strings, do we send all these string–objects to membrane 1, and we start checking the satisfiability of clauses. A string $ag_1g_2\dots g_n$ can exit a membrane with label j if and only if the truth–assignment encoded in it satisfies the clause C_j associated with the membrane. Thus, after m steps we have at least one string in the skin membrane if and only if there is a truth–assignment which satisfies all clauses, hence satisfies formula C. If this is the case, then in one further step we

send out all strings of the form *yes* $g_1 g_2 \ldots g_n$, where $g_1 g_2 \ldots g_n$ is a truth-assignment which satisfies formula C. Consequently, in $n+m+1$ steps we know whether or not the formula C is satisfiable. As it is clear that the system Π has the required properties, we conclude that SAT belongs to the class $\mathbf{LMC}_R$. □

The case of HPP is a little bit more complicated. The problem can be easily solved in linear time if we are allowed to use replication in more than two strings.

Indeed, let $\gamma = (N, E)$ be a directed graph with n nodes, and construct the system Π_0, of degree $n+2$, as follows:

$$\begin{aligned}
\Pi_0 &= (V, \mu, M_0, \emptyset, \ldots, \emptyset, R_0, R_1, \ldots, R_n, R_{n+1}),\\
V &= \{a_i \mid 1 \le i \le n\} \cup \{d, yes\}\\
&\quad \cup \{\langle a_i, r\rangle \mid 1 \le i \le n, 0 \le r \le n\},\\
\mu &= [_{n+1}[_n \cdots [_2[_1[_0\ ]_0]_1]_2 \cdots]_n]_{n+1},\\
M_0 &= \{d\langle a_i, 0\rangle \mid 1 \le i \le n\},\\
R_0 &= \{\langle a_i, r\rangle \to (a_i\langle a_{j_1}, r+1\rangle, here) || \ldots || (a_i\langle a_{j_k}, r+1\rangle, here) \mid\\
&\qquad (i, j_s) \in E \text{ for all } s = 1, 2, \ldots, k, 1 \le i \le n\}\\
&\quad \cup \{\langle a_i, n\rangle \to (a_i, out) \mid 1 \le i \le n\},\\
R_i &= \{a_i \to (a_i, out)\}, \text{ for all } 1 \le i \le n,\\
R_{n+1} &= \{d \to (yes, out)\}.
\end{aligned}$$

In n steps, in membrane 0 we generate all strings (of length at most $n+1$) of the form $da_{i_1}a_{i_2}\ldots a_{i_n}$ such that $i_1 i_2 \ldots i_n$ is a path in the graph γ. When we know that the strings have reached the length n (if there is no such string, then the computation stops earlier), then we send all these strings out of the inner membrane, namely that with label 0. The strings can then pass step by step from membrane to membrane towards the skin region only if they contain all symbols $a_1, \ldots, a_n$ (we exit membrane i only by rewriting the string by means of the rule $a_i \to (a_i, out)$). Thus, if any string reaches the skin membrane, this means that it encodes a Hamiltonian path in γ, and at the next step we can send out strings of the form *yes* $a_{i_1}a_{i_2}\ldots a_{i_n}$ (that is, together with the message *yes*, we also have the Hamiltonian paths). In $2n+2$ steps, the question whether or not γ contains any Hamiltonian path is answered.

It is clear that if the starting graph γ has the outdegree 2 (that is, from each node at most two nodes can be reached by edges in E), then the previous system has the replication factor equal to 2, hence HPP for such graphs can be solved in this way by systems from the class denoted above with R.

This observation can be used in order to solve HPP in polynomial time by means of P systems with 2-replication of strings (obtained in a direct manner). We give here only some hints about the proof of this assertion.

Consider a graph $\gamma = (N, E)$ with an arbitrary outdegree, s. Of course, $s \leq n-1$ (an edge of the form (i,i) can be removed, it will not appear on a Hamiltonian path). Take a node i such that $(i, j_1), \ldots, (i, j_k) \in E$, for some $3 \leq k \leq s$. We consider the new nodes $\langle j_2 j_3 \ldots j_k \rangle, \ldots, \langle j_{k-2} j_{k-1} j_k \rangle, \langle j_{k-1} j_k \rangle$ (there are $k-2$ nodes of this form), and we add these nodes to the graph γ (we call them *bifurcation nodes*, in order to distinguish them from the *initial nodes*, those from N). Then, we replace all edges $(i, j_t), 2 \leq t \leq k$, with the following edges:

$$
\begin{aligned}
&(i, \langle j_2 j_3 \ldots j_k \rangle),\\
&(\langle j_2 j_3 \ldots j_k \rangle, j_2),\ (\langle j_2 j_3 \ldots j_k \rangle, \langle j_3 \ldots j_k \rangle),\\
&\ldots\\
&(\langle j_{k-2} j_{k-1} j_k \rangle, j_{k-2}),\ (\langle j_{k-2} j_{k-1} j_k \rangle, \langle j_{k-1} j_k \rangle),\\
&(\langle j_{k-1} j_k \rangle, j_{k-1}),\ (\langle j_{k-1} j_k \rangle, j_k).
\end{aligned}
$$

We proceed in this way with all nodes i with the outdegree greater than or equal to 3. At the end of this procedure we get a graph $\gamma' = (N', E')$ with the number of nodes of the order of n^2. It is clear that the construction of γ' starting from γ takes a polynomial time (consider each of the n nodes of γ; for each of them scan E – it contains at most n^2 edges – and count how many descendants the node has; if the outdegree is greater than 3, then add the at most n bifurcation nodes, with at most n^2 new edges, in total, a cubic time). Assume that γ' contains m nodes. We modify the system Π_0 in such a way that we first generate all paths of length at most m in the graph γ', then we check whether or not at least one of these paths corresponds to a Hamiltonian path in γ. This can be easily done, just by ignoring the bifurcation nodes when checking whether or not all nodes from N are present on a path of length n. Consequently, we solve HPP for the graph γ in polynomial time, by means of systems of class R.

7.5 Using Pre-computed Resources

In the previous sections, we have always started from a given P system, hence a given initial configuration, and we have created an exponential workspace in linear time by means of various operations (with biological motivation). In this section we propose a different strategy: we start from an arbitrarily large initial membrane structure, without objects placed in its regions, and we trigger a computation by introducing both objects and rules related to a given problem in a specified membrane. The number of objects is finite, the number of rules is finite, but we consider the rule creation possibility, as briefly discussed in Subsection 5.2.1. In this way, both the number of objects and the number of available rules can increase, even exponentially.

As usual, the objects can be communicated from one region to another one. Because we need here to *activate* the regions of our membrane structure,

we also consider the possibility of communication for the rules. Thus, the general form of the rules, for the case of symbol–objects, will be

$$r : u \to v/z,$$

where u is a multiset of objects (represented by a string), v is a multiset of objects with attached target commands, hence pairs (a, tar), with $tar \in \{here, out, in\}$, and z is a string of pairs (p, tar), where p is the label of a rule from a given set of labels, and $tar \in \{here, out, in\}$. When the rule r is used, a copy of it is consumed, and the rules specified by z become available in the regions indicated by the associated targets. This way of moving rules from one region to another one is very powerful, but we do not consider it here from the point of view of the generative capacity of P systems, but as a way to bring evolution rules to the regions of a membrane structure which can start a computation without any rule in most of its membranes.

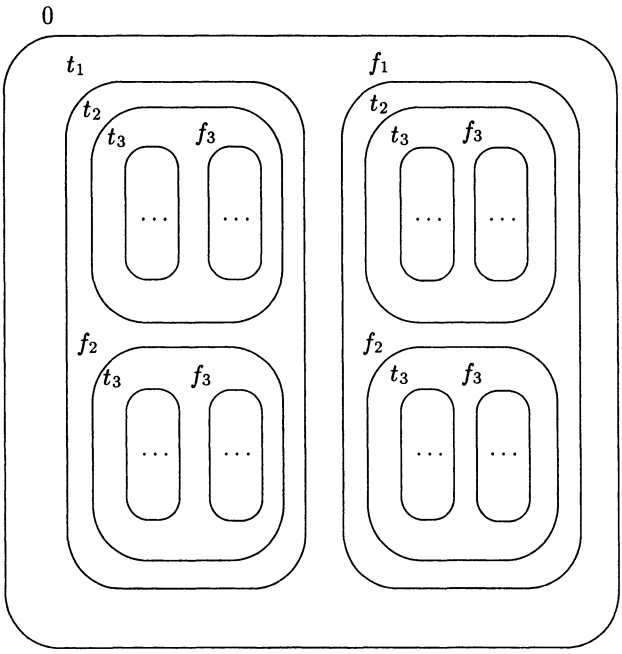


Fig. 7.4. Pre-computed membrane structure for solving SAT

In what follows, we will show the usefulness of this technique of activating regions of a given membrane structure in approaching computationally hard problems. We assume that the initial membrane structure is given "for free", as the result of a pre-computation whose duration does not matter

– although this membrane structure is not completely independent from the problem we want to solve. In the example which will follow, dealing with SAT, the membrane structure is assumed to be large enough for the instance we consider, but it can be used for any instance with a smaller number of variables. That is, assuming that the membrane structure is "arbitrarily large", we can handle different SAT problems with an "arbitrarily large" number of variables by reusing the same membrane structure. What we need to change from one problem to another problem, and from one instance to another one of the same problem, are the objects and the rules that we introduce in the system in order to initialize the computation. Thus, this kind of "solving" a problem does not lies in the framework established in Section 7.1.

Actually, what follows is mainly the formulation of a *research topic* (Problem Q27), somewhat specific to the membrane computing area, where we can assume that arbitrarily many cells are given, in a large tissue, just waiting to be used for a specific purpose, for instance a computation.

Let us consider an instance C of the SAT problem, with m clauses $C_1, \ldots, C_m$, using n variables $x_1, \ldots, x_n$.

We assume a membrane structure such as that suggested in Figure 7.4, hence described by a parenthetic formula of the following form

$$[_0[_{t_1}[_{t_2}[_{t_3}\cdots]_{t_3}[_{f_3}\cdots]_{f_3}]_{t_2}[_{f_2}[_{t_3}\cdots]_{t_3}[_{f_3}\cdots]_{f_3}]_{f_2}]_{t_1}$$
$$[_0[_{f_1}[_{t_2}[_{t_3}\cdots]_{t_3}[_{f_3}\cdots]_{f_3}]_{t_2}[_{f_2}[_{t_3}\cdots]_{t_3}[_{f_3}\cdots]_{f_3}]_{f_2}]_{f_1},$$

is given in advance. Both in the figure and in this formula we have considered three levels inside the skin membrane, but the structure is recurrent and can be as deep as we need. Specifically, for our instance of SAT, we need n internal levels.

Note that the "sister membranes", those placed in the same upper membrane, directly under it, are labelled with t_i and f_i, the truth values associated with the variable x_i. In this way, the membrane structure encodes all truth–assignments for the n variables. This is exactly what we have done in Section 7.2 in the first phase of solving SAT, when a similar membrane structure was created by means of repeated membrane division. Thus, we can assume that the membrane structure is not "too expensive", it can be created in a different computation, also done by biologically inspired tools.

As in Section 7.2 too, we assume that the membranes have associated electrical charges. Specifically, we assign the positive polarization to all membranes with labels of the form t_i, and the negative polarization to all membranes with labels of the form f_i. In the rules which will immediately follow, we also use the electrical charges for communicating objects and rules from a membrane to the immediately lower membranes. Because we always have exactly two membranes inside any membrane (excepting the elementary membranes, which can be supposed far away from the skin membrane, hence will not be activated in our computation), the communication is done in a deterministic manner, each object and each rule having only one membrane

of the opposite polarization which is accessible. (As usual, the neutral polarization is not indicated and it implies that the neutral objects and rules remain in the region where they are produced.)

What is not like Section 7.2 here is the fact that our membrane structure is empty: no object and no rule are present in its regions.

However, assume that we have available rules of the following forms (none of them is placed in the system yet):

$$\begin{aligned} r_i &: \langle M\rangle_i \to \langle M \cup \{j \mid x_{i+1} \text{ appears in } C_j\}\rangle_{i+1}^- \\ &\qquad \langle M \cup \{j \mid \neg x_{i+1} \text{ appears in } C_j\}\rangle_{i+1}^+ / \ r_{i+1}^- \ r_{i+1}^+, \\ &\quad \text{for } 0 \le i \le n-1, M \subseteq \{1,2,\ldots,m\}, \\ r_n &: \langle\{1,2,\ldots,m\}\rangle_n \to (yes, out)/(r_{n+1}, out), \\ r_{n+1} &: yes \to (yes, out)/(r_{n+1}, out). \end{aligned}$$

In these rules, each $\langle M\rangle_i$ is a symbol–object.

Assume now that, with the membrane structure empty, we throw inside the skin region the object $\langle\emptyset\rangle_0$, as well as a copy of the rule r_0. This initiates the computation, and we start to explore the membranes from the first n levels, one level at a time: each existing rule produces two new objects and two new rules; one object and one rule are negatively charged, hence they will enter the immediately lower membrane with positive polarization (that with the label of the form t_i); the other object and the other rule are positively charged, hence they will enter the lower membrane with the label f_i. The newly produced objects collect step by step the clauses which were already satisfied by the upper truth values assigned to variables, as well as the clauses satisfied by the truth values associated with the membranes where the objects enter. In this way, in n steps we reach the membranes with labels t_n, f_n, knowing at that time the clauses satisfied on the way from the skin membrane to the membranes from level n. If in any of the membranes from level n we have an object of the form $\langle\{1,2,\ldots,m\}\rangle_n$, hence telling us that all clauses were satisfied, then that object introduces the object *yes* and the rule r_{n+1}, and sends them to the upper membrane. The rule r_{n+1} reproduces itself and the object *yes* in the upper membrane, which will recurrently bring the object *yes* and the rule r_{n+1} into the skin region and from here out of the system. We need $n+1$ steps for sending the object *yes* outside the system, hence in total the computation lasts $2n+1$ steps. The objects $\langle M\rangle_n$ from the membranes in level n which have $M \neq \{1,2,\ldots,n\}$ remain forever in those membranes; in particular, if all objects from membranes in level n are of this type, then the formula σ is not satisfied, and we get no output.

This is one of the fastest solutions to SAT from this chapter (and the time depends only on the number of variables, not also on the number of clauses), but it has a drawback: in order to solve an arbitrarily given instance of SAT, we need not only an arbitrarily large membrane structure, but also an arbitrarily large set of possible rules, both of them pre-computed. However, the rules are

rather uniform in shape – in some sense, we have only three rules: r_n and r_{n+1}, which are unique, and the "parametrized rule"

$$r(\sigma, M, i) : \langle M \rangle_i \to \langle M \cup \sigma(t_{i+1}) \rangle_{i+1}^{-} \ \langle M \cup \sigma(f_{i+1}) \rangle_{i+1}^{+} / \\ r^{-}(\sigma, M \cup \sigma(t_{i+1}), i+1) \ \ r^{+}(\sigma, M \cup \sigma(f_{i+1}), i+1),$$

where $\sigma(v_i) = \{j \mid \text{clause } C_j \text{ is satisfied by } v_i\}$, $v_i \in \{t_i, f_i\}$ (t_i is interpreted as $x_i = true$ and f_i is interpreted as $x_i = false$), and M is any subset of $\{1, 2, \ldots, m\}$. Because of M, we have exponentially many (with respect to m) "interpretations" of this "rule pattern".

Whether or not there is a way to use a number of rules which is polynomial in n and m remains as a research topic – likewise the whole idea of using a pre-computed arbitrarily large amount of resources, membranes and/or rules.

7.6 Bibliographical Notes

P systems with membrane division were introducd in [192], where both universality and a linear time solution to SAT are given – but without a bound on the number of membranes in the case of the universality (also, using rules of all types (a)–(f)), and using rules of type (f) in the case of SAT. The results were improved in [199]: rules of type (f) are no longer used – but still without a bound on the number of membranes in the universality case. Then, in [175] the universality is given without any membrane division rules – still without a bound on the number of membranes. Such a bound has been found in [63] – four membranes. The result from Theorem 7.2.1 – three membranes suffice – is from [135]. The construction from Theorem 7.2.2 is from [247] (see also [242]).

A solution to SAT by means of systems using membrane division can also be found in [171], with a continuation in [173], where the issue of the uniformity (hence of the polynomial size, too) of the P systems able to solve a problem is considered for the first time; in [173] it is shown that the constructions from [171] and [192] are uniform. The Hamiltonian path problem was addressed for the first time in [113], but using rules for d-division, with an arbitrary d. In the same paper (and in the same framework) the Vertex cover problem is also solved in linear time. A polynomial solution to HPP using only 2-division is given in [175]. The solution we recall here in Theorem 7.2.4 is from [247].

The membrane division is also used in [170], where it is shown that the inverse of one-way functions, of importance in cryptography, can be computed in this context in polynomial time, as well as in [172]. Another important topic related to cryptography is addressed in [120]: breaking the widely used cryptosystem DES (of course, in the idealized framework of membrane division).

The contents of Subsection 7.2.3 are from [199]. Theorem 7.2.5 is from [247] (see also [242]).

The membrane creation was first considered in [100] and [131]. Theorem 7.3.1 is from [100]. Universality results are given in [131] (for symbol–objects) and [158] (for string–objects, Theorem 7.3.4). The possibility of introducing catalysts in the newly created membranes is considered in [215], and then in [135]. In [131] a solution to HPP using P systems with membrane creation is also given.

The string replication was introduced in [39], where also SAT and HPP are considered, and, independently, in [117], where the same two problems, SAT and HPP, are considered,

The contents of Section 7.5 are new. A similar idea, of activating membranes of a pre-computed membrane structure, also appears in [46], in a completely different setup.

A survey of results about variants of P systems able to attack **NP**-complete problems can be found in [188]. Consistent chapters about this topic can be found in [108] and [242]. A comparison of several DNA computing and membrane computing algorithms for solving SAT can be found in [139]. The diploma thesis [52] considers the question of checking the truth value of a propositional formula by using a P system with membrane division in the framework of an automated theorem prover, related to the Pascal implementation of P systems from [137]. The membrane division is also used in other papers, either in order to speedup the computation (as in [13, 14]), to handle the membrane structure in a more elaborate way (in [15]), in order to separate the contents of regions according to a given predicate, which is useful for formulating DNA computing experiments in terms of membrane computing ([87], with a continuation in [145]).

The replication of strings was considered also for contextual P systems in [112], where SAT and HPP are solved in polynomial time by such systems, and for the so-called P systems with valuation in [147, 148].

8. Further Technical Results

In the previous chapters we have systematically considered only two types of problems and results related to them: *computability power* (also looking for normal forms, from various points of view, especially in what concerns the number of membranes) and *computability efficiency*. However, in the membrane computing literature there are several directions of research which are different from the above two, some of them mathematically motivated, others having computer science motivations, and others aiming to get closer to reality (in two directions: to the living cell, and to possible applications). In the present chapter we briefly present some results from the first two categories, that is with mathematical or computer science motivations, while in the next chapter we will consider attempts to bring membrane systems closer to biology or to other areas where the paradigms of membrane computing seem to be useful.

8.1 Decidability Results

We start with types of problems which are standard in formal language theory, in general, in computer science, but which we have not discussed up to now in the framework of membrane computing: decidability. There are several problems of this type which can be formulated for P systems of different types or for their sets of numbers or languages. Can we decide, for systems Π from a given class X, whether or not $N(\Pi)$ (or $L(\Pi)$ in the case of string–objects) is empty/finite/infinite/semilinear/in a given family NFL, for FL a family of languages from the Chomsky or Lindenmayer hierarchy? Are two systems equivalent? Then, we have questions at the level of configurations: given an arbitrary configuration of a system Π, does the computation starting from it ever halt? (This is a reformulation of the emptiness problem.) Is that configuration reachable from the initial configuration? More specific problems: is a specified membrane from a given configuration (taken as an initial configuration of computations) ever dissolved? Does it ever become empty/elementary?

And so on, and so forth – both classic questions from language theory and specific questions.

Of course, if a class X of systems is universal, then it is expected that all (non-trivial) questions about systems of type X and about their sets of numbers/languages are undecidable. Also, some results from language theory can be imported in the new framework: if the class X is not universal, but covers at least the power of context-free languages (in a constructive manner; that is, starting from a grammar we can effectively construct an equivalent P system from class X), then the decidability problems which have a negative answer for context-free languages have a negative answer also for the languages generated by systems of type X.

We do not consider such possibilities here systematically, but we do consider two problems which are specific to P systems.

The first problem asks whether or not a given membrane is ever dissolved. This is of a clear interest for membrane computing, the action δ being one of the basic features used in many types of systems. Moreover, recall that the reason to have a passing from m membranes to $m+1$ membranes in Lemma 3.7.2 was the fact that we do not know if a given elementary membrane is dissolved during a computation, hence we cannot use it as an output membrane. As expected, the problem is undecidable for systems which are powerful enough, and decidable for systems with a reduced power. We prove this only for one case, exactly that of systems as used in Lemma 3.7.2: with symbol–objects, using catalysts, priorities, and, of course, the membrane dissolving action.

Theorem 8.1.1. *It is undecidable whether or not a given membrane of a P system of type* (cat, tar, pri, δ) *will ever be dissolved.*

Proof. According to Theorem 3.4.3, each recursively enumerable set of numbers $Q \subseteq \mathbf{N}$ belongs to the family $NOP_2(cat, tar, pri)$. Take an arbitrary set $Q \in NRE$ and consider a P system Π such that $Q = N(\Pi)$. From the proof of Theorem 3.4.3, it is clear that if we send out the object a instead of sending it to membrane 2, then we generate the same number (now, in the external mode). In this way, we can ignore membrane 2, hence we may assume that Π has only one membrane. Let $\Pi = (O, K, \{a\}, [_{1'}\ ]_{1'}, w_{1'}, (R_{1'}, \rho_{1'}))$ be the system modified in this way (note that its membrane is labelled with $1'$). Starting from Π, we construct a new system, Π_t, where t is any integer $t \geq 0$, as follows.

The total alphabet of Π_t is

$$O' = O \cup \{b, b', c', c'', d, d', e, e', f\},$$

where c', c'' are new catalysts (thus the set of catalysts of Π_t is $K \cup \{c', c''\}$); the objects $b, b', c', c'', d, d', e, e', f$ are not in O.

The system Π_t is as indicated in Figure 8.1; that is, it has the following components:

$$\mu' = [_1[_2\ \mu\ ]_2[_3\ ]_3[_4\ ]_4]_1,$$

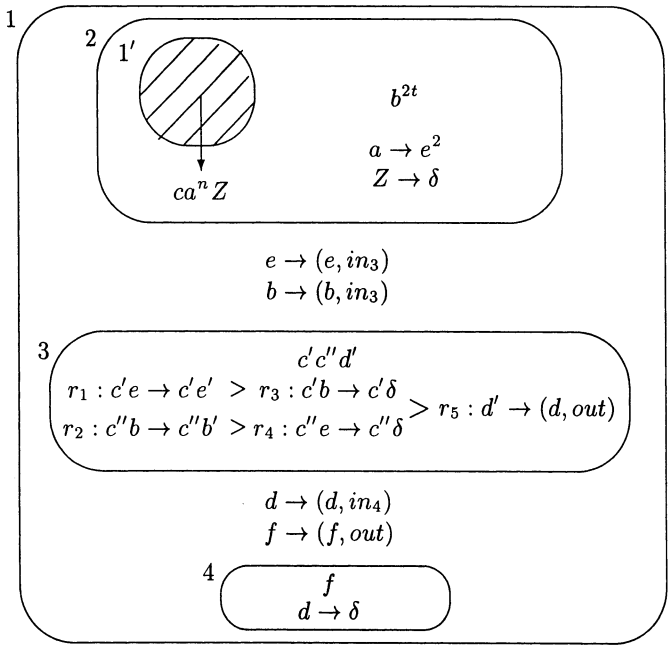


Fig. 8.1. The system from the proof of Theorem 8.1.1

$$
\begin{aligned}
w_1' &= \lambda, \\
w_2' &= b^{2t}, \\
w_3' &= c'c''d', \\
w_4' &= f, \\
R_1' &= \{e \to (e, in_3),\ b \to (b, in_3), \\
&\qquad f \to (f, out),\ d \to (d, in_4)\}, \\
\rho_1' &= \emptyset, \\
R_2' &= \{a \to e^2,\ Z \to \delta\}, \\
\rho_2' &= \emptyset, \\
R_3' &= \{r_1 : c'e \to c'e',\ r_2 : c''b \to c''b',\ r_3 : c'b \to c'\delta, \\
&\qquad r_4 : c''e \to c''\delta,\ r_5 : d' \to (d, out)\}, \\
\rho_3' &= \{r_1 > r_3,\ r_2 > r_4,\ r_3 > r_5,\ r_4 > r_5\}, \\
R_4' &= \{d \to \delta\}, \\
\rho_8' &= \emptyset.
\end{aligned}
$$

All the components of Π remain unchanged, with the exception of the fact that we introduce the rule $Z \to Z\delta$ in its set of rules (remember that Z

indicates the end of the computation, because the construction from the proof of Theorem 3.4.3 has started from a matrix grammar in the Z-binary normal form). The shaded membrane $1'$ in Figure 8.1 indicates that the contents of this membrane are not of interest for what follows (complete computations of Π in this membrane produce strings a^n, for $n \in Q$).

Claim: *Membrane 4 of system Π_t is dissolved if and only if $t \in Q$.*

No rule can be used in membrane 2 before receiving here objects from membrane $1'$. When any copy of a exits from membrane $1'$ it is replaced by e^2. When the computation in Π is finished, the object Z will dissolve membrane $1'$, hence both Z and c, the catalyst of Π, will be released in membrane 2. At the next step, Z dissolves this membrane, hence $2n$ copies of a, $2t$ copies of b, as well as c and Z are left free in the skin membrane of the system Π'.

The rules $e \to (e, in_3), b \to (b, in_3)$ just pass the $2n$ copies of e and the $2t$ copies of b to membrane 3. Note that if $n \neq t$, then $2n$ differs from $2t$ by at least 2. This is important in the reasoning below. In membrane 3 we have two catalysts, c' and c'', one copy for each. During each time unit, each catalyst decreases by one the number of copies of e and of b (by priming them). This is done by the rules $c'e \to ce'', c''b \to c''b'$ which have priority over all other rules in membrane 3. If $n = t$, then the objects e, b are finished at the same time. This makes the use of the rule $d' \to (d, out)$ possible (no rule with a higher priority is applicable). From membrane 1, the object d enters membrane 4 and dissolves it. Just to have some output, the object f released in this way is sent out of the system.

Assume now that $n \neq t$. The two possibilities, $n > t$ and $t > n$, are handled in the same way, so we discuss only the case when $n > t$. This implies that $2n - 2t \geq 2$. After using $2t$ times the rules r_1, r_2, we have no further copy of b in membrane 3, but we still have at least two copies of e. One of them will be used by the rule $c'e \to ce''$; because the rule $c'e \to c'e'$ cannot be used once again (the catalyst c is present in one copy only), and the same holds for the rule $c''b \to c''b'$ (no b is present), the rule $c''e \to c''\delta$ can (and *should*) be used. This means that membrane 3 is dissolved. Its content is released in membrane 1, its rules are lost. Therefore the object d' will remain unmodified, and so the rule $d \to (d, in_4)$ from membrane 1 cannot be applied. The computation stops here without dissolving membrane 4 (and without sending out any object, because f remains "locked" inside membrane 4).

In conclusion, membrane 4 is dissolved if and only if $t = n$, that is if and only if $t \in Q$. Because Q is an arbitrary recursively enumerable set, the membership with respect to Q is undecidable. □

Actually, we have proved here that the problem is undecidable for systems of the considered type *containing at least five membranes.* It would be interesting to see whether or not the result remains true for a smaller number of membranes.

From the previous proof, we can infer many other undecidability results. Note that the appearance of the object d, the use of the rule $d \to \delta$, and the fact whether or not f is sent out of membrane 1 depend on the fact whether or not $t \in Q$, which is undecidable.

Corollary 8.1.1. *It is undecidable whether or not any of the following events take place in a P system of type* (cat, tar, pri, δ)*:*

1. *a given object ever appears in the system;*
2. *a given object is sent out of the system;*
3. *a given evolution rule is ever used.*

In the search for a positive decidability result, consider now a class of systems with a reduced computing power.

Theorem 8.1.2. *It is decidable whether or not a given membrane in a P system of type* $(ncoo, tar, \neg pri, \delta)$ *is ever dissolved during a successful computation.*

Proof. Consider a system Π of type $(ncoo, tar, \neg pri, \delta)$ and a membrane i of it. Assume that the alphabet of Π is V. We modify the system as follows. Take a new symbol, d, and add it to V; denote this new alphabet by V'. If there is a rule of the form $a \to v\delta$ in membrane i, then we replace it with $a \to vd\delta$. Then we add the rule $d \to (d, out)$ to all membranes, including the skin one. Let us denote by Π' the system obtained in this way and consider it as a language generating device, in the external mode.

It is clear that $L(\Pi')$ contains all the strings in $L(\Pi)$ which were obtained at the end of computations where membrane i was not dissolved, as well as strings of the form $w_1 d w_2 \ldots d w_k$, for $k \geq 1$ and $w_1 w_2 \ldots w_k \in L(\Pi)$, which were obtained by computations where membrane i was dissolved. (The computation in Π' cannot stop before sending out the symbol d, because each region contains the rule $d \to (d, out)$. It is possible to use several rules of the form $a \to vd\delta$ at the step when membrane i is dissolved.)

From Theorem 3.4.2 we know that $NOP_*(ncoo, tar, \delta) \subseteq NET0L$. This inclusion is proved in a constructive way. Therefore, starting from our system Π', we can effectively construct an ET0L system G such that $\Psi_{V'}(L(\Pi')) = \Psi_{V'}(L(G))$. Consider the language $L = L(G) \cap V'^*\{d\}V'^*$. This is also an ET0L language (the family $ET0L$ is closed under intersection with regular languages). Clearly, L is non-empty if and only if the symbol d appears in at least one string of $L(G)$; that is, if and only if membrane i was ever dissolved during the computation of a string in $L(\Pi)$. The emptiness is decidable for ET0L languages, hence our problem is decidable, too. □

Let us now consider another problem which is specific to P systems: the reachability of a given configuration. For systems from most (if not all) classes able to characterize NRE the fact that this problem is undecidable easily follows from the proofs of the universality results from Chapters 3 and 4: we

always know which is the membrane structure of the halting configurations, we always have copies of a specified object a in the output membrane, and other possible objects in other membranes (sometimes we have no other object). If the objects different from a which appear in the halting configuration are known in advance (they are, for instance, catalysts, which never evolve), then we know the halting configuration completely – excepting the number of copies of a in the output membrane. Then, the undecidability follows: take an arbitrary integer n and ask whether or not the configuration having the known objects different from a in its regions, and n copies of a in the output region is reachable. This is equivalent with the membership of n to the set of generated numbers, and this problem is undecidable for systems which characterize NRE. Sometimes, also in the cases when we do not know exactly the objects different from a which can be present in the halting configuration, we can arrange the work of the system in such a way that after finishing the computation such objects are sent out of the system, so that the configuration remains "clean". Such an operation is possible when we know the moment of reaching a halting configuration (for instance, when we start the proof from a matrix grammar in the Z-binary normal form, or when we simulate in the last step a matrix which introduces a special symbol, such as f in several proofs from Chapters 3 and 4).

A very interesting case appears for systems with symport/antiport rules, where in a configuration we have also considered the objects which were sent out of the system and which are different from the objects present in the environment in arbitrarily many copies. We start with a finite number of (copies of) objects present initially in the system. In the halting configuration, they are inside or outside, but we can proceed as above and send them out, for instance, by symport rules of the form $(f\alpha, out)$, introduced in the system for all "undesirable" objects α (of course, we also need a rule to bring back the special object f, for instance (f, in), if it does no harm, or (fg, in), for a special object g). Because we do not care about objects which are present initially in the environment in arbitrarily many copies, the reachability problem is undecidable for all classes of systems with symport/antiport rules which are universal. This is the case for several combinations of parameters (number of membranes, weight of symport rules, weight of antiport rules) – see Theorems 4.2.1, 4.2.2, 4.2.3, 4.2.4.

The surprise comes if we slightly modify the definition of a configuration, and we also take into account the objects from the environment which were sent out of the system during the computation after being brought inside for a while. Remember that rules of the form (v, in), for multisets v whose elements appear in the environment in arbitrarily many copies, are not allowed (they transform the system into a "black hole", bringing inside infinitely many copies of objects, and this is not "legal"). Because of this assumption, the number of objects sent out of the system during a computation is finite (it

can be arbitrarily large, but it cannot be infinite). Thus, we can represent their multiset by a string, as usual for internal multisets.

If the membrane structure has m membranes, labelled by $1, 2, \ldots, m$, then we call *extended configuration* the $(m+1)$-tuple $(z_1, \ldots, z_m, z_{m+1})$, of strings over O, where z_i, $1 \leq i \leq m$, represents the multiset of objects present in membrane i, and z_{m+1} represents the multiset of all objects sent into the environment during a computation in Π which has led to $z_1, \ldots, z_m$ inside the system; that is, z_{m+1} also contains the objects from E which have entered the system and after that were sent out.

In this definition of an extended configuration (as well as in the proof of Lemma 8.1.1) we keep in z_{m+1} a copy of an object after being sent out, irrespective of whether or not it has entered the system again, but the proof can be slightly changed in order to have in z_{m+1} only the copies of the objects which are currently present in the environment, after being expelled from the system. Thus, the result from Theorem 8.1.3 is true also for this "more economical" definition of an extended configuration.

For an easier representation, we write an extended configuration as a string over $O \cup \{d\}$, where d is a new symbol (a delimiter), representing the comma, namely, by $z_1 d z_2 d \ldots d z_m d z_{m+1}$. Furthermore, we assume the alphabet O is ordered, $O = \{a_1, \ldots, a_p\}$, and we write the strings z_j in the "standard form", that is in the form $a_1^{i_1} \ldots a_p^{i_p}$, where $i_1, \ldots, i_p$ are the multiplicities of objects $a_1, \ldots, a_p$, respectively, in the multiset represented by z_j (of course, we can have $i_k = 0$, which means that the corresponding symbol a_k does not appear in z_j). We will denote by w_i the string that represents the objects present in membrane i in the initial configuration.

We denote by $econ(\Pi)$ the language of all strings of this form, describing (in the standard form) the extended configurations of Π which can be reached from the initial configuration $w_1 d w_2 d \ldots d w_m d$ (with λ describing the multiset of objects sent into the environment before starting any computation). Moreover, we denote by $ehalt(\Pi)$ the language of strings describing (in the standard form) halting extended configurations which can be reached from $w_1 d w_2 d \ldots d w_m d$.

The following two lemmas are the basis of the announced decidability of the reachability problem for extended configurations.

Lemma 8.1.1. *For every P system Π with symport/antiport (with free application of rules, that is without permitting or forbidding contexts), the language $econ(\Pi)$ belongs to the family MAT_{ac}.*

Proof. Let us consider a P system with symport/antiport $\Pi = (O, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m, i_o)$. As usual, we assume that the skin membrane has label 1 and that different membranes have different labels. For a membrane i from μ we denote by $up(i)$ the membrane immediately outside i; for $i = 1$ we put $up(1) = 0$. Assume that O contains g elements. Moreover, we assume that the rules of $R_1, \ldots, R_m$ are labelled in a one-to-one manner, by

$r_1, \ldots, r_s$, with r_1 being a rule from R_1 (if R_1 is empty, then the changes in the construction below are straightforward and are left to the reader).

We construct the matrix grammar $G = (N, T, S, M, F)$, with

$$\begin{aligned} N &= \{S, X, Y, Z, \#\} \cup \{X_i \mid 1 \leq i \leq s\} \\ &\quad \cup \{(a,i)', (a,i)'' \mid a \in O, 0 \leq i \leq m\}, \\ T &= \{(a,i) \mid a \in O, 0 \leq i \leq m\} \cup \{d\}, \end{aligned}$$

and the set of matrices constructed in the following way.

Consider the morphisms $h_i, 1 \leq i \leq m$, defined by $h_i(a) = (a,i)'$, for each $a \in O$. Then,

$$\begin{aligned} M = \{&(S \to X\ h_1(w_1)d\ h_2(w_2)d \ldots d\ h_m(w_m)d), \\ &(S \to Z\ h_1(w_1)d\ h_2(w_2)d \ldots d\ h_m(w_m)d)\} \\ \cup\ \{&(X \to X, (a_1,i)' \to (a_1, up(i))'', \ldots, (a_u,i)' \to (a_u, up(i))'', \\ &(b_1, up(i))' \to (b_1,i)'', \ldots, (b_v, up(i))' \to (b_v,i)'') \mid \\ &(a_1 \ldots a_u, out; b_1 \ldots b_v, in) \in R_i, 2 \leq i \leq m, u, v \geq 0, \\ &\text{where } a_j, b_j \in O \text{ for all } j\} \\ \cup\ \{&(X \to X(b_{j_1},1)'' \ldots (b_{j_q},1)'', (a_1,1)' \to (a_1,0)', \ldots, \\ &(a_u,1)' \to (a_u,0)', (b_{k_1},0)' \to (b_{k_1},1)''(b_{k_1},0), \ldots, \\ &(b_{k_r},0)' \to (b_{k_r},1)''(b_{k_r},0)) \mid \\ &(a_1 \ldots a_u, out; b_1 \ldots b_v, in) \in R_1, u, v \geq 0, \text{ where} \\ &a_j, b_j \in O \text{ for all } j, \{1,2,\ldots,v\} = \{j_1,\ldots,j_q\} \cup \{k_1,\ldots,k_r\}, \\ &\{j_1,\ldots,j_q\} \cap \{k_1,\ldots,k_r\} = \emptyset, \text{ and } b_{j_l} \in E, \text{ for } 1 \leq l \leq r\} \\ \cup\ \{&(X \to X_1, (a_1,1)' \to ck((a_1,1)'), \ldots, (a_u,1)' \to ck((a_u,1)'), \\ &(b_{k_1},0)' \to ck((b_{k_1},0)'), \ldots, (b_{k_r},0)' \to ck((b_{k_r},0)')) \mid \\ &r_1 = (a_1 \ldots a_u, out; b_1 \ldots b_v, in) \in R_1, u, v \geq 0, \text{ where } a_j, b_j \in O \\ &\text{for all } j, ck((\alpha,j)') \in \{(\alpha,j)', \#\}, \text{ for all } \alpha \text{ and } j \text{ as above,} \\ &\text{and there is at least one } j \text{ such that } ck((a_j,i)') = \#, \text{ or} \\ &ck((b_j, up(i))') = \#; \text{ moreover, } b_{k_l} \notin E, 1 \leq l \leq r, \text{ and, if} \\ &l \in \{1,2,\ldots,v\} - \{k_1,\ldots,k_r\}, \text{ then } b_l \in E\} \\ \cup\ \{&(X_{t-1} \to X_t, (a_1,i)' \to ck((a_1,i)'), \ldots, (a_u,i)' \to ck((a_u,i)'), \\ &(b_1, up(i))' \to ck((b_1, up(i))'), \ldots, (b_v, up(i))' \to ck((b_v, up(i))')) \mid \\ &r_t = (a_1 \ldots a_u, out; b_1 \ldots b_v, in) \in R_i, 2 \leq i \leq m, u, v \geq 0, \\ &\text{where } a_j, b_j \in O \text{ for all } j,\ ck((\alpha,j)') \in \{(\alpha,j)', \#\}, \\ &\text{for all } \alpha \text{ and } j \text{ as above, and there is at least one } j \\ &\text{such that } ck((a_j,i)') = \#, \text{ or } ck((b_j, up(i))') = \#, 2 \leq t \leq s-1\} \\ \cup\ \{&(X_{t-1} \to X_t, (a_1,1)' \to ck((a_1,1)'), \ldots, (a_u,1)' \to ck((a_u,1)'), \\ &(b_{k_1},0)' \to ck((b_{k_1},0)'), \ldots, (b_{k_r},0)' \to ck((b_{k_r},0)')) \mid \end{aligned}$$

$r_t = (a_1 \dots a_u, out; b_1 \dots b_v, in) \in R_1, u, v \geq 0,$ where $a_j, b_j \in O$
for all $j, ck((\alpha, j)') \in \{(\alpha, j)', \#\}$, for all α and j as above,
and there is at least one j such that $ck((a_j, i)') = \#$, or
$ck((b_j, up(i))') = \#$; moreover, $b_{k_l} \notin E, 1 \leq l \leq r$, and, if
$l \in \{1, 2, \dots, v\} - \{k_1, \dots, k_r\}$, then $b_l \in E, 2 \leq t \leq s-1\}$
$\cup \{(X_{s-1} \to Y, (a_1, i)' \to ck((a_1, i)'), \dots, (a_u, i)' \to ck((a_u, i)'),$
$(b_1, up(i))' \to ck((b_1, up(i))'), \dots, (b_v, up(i))' \to ck((b_v, up(i))') \mid$
$r_s = (a_1 \dots a_u, out; b_1 \dots b_v, in) \in R_i, 2 \leq i \leq m, u, v \geq 0,$
where $a_j, b_j \in O$ for all $j, ck((\alpha, j)') \in \{(\alpha, j)', \#\}$,
for all α and j as above, and there is at least one j
such that $ck((a_j, i)') = \#$, or $ck((b_j, up(i)')) = \#\}$
$\cup \{(X_{s-1} \to Y, (a_1, 1)' \to ck((a_1, 1)'), \dots, (a_u, 1)' \to ck((a_u, 1)'),$
$(b_{k_1}, 0)' \to ck((b_{k_1}, 0)'), \dots, (b_{k_r}, 0)' \to ck((b_{k_r}, 0)')) \mid$
$r_s = (a_1 \dots a_u, out; b_1 \dots b_v, in) \in R_1, u, v \geq 0,$ where $a_j, b_j \in O$
for all $j, ck((\alpha, j)') \in \{(\alpha, j)', \#\}$, for all α and j as above,
and there is at least one j such that $ck(a_j, i) = \#$, or
$ck((b_j, up(i))') = \#$; moreover, $b_{k_l} \notin E, 1 \leq l \leq r$, and, if
$l \in \{1, 2, \dots, v\} - \{k_1, \dots, k_r\}$, then $b_l \in E\}$
$\cup \{(Y \to Y, (a, i)'' \to (a, i)') \mid a \in V, 1 \leq i \leq m\}$
$\cup \{(Y \to X, (a_1, i)'' \to \#, \dots, (a_g, i)'' \to \#),$
$(Y \to Z, (a_1, i)'' \to \#, \dots, (a_g, i)'' \to \#) \mid 1 \leq i \leq m,$
and $O = \{a_1, \dots, a_g\}\}$
$\cup \{(Z \to Z, (a, i)' \to (a, i)) \mid a \in O, 0 \leq i \leq m\}$
$\cup \{(Z \to d)\}.$

The set F, of rules to be used in the appearance checking mode, consists of all rules of the form $\alpha \to \#$ from the matrices of M.

Let us briefly examine the work of the grammar G.

The symbols X, Y, Z, as well as $X_i, 1 \leq i \leq s$, control the work of the other rules, in such a way that a computation in Π is simulated by a derivation in G. Specifically, in the presence of X one simulates a non-deterministic maximally parallel use of rules from $R_1, \dots, R_m$, while the symbols $X_1, \dots, X_s$ are used for checking that no further application of the rules $r_1, \dots, r_s$, respectively, is possible at that step. One starts from a description of the initial configuration where all symbols a present in a region i are replaced by the new symbol $(a, i)'$. When simulating the rules of Π, one passes from primed symbols $(a, i)'$ to double primed symbols $(a, j)''$, where j is the membrane outside membrane i in the case when the symbol a is sent *out*, and, conversely, i is the membrane outside j, in the case when a is sent *in*. The symbols sent out of the system

are written in the form $(a,0)'$. All double primed symbols present inside the system are replaced by single primed symbols in the presence of Y. When this operation is complete, either we return to X, hence the process can be iterated, or we replace Y by Z. In the presence of Z we replace each primed or double primed symbol by the non-primed symbol, which is considered terminal with respect to G, and the derivation stops. All these operations are performed by using the rules of the form $\alpha \to \#$ in the appearance checking manner, with $\#$ being a trap symbol, which is never removed.

The first sentential form generated by G from its axiom S is

$$X\ h_1(w_1)d\ h_2(w_2)d \ldots d\ h_m(w_m)d.$$

During the work of G, the symbols corresponding to various membranes are mixed, while the symbols d are never rewritten. However, the second component of symbols of the form $(a,i)'$ present in any current sentential form identifies the membrane i where the symbol a is placed. After using any rule, the symbols from the system change their second component, according to the *in/out* indication of the rule, and it is double primed. This latter fact prevents the use of the same symbol by another rule at the same step.

In the case of membranes $2 \leq i \leq m$, the simulation of rules from R_i is rather easy. Some precautions should be taken in the case of rules from R_1 with the symbols which are introduced from the environment into the system. If these symbols appear in the environment in arbitrarily many copies, then again there is no difficulty (such symbols are introduced by the rules of the form $X \to X(b_{j_1},1)'' \ldots (b_{j_q},1)''$). If the symbols to be introduced in the system are not from E, but were previously sent out of the system, then they appear in a bounded number of copies in the environment, hence we have to make sure that enough copies of them are present; moreover, we still preserve copies of them in the environment, but without any prime (the corresponding rules are those of the form $(b_{k_1},0)' \to (b_{k_1},1)''(b_{k_1},0), \ldots,$ $(b_{k_r},0)' \to (b_{k_r},1)''(b_{k_r},0)$). The non-primed symbols which remain in the environment cannot be used by a further rule, and they remain unchanged until the end of the derivation.

At any moment, we can use the matrix which starts with the rule $X \to X_1$. This matrix checks whether or not the rule r_1 can be applied once again to the symbols which are single primed. If this is the case, then the trap symbol $\#$ is introduced and the sentential form will never lead to a terminal string. Otherwise, we continue by introducing the symbol X_2, a step when we check whether or not the rule r_2 can be applied. We continue with $X_3, \ldots, X_{s-1}$; in each case, the applicability of the corresponding rules $r_2, \ldots, r_s$ is checked, and always the trap symbol is introduced if the rule can be applied. If no rule can be applied, that is the computation was maximally parallel (hence correct with respect to Π), then the symbol Y is introduced.

In the presence of Y we simply replace each double primed symbol with its single primed variant. When this process is complete (this is checked by

the rules of the form $(a_j, i)'' \to \#$ from the matrix which starts with the rule $Y \to X$), then we return to the symbol X and the process can be iterated, we can simulate another transition with respect to Π.

At any moment when Y can be replaced by X, we can also introduce the control symbol Z. In the presence of this symbol we replace each primed symbol with its non-primed version. When all symbols are thus replaced by terminal symbols, we can also replace the symbol Z by d. Therefore, all configurations which can be reached after at least one transition in Π correspond to a sentential form generated by G.

In order to produce also the initial configuration of Π, we use the matrix $(S \to Z\ h_1(w_1)d\ h_2(w_2)d \dots d\ h_m(w_m)d)$, then Z assists the removal of primes, and after that it is replaced by d.

Therefore, the strings from $L(G)$ are of the form dw, where w is a permutation of a string over $V \cup \{d\}$ describing a configuration which can be reached by a correct computation in Π. Let us now consider the regular language

$$\begin{aligned} R = {} & (a_1, 1)^*(a_2, 1)^* \dots (a_g, 1)^* d (a_1, 2)^*(a_2, 2)^* \dots (a_g, 2)^* d \dots \\ & d(a_1, m)^*(a_2, m)^* \dots (a_g, m)^* d (a_1, 0)^*(a_2, 0)^* \dots (a_g, 0)^*. \end{aligned}$$

We denote by $Perm(L)$ the permutation closure of a language L. Consider also the morphism h which maps each symbol $(a, i), a \in V, 0 \le i \le m$, into a, and also maps d into d. Then we obtain

$$econ(\Pi) = h(R \cap Perm(\partial_d(L(G)))).$$

Indeed, the left derivative ∂_d removes the leftmost occurrence of the symbol d (the symbol obtained from Z), $Perm$ arbitrarily rearranges the symbols, the intersection with R selects only those strings which have the symbols in the "correct" place, first the multiset from the first membrane, then the multiset from the second membrane, and so on, with the symbols in the standard order; finally, the morphism h returns the symbols (a, i) to a. (Note that this morphism is, in fact, a coding.)

Because the family MAT_{ac} is closed under left derivatives, intersection with regular languages, permutation, and λ-free morphisms, it follows that $econ(\Pi) \in MAT_{ac}$, and this concludes the proof. □

In the previous proof we have considered all configurations which are accessible from the initial configuration, without taking care whether or not the corresponding computations will eventually halt. In order to obtain a matrix grammar which generates only the halting accessible configurations, we have to modify the previous construction in a way which is already suggested in the proof of Lemma 8.1.1. Specifically, after introducing the symbol Z, we do not directly replace all symbols $(a, i)'$ by (a, i), but we first check whether or not any further transition is possible. This amounts to checking whether or not any rule $r_1, \dots, r_s$ is applicable to the current configuration, and this can be done in the same way as when we have checked the maximality of

the use of rules (in the phase starting with the use of the rule $X \to X_1$). That is, by using new symbols $Z_1, \ldots, Z_s$ in the same way as we have used the symbols $X_1, \ldots, X_s$, we can check whether or not the rules $r_1, \ldots, r_s$, respectively, can be applied. In the affirmative case, we introduce the trap symbol, in the negative case we (conclude that the computation halts, and we) remove the primes. We skip the technical details and we conclude by stating this observation as a lemma.

Lemma 8.1.2. *For every P system Π with symport/antiport rules used in the free mode, the language $ehalt(\Pi)$ belongs to the family MAT_{ac}.*

As we mentioned in Subsection 2.3.7, $MAT_{ac} \subset CS$, while the context-sensitive languages are recursive (the membership problem is decidable for them). Therefore, the membership problem with respect to the languages $econ(\Pi), ehalt(\Pi)$ is decidable, and this proves the following result.

Theorem 8.1.3. *The reachability of extended configurations of P systems with symport/antiport rules used in the free mode is decidable, both in the general case of considering arbitrary configurations, and when considering halting configurations.*

The proofs of the previous Lemmas 8.1.1 and 8.1.2 also imply the inclusion $NOP_*(sym_*, anti_*) \subseteq NRE$: if also erasing rules are used, then we can get the language $\{a^n \mid n \in N(\Pi)\}$, hence this language is in the family MAT_{ac}^{λ}, which is equal to RE, and this implies that $N(\Pi) \in NRE$. Note that always in the previous chapters of the book, when proving universality results of the form $NOP_*(sym_*, anti_*) = NRE$, one proves the inclusion $NRE \subseteq NOP_n(sym_r, anti_t)$, for some n, r, t, and one does not pay attention to "the opposite straightforward inclusion", which is considered a routine task. The constructions in the proofs of Lemmas 8.1.1 and 8.1.2 give a direct proof for such a "straightforward" inclusion, not in terms of Turing machines (or Chomsky type-0 grammars), but in terms of the more restrictive formalism of matrix grammars with appearance checking.

Note. Theorems 8.1.1 and 8.1.2 are from [196], while Theorem 8.1.3 is from [193].

8.2 Unary Systems

We pointed out in Section 2.3 that the computational universality implies that the hierarchy on the number of membranes cannot be infinite. Actually, as we have seen in the previous chapters of the book, most of the hierarchies with respect to the number of membranes collapse at a very low level (the biggest number of membranes which appears in a universality theorem is six: Theorems 3.6.5, 5.2.10, 5.2.12, 5.2.16), and it is possible that these results

can be improved (we have not mentioned the case of Theorem 5.3.4, where no bound on the number of membranes is given, as we believe that it can also be improved by a careful approach to its proof). An interesting research topic in membrane computing is to find a class of systems for which the number of membranes induces an infinite hierarchy of the generated sets of numbers (or languages). Of course, it is supposed that the class of systems is non-trivial, in the sense that it is not "visibly" defined in order to get the infinite hierarchy. For instance, allowing only a bounded number of copies of objects in each region – excepting the output one, if we work in the case of internal input – obviously induces an infinite hierarchy.

On the other hand, for systems of a general form, Theorem 3.3.1 shows that the hierarchy on the number of membranes collapses at the second level. Thus, it is not easy to find a class of systems which are at the same time not "too general" and "non-trivial", for which we can obtain an infinite hierarchy.

In what follows, we present a class of systems for which the hierarchy on the number of membranes is infinite, but we still consider that the problem remains *open* (Problem Q28): the definition is not completely innocent, as we impose three rather strong restrictions: to work with only one object, to have a membrane structure described by a linear tree, and to start with objects only in the elementary membrane, the central one.

Such systems using only one object can have as the output either a number or a string, and there is no difference between the two ways to interpret the output (on the one-letter alphabet a language is precisely identified by its length set). Of course, without catalysts we cannot do too much, but when membrane dissolution is possible, we can compute non-trivial sets of numbers – even when the membrane structure is linear, with only one membrane in each level. The language from Example 3.7.5, Section 3.7, illustrates this possibility.

We do not investigate systematically here the unary P systems of various types; this remains a *research topic* (Problem Q29; an interesting case can be a sort of "quasi-unary" systems which can also use catalysts together with the unique object), but we only answer the above-mentioned problem for a restricted class of systems, those working with symbol–objects, of type $(ncoo, in, \delta)$, with a linear membrane structure, and having objects at the beginning of a computation only in the central membrane, the elementary one. For uniformity with Example 3.7.5, which will be used in the proof, we work with the external mode of defining the output, hence considering the symbols which leave the system, but the result is the same if we consider the internal output, and the necessary changes in the proof are obvious. In order to show that we work with systems of a restricted form in what concerns the membrane structure and the initial multisets, we add the letter "r" in front of the family of generated sets of numbers.

Theorem 8.2.1. $rNOP_m(ncoo, in, \delta) \subset rNOP_{m+1}(ncoo, in, \delta)$, *for all* $m \geq 1$.

Proof. The inclusions are obvious. For $m = 1$ the strictness follows from the fact that systems with only one membrane cannot use the action δ, hence they generate only semilinear one-letter languages (Theorem 3.3.2), but systems with two membranes can already generate all E0L languages (Theorem 3.4.1).

For $m \geq 2$, let $p_1, p_2, \ldots, p_m$ be different prime numbers (clearly, all of them are at least 2) and consider the system (of degree $m+1$)

$$\Pi = (\{a\}, [_1[_2[_3 \cdots [_{m+1} \;]_{m+1} \cdots]_3]_2]_1, \lambda, \ldots, \lambda, a, R_1, \ldots, R_{m+1}),$$

with

$$R_1 = \{a \to (a, out)\},$$
$$R_{i+1} = \{a \to a^{p_i},\ a \to a^{p_i}\delta\},\ 1 \leq i \leq m.$$

As we pointed out in Example 3.7.5, we have

$$N(\Pi) = \{p_1^{i_1} p_2^{i_2} \ldots p_m^{i_m} \mid i_j \geq 1, 1 \leq j \leq m\}.$$

Consequently, $N(\Pi) \in rNOP_{m+1}(ncoo, in, \delta)$.

Let us assume that this set can be generated by a P system Π' with at most m membranes, arranged vertically, and starting with objects only in the elementary membrane.

We denote by $prim_m$ the set $\{1, p_1, \ldots, p_m\}$, and for a number t we denote by $fact(t)$ the set of prime factors of t to which we add the number 1.

Claim 1. *The skin membrane of the system Π' has only one rule, of the form $a \to (a, out)$.*

If the skin membrane contains no rule of the form $r_1 : a \to (a^s, out)$, then we get no output, hence the existence of at least one rule of this form follows. We can use the same rule r_1 for sending out all the symbols of a string, hence we must have $fact(s) \in prim_m$. However, if all rules $a \to (a^s, out)$ have $s \neq 1$, then the factors of s will appear arbitrarily many times in each number of the form $p_1 \ldots p_{i-1} p_i^{n_i} p_{i+1} \ldots p_m$, with n_i arbitrarily large, which is not possible. Therefore, we have at least a rule $r_1 : a \to (a, out)$.

Assume that we have two rules in the skin membrane, $r_1 : a \to (a, out)$ as above, and some $r_2 : a \to u$. Consider a computation in Π', which produces a number $q = p_1^{n_1} \ldots p_m^{n_m}$ with all $n_1, \ldots, n_m$ arbitrarily large. Let us change this computation in two ways, producing two different computations. First, always use the rule r_1 in the skin membrane. This is correct, and what we get should be a number g from $N(\Pi)$. Second, instead of always using the rule r_1, let us use the rule r_2 exactly once. If the rule r_2 is of the form $a \to (a^i, here)(a^j, in)(a^k, out)$, for some $i, j, k \geq 0$, then at the next step we send out the j symbols from the skin membrane by the same rule r_1, we return to the skin membrane the i symbols sent to the lower membrane, and then we send out of the system the i' symbols received in this way from the lower membrane, again using the rule r_1. The string we obtain in this case

has the length $g' = g - 1 + k + j + i'$, for some i' which is smaller than $i \cdot h$, where $h = \max\{l \mid a \to (a^l, out)$ or $a \to (a^l, \alpha)\delta$ is a rule from the membrane directly inside the skin membrane$\}$. Both g and g' should be numbers of the form $p_1^{n_1} \dots p_m^{n_m}$ with all $n_1, \dots, n_m$ arbitrarily large. Their difference is also arbitrarily large, hence it cannot be equal to the bounded value of $k + j + i' - 1$. The contradiction shows that the skin membrane contains only the rule $a \to (a, out)$.

Claim 2. *No internal membrane of Π' contains rules which use the command "out" – that is, the communication of objects from bottom up, to the skin membrane, is done only by dissolving membranes.*

Because the skin membrane does not increase the number of copies of a, but just passes them to the environment, this means that it receives from the inner membrane a number of copies of a which is of the form $p_1^{n_1} \dots p_m^{n_m}$. In order that any membrane can halt its work, we have to send all its objects outside it by commands (a, out), or to dissolve the membrane, or to use a combination of these operations. Consider the membrane placed immediately inside the skin membrane. If we have a rule of the form $a \to (a^s, out)$ in it, then we can repeat the above reasoning: take a computation leading to a number g of the form $p_1^{n_1} \dots p_m^{n_m}$ with all $n_1, \dots, n_m$ arbitrarily large, change the computation, first using the rule $a \to (a^s, out)$ for all symbols, and then for all but one, which is processed by another rule. This "another rule" should be identical to $a \to (a^s, out)$, otherwise the two numbers cannot be both in $N(\Pi)$. If we have a rule of the form $a \to (a^i, in)(a^j, here)(a^s, out)$, with $j \geq 1$, or with $i \geq 1$, then the computation never stops without using either a rule $a \to (a^t, out)$ – and we have seen that this case is contradictory – or a rule which dissolves the membrane.

Thus, we need at least a rule which dissolves the membrane. If it also contains symbols having associated commands *out*, then these commands can be replaced by *here* and nothing is changed. If the rule also contains symbols (a, in), then we proceed as above and we get a contradiction (a bounded number of symbols can go to a lower membrane and come back, hence this number is not sufficient to equal the difference between two different numbers of the form $p_1^{n_1} \dots p_m^{n_m}$ with all $n_1, \dots, n_m$ arbitrarily large).

We conclude that we have a rule of the form $a \to a^s\delta$ in the membrane which is placed directly inside the skin membrane, and that only by using rules of this form can we send symbols to the skin membrane.

The argument can now be applied to the membrane on the second level from the skin, and so on and so forth, until reaching the elementary membrane, and we conclude that each membrane contains a rule of the form $a \to a^s\delta$, and no rule contains a symbol (a, out).

Claim 3. *Each membrane contains two rules of the forms $a \to a^s$ and $a \to a^s\delta$.*

If a membrane contains only a rule of the form $a \to a^s\delta$, then that membrane can be removed. We start again from the skin membrane. If the directly

inner membrane, say, with label i, contains only such a dissolving rule, then it is dissolved immediately after receiving any symbol from the inner membrane, and this is done at the moment of using the rule $a \to a^t\delta$ in the inner membrane. We replace this rule with $a \to a^{st}\delta$, we remove the membrane i, and the generated set of numbers is not changed. (It is important that in the skin membrane we have only the rule $a \to (a, out)$.)

If in the membrane placed directly in the skin membrane, besides the rule $r_1 : a \to a^s\delta$ we also have a rule $r_2 : a \to a^t\delta$, or $r_3 : a \to a^t$, with $s \neq t$, then we proceed as above, changing slightly one computation which produces an arbitrarily large number (first, use only r_1, then use r_1 for all symbols but one, which is processed by r_2 or r_3), and we get that, provided that the second rule exists, it must have $s = t$.

Therefore, the membrane directly inside the skin membrane only multiplies by s the number of copies of a received from the inner membranes.

The reasoning can then be repeated for the membranes from the inner levels of the membrane structure, hence we conclude that all of them contain two rules, of the forms $a \to a^s, a \to a^s\delta$.

This implies that we have $fact(s) \in prim_m$ for all rules $a \to a^s$ and $a \to a^s\delta$. Because a number of the form $p_1^{n_1} \dots p_m^{n_m}$ has n_i factors equal to p_i, and n_i can be arbitrarily large, it follows that the rules of Π all have prime numbers $p_1, \dots, p_m$ among the factors of various s, the number of copies of a they introduce. On the other hand, there are in $N(\Pi)$ numbers of the form $g = p_1p_2 \dots p_{i-1}p_i^{n_i}p_{i+1} \dots p_m$, hence containing only one factor p_j for all $j \neq i$, and arbitrarily many factors p_i. Consequently, there is a rule of the form $a \to a^{p_i^j}$, with or without δ, with $j \geq 1$, for each $i = 1, 2, \dots, m$. Two rules of this form, for different primes p_i, cannot be present in the same membrane, which means that we have at least m membranes in Π', each of them holding two rules $a \to a^{p_i^j}, a \to a^{p_i^j}\delta$. Together with the skin membrane, we have at least $m + 1$ membranes, and this concludes the proof. □

Note. Theorem 8.2.1 is an improved version of a result from [61], where a series of similar results are given, but using still more restrictive classes of P systems, both with internal and with external output. The ideas from [61] are considered in [109] for string–objects, and infinite hierarchies on the number of membranes are obtained, in the same very restricted framework.

8.3 A Representation of Context–free Languages

Both in Section 3.7 and in Section 4.4 we have obtained representations of regular languages and of recursively enumerable languages by using P systems with symbol–objects, but representations of this type were not obtained also for other families of languages from the Chomsky hierarchy. The problem is challenging, because of the qualitative difference between the internal data

structure (multisets, hence numbers) and the result (a language, where the ordering of symbols in strings matters), and it remains to be investigated further. However, the membrane systems essentially use a further ingredient (a further data structure), which was never sufficiently exploited in what concerns the result of a computation: the membrane structure itself.

In all previous chapters of the book, the purpose of a computation was related to the objects present in the regions of the system, symbols or strings, while the membrane structure was only a tool used in the computation. Let us reverse the perspective, and take the membrane structure – and the tree describing it – obtained at the end of a halting computation as the result of that computation, ignoring at the end the objects present in the system or sent out of it. In the case of systems which use only the action of membrane dissolving, the set of all trees associated in this way with a system Π is finite, but this is not the case when we can make use of membrane division or of membrane creation. Moreover, further operations with membranes can be considered, for instance modelling the biological phenomena of exocytosis and endocytosis (in Section 8.5 we will briefly discuss some operations of this kind). We do not address the general problem here, it remains as a promising *research topic* (Problem Q30), but we deal with a related one, about a possibility of representing the context-free languages by making essential use of the membrane structure.

Still, the representation is "classic"; that is, we work with symbol–object P systems, capable of manipulating the membranes to a larger extent than in the previous chapters, and generating strings in the external mode, by sending objects into the environment.

Specifically, we consider systems $\Pi = (O, H, \mu, w_1, \ldots, w_m, R)$, where O is an alphabet of objects, H is a set of labels for membranes, of all posssible membranes, μ is a membrane structure of degree m, $w_1, \ldots, w_m$ are the multisets of objects present in the initial configuration in the m regions of the membrane structure, and R is a finite set of rules, of the following four forms:

1. $[_i a \to v]_i$, where a is an object and v is a string representing a multiset of objects;
2. $[_i a]_i \to [_i\ ]_i b$, where a and b are objects;
3. $[_i\ ]_i \to [_k\ ]_k$, where i and k are labels of possible membranes;
4. $[_i\ ]_i \to [_j[_k\ ]_k]_j$, where i, j, and k are labels of possible membranes.

Note that the rules of types 1 and 2 are as already used in P systems with active membranes, but rules of type 3 allow the change of the label of a membrane (hence the change of rules to be used for the associated objects), and rules of type 4 allow the duplication of a membrane, but not as in the case of membrane division, with the two new membranes placed at the same level, but with the two new membranes placed one into the other; also, the labels of the new membranes can be new.

In the case of rules of types 3 and 4, the contents of the membrane from the left-hand side of the rule, with label i in the above writing, are transferred into the central membrane from the right-hand side of the rule, namely that with label k in both cases. This is valid for both the objects and the membranes from the former membrane i. The rules are applied in the maximally parallel manner, both for objects and membranes, hence the inner objects and membranes can evolve at the same time as the application of a rule of type 3 or 4 for a membrane i as above. As was the case for systems with active membranes, the use of a rule of type 1 counts as involving the object a but not the membrane i, for a rule of type 2 both the object and the membrane are counted as used, while for rules of types 3 and 4 only the membranes are involved in the application of rules.

Note that we do not have electrical charges associated with the membranes, although this feature can be added without any difficulty.

Let us denote by $NOP_{m,n}(activ'), LOP_{m,n}(activ')$, the families of sets of numbers and the family of languages generated in the external mode by systems with the type of rules considered above, starting with at most $m \geq 1$ membranes, and using at most $n \geq 1$ types of membranes. We do not investigate here the size of these families, but we only give one result, a representation of context-free languages by means of such systems, proving the unexpected usefulness of operations with membranes as described by rules of types 3 and 4.

First, let us examine an **example** also of interest for our main result: consider an E0L system $G = (V, T, w, R)$ and construct the P system

$$\begin{aligned}
&\Pi = (V, \{1,2\}, [_1\ ]_1, w, R'), \text{ with the following rules:}\\
&R' = \{[_1 a \to v]_1 \mid \text{for } a \to v \in R\}\\
&\quad \cup \{[_1\ ]_1 \to [_2\ ]_2\}\\
&\quad \cup \{[_2 a]_2 \to [_2\ ]_2 a \mid \text{for } a \in T\}\\
&\quad \cup \{[_2 a \to a]_2 \mid \text{ for } a \in V - T\}.
\end{aligned}$$

One can easily see that $length(L(G)) = N(\Pi)$: as long as the unique membrane of the system has label 1, we simulate the rules of G, in the parallel manner; after changing the label of the membrane to 2, we send out all terminal symbols and evolve the symbols from $V - T$ forever, and this guarantees that we stop only after simulating a terminal derivation in G.

Theorem 8.3.1. $CF \subset LOP_{3,*}(active')$.

Proof. In view of the previous example, we only have to prove the inclusion.

Let $L \subseteq T^*$ be a context-free language, and let $G = (N, T, S, R)$ be a context-free grammar in the Chomsky normal form, generating the mirror image of L. That is, the rules from R are of the forms $A \to BC, A \to a$, for $A, B, C \in N$ and $a \in T$ (remember that we ignore the empty string when comparing two languages). We construct the following P system:

$$
\begin{aligned}
\Pi &= (T \cup \{f, f', f'', \#\}, N \cup T \cup \{0,1\}, [_0[_S[_1\ ]_1]_S]_0, f, R'),\\
R' &= \{[_A\ ]_A \to [_B[_C\ ]_C]_B \mid \text{for } A \to BC \in R\}\\
&\quad \cup \{[_A\ ]_A \to [_a\ ]_a \mid \text{for } A \to a \in R\}\\
&\quad \cup \{[_1 f \to f]_1,\\
&\qquad [_1 f]_1 \to [_1\ ]_1 f\}\\
&\quad \cup \{[_a f \to f' a]_a,\\
&\qquad [_a f' \to f'']_a,\\
&\qquad [_a f'']_a \to [_a\ ]_a f \mid \text{for } a \in T\}\\
&\quad \cup \{[_A f \to \#]_A \mid \text{for } A \in N\}\\
&\quad \cup \{[_\alpha \# \to \#]_\alpha,\\
&\qquad [_\alpha a]_\alpha \to [_\alpha\ ]_\alpha a \mid \text{for } a \in T \text{ and } \alpha \in N \cup T \cup \{0\}\}.
\end{aligned}
$$

The idea behind this construction is to simulate the rules of the grammar G by operations on the membrane structure of the system Π, and to this end we label the membranes with the symbols from $N \cup T$. Initially, besides the membranes with labels 0 and 1, we have only one membrane with the label S, the axiom of G. In the central membrane, that with label 1, we have the unique object f, which can evolve an arbitrary number of steps without changing. At any moment, the object f can exit membrane 1. If it reaches a membrane labelled with a non-terminal of G, then the trap symbol # is introduced, which can evolve forever in all membranes, hence the computation will never stop. If f reaches a membrane with a label $a \in T$, then it introduces the objects f' and a. The terminal a can go out of the system, crossing unmodified each membrane, irrespective of whether the label of the membrane is from N or from T. In its turn, f' is first replaced by f'' (at this time, a will exit, as the rule $[_a f' \to f]_a$ does not count as using the membrane), then f'' exits the membrane returned to f, hence the operation of sending out the labels of membranes continues. It is important to note that the symbols a are going out in the ordering of membranes, starting from the central membrane, hence outside we get the string of the labels of membranes in the reversed order, first the central label and at the end the label of the membrane which is the last in the skin. Consequently, if all labels of membranes are from T, then we get a halting computation.

Let us now examine how the membrane structure evolves. Clearly, we always have a linear membrane structure – the membranes are placed one into the other. Thus, assume that we have a membrane structure described by a parenthetic expression

$$[_0[_{\alpha_1} \cdots [_{\alpha_2}[_A[_{\alpha_3} \cdots [_{\alpha_4}[_1\ ]_1]_{\alpha_4} \cdots]_{\alpha_3}]_A]_{\alpha_2} \cdots]_{\alpha_1}]_0,$$

and we use a rule of the form $[_A\ ]_A \to [_B[_C\ ]_C]_B$, corresponding to a rule $A \to BC$ from R. We obtain the membrane structure described by the expression

$$[_0[_{\alpha_1}\cdots[_{\alpha_2}[_B[_C[_{\alpha_3}\cdots[_{\alpha_4}[_1\ ]_1]_{\alpha_4}\cdots]_{\alpha_3}]_C]_B]_{\alpha_2}\cdots]_{\alpha_1}]_0.$$

If the rule used is $[_A\ ]_A \to [_a\ ]_a$, corresponding to $A \to a \in R$, then we obtain the expression

$$[_0[_{\alpha_1}\cdots[_{\alpha_2}[_a[_{\alpha_3}\cdots[_{\alpha_4}[_1\ ]_1]_{\alpha_4}\cdots]_{\alpha_3}]_a]_{\alpha_2}\cdots]_{\alpha_1}]_0.$$

The rules of types 1 and 2 do not change the membrane structure, but only the objects present in its regions.

It is easy to see that the string of labels from the first half of the expressions, starting after $[_0$ and ending before $[_1$, is exactly the sentential form generated in the grammar G by using the rules from R which correspond to the rules from R' which were used in the computation of Π. Consequently, all sentential forms of G can be obtained in this way, and, conversely, all membrane structures of computations of Π correspond to derivations in G. If a derivation in G is terminal, then the labels of the membranes precisely identify the string generated by that derivation. If the object f leaves membrane 1 after terminating a derivation in G, then it will release this terminal string (in the mirrored form), and will finish its work in the skin membrane, when the computation stops. If the object f exits membrane 1 "too early", and it reaches a membrane which is labelled with a non-terminal from N, then the computation will never stop, even if the membranes can further evolve in such a way as to reach a terminal derivation in G.

We conclude that $L(\Pi) = mi(L(G)) = L$, and this proves the inclusion $CF \subseteq LOP_{3,*}(active')$. □

Note. The idea of using the membrane structure in order to encode information about strings was first considered in [15], but using operations completely different from those considered here. Based on the representation of strings of context-free languages by means of membrane structures of P systems, in [15] several computability problems about context-free languages are addressed in this framework: looking for strings of a given length, or bounded by a given constant, comparing two strings, etc.

8.4 Valuating the String–Objects

The idea of associating electrical charges to membranes and to objects, and to use these charges as communication commands, can be extended from symbol–objects to string–objects. Moreover, while in the case of symbol–objects the polarization is assigned to objects by the rules used for evolving them, in the case of strings we can compute the polarization as a result of the structure of the string. This looks rather attractive, because in this way the destination of a string fully depends on its structure, which is very much as in biology.

Also the manner of associating a polarization, hence a sign $+, -$, or 0 (for neutral charge), is very natural: just take a morphism from strings to the set of integers and consider the image of the string through this morphism (the sum of values of all symbols from the string); if this image is a negative (positive) number, then we say that the string is negatively (positively) charged.

Formally, a *rewriting P system* (of degree $m \geq 1$) *with valuation* is a construct

$$\Pi = (V, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m, \varphi),$$

where, as usual, V is the alphabet of the system, μ is a membrane structure with m membranes, having associated electrical charges $+$ and $-$, $M_1, \ldots, M_m$ are finite languages over V, representing the strings initially present in the regions $1, 2, \ldots, m$ of the system, $R_1, \ldots, R_m$ are finite sets of context-free rules over V associated with the regions of μ, and φ is a morphism from the monoid $(V^*, \cdot, \lambda)$ to the additive monoid $(\mathbf{Z}, +, 0)$ (called the *valuation mapping* of Π).

The transitions are defined as usual for rewriting membrane systems, with the following two differences: (1) the rules are always applied only in the ends of the strings, rewriting either the first or the last symbol of each string; (2) the communication of strings from one region to another one is controlled by the charges of membranes and of strings, in the following way. If $\varphi(x) > 0$, then we say that the string is positively charged, if $\varphi(x) < 0$, then the string is negatively charged, if $\varphi(x) = 0$, then the string has no charge. If an uncharged string is produced in the skin membrane, then it leaves the system, but in any other membrane such a string "vanishes". If in a given region, associated with a membrane of a given charge α, we produce a string of the opposite charge, then it remains in the same region (and it can be rewritten again). If we produce a string of the same charge α, then the string must exit; it is then replicated in as many copies as there are adjacent membranes with the opposite polarization, and one copy is sent to each of these membranes. If there is no adjacent membrane of the opposite polarization, then the string "vanishes". (The deadlocks are solved by removing the strings from the system.)

Note two important facts here: a string which is "rejected" from its region, because it has the same polarization with the host membrane, will go to *all* neighbouring regions of the opposite polarization, both to the lower level and to the upper level with respect to the membrane where the string was produced; thus, replication is defined in an "objective" manner, in the interaction of strings with membranes, not by rules applied to strings. Then, we have the new feature of making the strings disappear, if they are electrically neutral, with the exception of the skin region, where the neutral polarization is used to send the strings out. The restriction on using the rules only in the ends of the strings can be removed, but we follow the papers [147, 148] and we preserve it in what follows.

The family of all languages $L(\Pi)$, generated as above by rewriting P systems with valuation, Π, of degree at most $m \geq 1$, is denoted by $LSP_m(rw_{lr}, valuat)$. When the rewriting is allowed only in the left-hand or only in the right-hand end of the strings we write $LSP_m(rw_l, valuat)$, $LSP_m(rw_r, valuat)$, respectively.

The rewriting in the ends of a string is sufficient in order to generate more than the regular languages.

Theorem 8.4.1. *$REG \subset LSP_m(rw_\alpha, valuat)$, for each $\alpha \in \{l, r\}$.*

The inclusion is easy to prove (we take a right-linear or a left-linear grammar, and we associate the value 0 with each terminal and the value -1 with each non-terminal, then we consider a P system with only one membrane, positively charged, having the same rules as the grammar).

The properness is proved by the P systems with one membrane, positively charged, containing initially the string A and the rules $A \to aA$, $A \to aB$, $B \to bB$, $B \to \lambda$, in the case of $LSP_m(rw_r, valuat)$, and the rules $A \to Aa, A \to Ba, B \to Bb, B \to \lambda$, in the case of $LSP_m(rw_l, valuat)$. Consider also the valuation mapping φ defined by $\varphi(a) = -2$, $\varphi(b) = 2$, $\varphi(A) = \varphi(B) = -1$. It is easy to note that both these systems compute the language $\{a^n b^n \mid n \geq 1\}$, which is not regular.

As expected, systems allowed to work in both ends of strings (and using two membranes) are computationally universal.

Theorem 8.4.2. *$LSP_2(rw_{lr}, valuat) = RE$.*

The proof can be found in [148]; it starts from a type-0 grammar in the Kuroda normal form and makes intensive use of the valuations in ensuring the correct simulation of the context-sensitive rules of the grammar by the context-free rules of the system with valuations obtained; moreover, the rotate and simulate technique from the splicing area is used, so that we can work only in the ends of the strings.

The fact that the string–objects can be replicated during the computation can be used in order to get an exponential space for solving **NP**-complete problems, in a way rather similar to that already discussed in Section 7.4. Specific constructions of "P algorithms" for the 3-colourability problem for directed graphs and for the so-called bounded Post Correspondence Problem can be found in [147] (for the reader's convenience, we recall that this latter problem asks whether or not, given two lists of non-empty strings over a given alphabet with at least two symbols, $(u_1, u_2, \ldots, u_n)$ and $(v_1, v_2, \ldots, v_n)$, there exists a sequence $i_1, i_2, \ldots, i_k$ of integers between 1 and n such that $u_{i_1} u_{i_2} \ldots u_{i_k} = v_{i_1} v_{i_2} \ldots v_{i_k}$; the problem is undecidable when no bound on k is imposed and **NP**-complete when a constant given in advance bounds k). Also, HPP and SAT are treated in [148].

8.5 Systems with Enhanced Membrane Handling

The main operations with membranes used in the previous chapters were the dissolution of a membrane, dividing a membrane, and creating a new membrane. In Section 8.3 we used two further operations: changing the label of a membrane (which, in some sense, cannot be considered a real operation with membranes, but on their labels) and creating a new membrane inside (or outside, it is not important) an existing membrane, also changing the label of the former membrane. In Section 9.2 we will also introduce the possibility of creating a membrane (a vesicle), which is meant to carry a given content to a specified destination.

In total, there are not a very large variety of operations, although the number of possibilities is much larger. The motivation can come both from mathematics and from biology. In what follows we will start from a somewhat intermediate point, the paper [87], where several experiments in DNA computing are interpreted in terms of membrane computing. In this context, several new operations with membranes are proposed, some of them of a kind which is completely different from those mentioned above. We present them as formalized in [145].

For instance, a powerful operation is that suggested by the following rule:

$$[_i a_1 a_2 \ldots a_n]_i \to [_i a_1]_i [_i a_2]_i \ldots [_i a_n]_i,$$

where i is the label of a membrane and $a_1 a_2 \ldots a_n$ is a string, composed of n symbols; by this operation, the membrane with label i is replicated in as many symbols as the string has, and one symbol is placed in each new membrane.

We will not discuss this operation further, but we will enter into some details about the following operations, some of them being new forms of existing operations:

- *Merge:* the fact that two membranes, with labels i and j, can be merged into a single membrane, with label k, provided that they contain the multisets of objects M_i, M_j, respectively, is indicated by

  $$[_i M_i]_i [_j M_j]_j \to [_k \ ]_k.$$

  The contents of the former membranes (objects and inner membranes included) are put together in the new membrane. Of course, in order to perform a merging operation, the two membranes must be adjacent.
- *Divide:* a membrane i, containing the multiset of objects M, can be divided into two new membranes by a rule

  $$[_i M]_i \to [_j \ ]_j [_k \ ]_k.$$

  The contents of the former membrane are reproduced in the two new membranes. Note that one gives a multiset M which controls the operation, not

a single object, as in Section 7.2; moreover, the two new membranes can have labels which are different from the label of the divided membrane.

- *Separate:* if we do not want to replicate the contents of a membrane, but to separate its objects according to a given property π, then we can use a rule

$$[_i\ ]_i \to [_j\pi]_j[_k\neg\pi]_k.$$

The objects which have the property π are placed in membrane j, the objects which do not have the property π are placed in membrane k.

- *Create:* we can also separate the objects of a membrane in such a way that the objects with the property π are put into a new membrane created inside it, by using rules of the form

$$[_i\ ]_i \to [_i[_j\pi]_j\neg\pi]_i.$$

The objects which do not have the property π remain outside the new membrane.

The previous rules were formulated for any type of membrane, but they are natural (and mathematically elegant) for elementary membranes. In particular, the skin membrane should be handled with care; for instance, by a division or a separation operation, we can duplicate the system itself. From a computational point of view, this does not make much sense (we have to change the definition of the computation, of its result, etc.), but from other points of view, such as the replication of a system, this can be of central interest.

In what follows, all operations are allowed only for elementary membranes.

Moreover, we consider string–objects, processed by rewriting rules, which, however, have restrictions on the place where they can be used: in any place, in the leftmost position of a string, or in the rightmost position. Formally, we consider rules of the forms $a \to u$ (used freely, in any place), $a \to_{pref} u$ (used only if a is the first symbol of the string to be processed), and $a \to_{suf} u$ (used only if a is the last symbol of the string).

In the same way as each string–object is processed at each step by only one rule (the strings which cannot be rewritten are passed unchanged to the next configuration), also the membranes are processed by at most one rule each: if there are some rules which can be applied to a membrane, then one of them is non-deterministically chosen and applied, but if no rule can be applied to a given membrane, then it remains unchanged.

Note that we have mentioned no communication rule, but only evolution rules and membrane handling rules. Because we cannot send strings out of the system, the result of a computation will be the set of terminal strings present at the end of a computation in a specified output membrane. The family of languages computed in this way by systems with at most m membranes present in any configuration, out of n possible types of membranes, using, for instance, the operations of creating and dissolving membranes, is denoted

by $LSP_{m,n}(rw_{mode}, create, \delta)$, where *mode* is *any* when the rewriting can be done in any place, or *lr* when only the first or the last symbol of a string can be rewritten. When any of m or n above is not bounded in advance, we replace it with $*$.

We mention here three results about such families.

Theorem 8.5.1. $LSP_{3,*}(rw_{lr}, create, \delta) = LSP_{4,*}(rw_{any}, create, \delta) = LSP_{3,9}(rw_{lr}, create, separate, \delta) = RE$.

The proof of the first equality starts from a Chomsky type-0 grammar G in the Kuroda normal form and constructs a P system which simulates the derivations of G by using the "rotate and simulate" technique, while the other two equalities are based on the characterization of recursively enumerable languages by means of the matrix grammars with appearance checking in the binary normal form.

We omit these proofs here, but we enter into some details about the usefulness of these operations from a computational complexity point of view. As expected (we have membrane division at our disposal), we can solve SAT in linear time, which is not a new result, but its proof is still of interest because of the way of using the new operations with membranes.

Thus, consider m clauses C_i, involving n variables $x_1, \ldots, x_n$, and let us construct the P system with the initial membrane structure

$$[_s[_m[_{m-1} \cdots [_1[_0\ ]_0]_1 \cdots]_{m-1}]_m]_s,$$

with the strings a_1, d_1 placed in membrane 0; consider also the following rules:

$$\begin{aligned}
&[_0 d_i]_0 \to [_{i1}\ ]_{i1}[_{i2}\ ]_{i2}, 1 \le i \le n,\\
&a_i \to t_i a_{i+1},\\
&d_i \to d_{i+1}, \text{ in membrane } i1, 1 \le i \le n,\\
&a_i \to f_i a_{i+1},\\
&d_i \to d_{i+1}, \text{ in membrane } i2, 1 \le i \le n,\\
&[_{i1} d_{i+1}]_{i1}[_{i2} d_{i+1}]_{i2} \to [_0\ ]_0, 1 \le i \le n-1,\\
&[_{n1} d_{n+1}]_{n1} \to d_{n+1},\\
&[_{n2} d_{n+1}]_{n2} \to d_{n+1},\\
&[_i w]_i \to [_i\ ]_i w, \text{ if } w \text{ satisfies } C_i, 1 \le i \le m,\\
&[_s w]_s \to [_s\ ]_s w, \text{ for all } yes\ w.
\end{aligned}$$

By division (and replication) and making use of membranes $i1, i2$, for $1 \le i \le n$, in membrane 0 one generates all truth–assignments in the form of strings $w \in \{t_i, f_i \mid 1 \le i \le n\}^*$ of length n (this takes $3n$ steps); when this operation is completed and membranes $n1, n2$ are dissolved (the task of counting to n and to dissolve membranes $n1, n2$ is covered by the counters $d_i, 1 \le i \le n+1$), the truth–assignments can pass through membranes provided that

they satisfy the clauses associated with membranes (m more steps). In this way, in $3n+m+1$ steps we get a string outside the system if and only if the set of clauses is satisfiable.

The communication rules can be avoided in the following way. Start from only one membrane inside the skin, membrane 0 as above, but with the objects ta_1, d_1 inside; use the previous rules for generating the truth-assignments. After dissolving membranes $n1, n2$ (hence membrane 0 is not created again), we use the following rules (including separation rules):

$[_s\ ]_s \to [_s[_{c1} w$ satisfies $C_1\,]_{c1}$otherwise$]_s$,
$[_{ci}\ ]_{ci} \to [_{ci}[_{c(i+1)} w$ satisfies $C_{i+1}\,]_{c(i+1)}$otherwise$]_{ci}, 1 \leq i \leq m-1$,
$t_i \to \lambda$, and
$f_i \to \lambda, 1 \leq i \leq n$, in the membrane with the label cm,
$[_{ci} t]_{ci} \to t, 1 \leq i \leq m$.

The truth-assignments which satisfy the clauses are encapsulated in inner membranes which grow iteratively until producing a membrane with the label cm; in this membrane all symbols $t_i, f_i, 1 \leq i \leq n$, are removed, which makes it possible to dissolve all membranes $cj, 1 \leq j \leq m$, under the influence of the object t. Thus, we get t in the skin membrane (after $3n+2m$ steps) if and only if the set of clauses is satisfiable.

8.6 Brief Excursion Through the Literature

Because there are several further theoretical developments which, on the one hand, are of (mathematical or computer science) interest, but, on the other hand, are not on the lines considered in the previous chapters, we briefly present them, without entering into details, but only giving the main ideas and the main results.

8.6.1 Generalized Sequential Membrane Systems

We start by introducing one of the most general types of string-object P systems, investigated in several papers by R. Freund and his collaborators: [58, 59, 60, 65, 66, 227].

Let X be a class of rewriting systems (for instance, context-free grammars, written in the form (V, T, S, P), with $V = N \cup T$ being the total alphabet). A *generalized P system of type* X is a construct Π of the form

$$\Pi = ((V, T, S, P), \mu, M_0, M_1, \ldots, M_m, Op, R, i_o),$$

where:

- (V, T, S, P) is a rewriting system of type X;

- μ is a membrane structure of degree $m+1$ (with the membranes labelled by natural numbers $0, 1, \ldots, m$, where 0 is the label of the skin membrane);
- $M_0, M_1, \ldots, M_m$ are the initial contents of regions $1, 2, \ldots, m$, that is consisting of a finite multiset of strings from V^* (called objects) as well as a finite multiset of operators from Op and of rules from R; S is always an element of M_0;
- Op is a finite set of operators (which will be described in detail below);
- R is a finite set of (evolution) rules of the form $(op_1, \ldots, op_k; op'_1, \ldots, op'_l)$, with $k \geq 1$ and $l \geq 0$, where $op_1, \ldots, op_k$, $op'_1, \ldots, op'_l$ are operators from Op;
- $i_o \in \{0, 1, 2, \ldots, m\}$ is the label of the output region; always $w_{i_o} = \lambda$.

Note that the skin membrane is labelled with 0, and that we speak about an output region, not about an output membrane.

The main power of such systems lies in the operators, which can be of the following types:

- $P \subseteq Op$, that is the productions working on the objects from V^* are operators;
- $Op_0 \subseteq Op$, where Op_0 is a finite set of special symbols, which are called *ground operators*;
- $Tr_{in} \subseteq Op$, where Tr_{in} is a finite set of *transfer operators*, acting on objects from V^* of the form $(\tau_{in,k}, E, F)$, $1 \leq k \leq n$, $E \subseteq P$, $F \subseteq P$; the operator $(\tau_{in,k}, E, F)$ transfers an object $w \in V^*$ from region l into region k provided that:
  1. region l contains membrane k,
  2. every production from E could be applied to w (hence, E is a permitting transfer condition),
  3. no production from F can be applied to w (hence, F is a forbidding transfer condition);
- $Tr_{out} \subseteq Op$, where Tr_{out} is a finite set of transfer operators, acting on objects from V^* of the form (τ_{out}, E, F), $1 \leq k \leq n$, $E \subseteq P$, $F \subseteq P$; the operator (τ_{out}, E, F) transfers an object $w \in V^*$ from region l into region k provided that:
  1. region k contains membrane l,
  2. every production from E could be applied to w,
  3. no production from F can be applied to w;
- $Tr'_{in} \subseteq Op$, where Tr'_{in} is a finite set of transfer operators, working on operators from P, Op_0, Tr_{in}, and Tr_{out}, or even on rules from R; a transfer operator $\tau_{in,k}$ moves such an element from region l into region k provided that region l contains membrane k;
- $Tr'_{out} \subseteq Op$, where Tr'_{out} is a finite set of transfer operators, working on operators from P, Op_0, Tr_{in}, and Tr_{out}, or even on rules from R; a transfer operator τ_{out} transfers such an element from region l into the surrounding region.

In sum, Op is the disjoint union of P, Op_0, and Tr, where Tr itself is the (disjoint) union of the sets of transfer operators Tr_{in}, Tr_{out}, Tr'_{in}, and Tr'_{out}. In the following we shall assume that the transfer operators in Tr'_{in} and Tr'_{out} do not work on rules from R; hence, the distribution of the evolution rules is static and given by $M_0, M_1, \ldots, M_m$. If in all transfer operators the permitting and the forbidding sets are empty, then Π is called a generalized P system without transfer checking.

A *computation* in Π starts with the initial configuration with M_k being the contents of region k. A transition from one configuration to another one is performed by evaluating one evolution rule $(op_1, \ldots, op_k; op'_1, \ldots, op'_l)$ in some region k, which means that the operators $op_1, \ldots, op_k$ are applied to suitable elements in region k in the multiset sense (that is, they are "consumed" by the usage of the rule; observe that ground operators have no arguments and are simply consumed in that way); thus we may obtain a new object by the application of a production and/or we may move elements out or into inner regions by the corresponding transfer operators, yet we also obtain the operators $op'_1, \ldots, op'_l$ (in the multiset sense) in region k.

The language generated by Π is the set of all terminal objects $w \in T^*$ obtained in the terminal compartment i_o by some computation in Π.

It is interesting to note the way of handling the operators, treated in the multiset sense, and that we dispose of an underlying rewriting system, which can be of various forms.

We recall a few results about the generative power of this type of P system, following [227].

Theorem 8.6.1. *Generalized sequential P systems of type REG or CF, without forbidding transfer checking, without transfer of operators, with evolution rules of the form $(r_i; r_i)$, and with the initial objects being non-terminal symbols, are equivalent to the random context grammars of the same type and without appearance checking.*

Using transfer checking or more complex rules adds to the generative power.

Theorem 8.6.2. *Any graph-controlled grammar of any type can be simulated by a generalized sequential P system of the same type, with the membrane structure of the form $[_0[_1\ ]_1[_2\ ]_2]_0$, without permitting transfer conditions and with evolution rules of the (context-free) forms $(op_1;)$, $(op_1; op_2)$, and $(op_1; op_2, op_3)$.*

If one starts from a graph-controlled grammar without appearance checking, then the membrane structure of the simulating system is simpler, $[_0[_1\ ]_1]_0$, and no transfer condition is necessary.

Generalized P systems (using transfer operators, treating rules and operators in the multiset sense, etc.) can be considered also for other types of underlying string processing devices (V, T, S, P), for instance for splicing

systems and for cut-and-paste systems. Details can be found in the above-mentioned papers.

8.6.2 Bidimensional Objects

Up to now, the objects of P systems were symbols or strings of symbols, but it is clear that this is not at all obligatory. The style of working in a membrane system, with evolution rules applied in parallel in the compartments of a membrane structure, is of a maximal generality, both the objects which are processed and the rules applied to them can be of many forms. In particular, we can work with *bidimensional objects* (even with multi-dimensional objects), and there is a large variety of possibilities: the objects can be trees, can be graphs (arbitrary or from restricted classes), can be arrays of symbols, or can be "matrices", in the sense of [226]. The fact that we work synchronously in several regions at the same time, and having possibilities to communicate among these regions, makes membrane systems very powerful in comparison with many known picture handling devices. Because of the complex formalism needed, we do not enter into any details, and refer to [110] for a consistent attempt to use P systems in two-dimensional picture generation.

8.6.3 Membrane Systems and Stream X-machines

There are numerous distributed – parallel or not necessarily parallel – models in computer science. Many of them have a completely different style and goal than membrane systems, but many of them are computing devices in the Turing sense, often dealing with strings (languages) or with numbers. The comparison of such models with membrane systems is a potentially interesting and fruitful *research topic* (Problem Q31). The first "temptation" is to simulate a mechanism in terms of the other mechanism, thus transferring results from one area to another area (because the power of P systems of various types is well known and because this power is in many cases equal to that of Turing machines, it is possible to transfer in this way universality results from the membrane systems area to other formalisms). Perhaps more important is the transfer of notions and ideas from one domain to another domain.

In what follows, we mention two such connections, between P systems and *stream X-machines* and then with *ambient calculus*.

For the first area we will mainly use the paper [2] (further details about X-machines and their usefulness for specifying and verifying dynamic system specifications can be found, for instance, in [93] and [95]). We only recall some basic definitions, following [94].

A *stream X-machine* (SX-machine, for short, also called Eilenberg machine, because X-machines were originally investigated in [54]) is a construct

$$\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0),$$

where:

- Σ and Γ are *finite* sets called the *input* and *output* alphabet, respectively;
- Q is the *finite* set of states;
- M is a (possibly infinite) set called *memory*;
- Φ is the *type* of $\mathcal{M}$, a finite set of *basic functions* of the form

$$\varphi : M \times \Sigma \longrightarrow \Gamma \times M;$$

- $F : Q \times \Phi \longrightarrow \mathcal{P}(Q)$ is the *next state* (partial) function;
- I and T are the sets of *initial* and *terminal* states, respectively, $I \subseteq Q$, $T \subseteq Q$;
- $m_0 \in M$ is the *initial memory value.*

A *stream X-machine with relations* (SRX-machine, for short) is a device having the same elements as those of an SX-machine, but the set Φ, the type of the SRX-machine, contains relations of the form $\varphi : M \times \Sigma \leftrightarrow \Gamma \times M$ rather than functions.

A *generalized stream X-machine* (GSX-machine, for short) is an SRX-machine which has relations of the form $\varphi : M \times \Sigma \leftrightarrow \Gamma^* \times M$. In order to change its configuration, the GSX-machine will read an input symbol, discard it, and produce a *sequence* of output symbols while (possibly) changing the value of the memory.

If $\mathcal{M} = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$ is a GSX-machine, then $\mathcal{A} = (\Phi, F, I, T)$ is its *associated automaton.*

A GSX-machine with an *output delimiter* is a system with all the elements of Φ being relations of the form

$$\varphi : M \times \Sigma \leftrightarrow (\Gamma^*\{\tau\}) \times M,$$

where τ, the delimiter, is a new symbol which is not in Γ.

For a GSX-machine, each path $x = \varphi_1 \ldots \varphi_n$, $n \geq 1$, $\varphi_i \in \Phi$, in the associated automaton gives rise to a relation (the *path relation*)

$$\langle x \rangle : M \times \Sigma^* \leftrightarrow \Gamma^* \times M,$$

where $(m, s)\langle x \rangle(g, m')$ if and only if there exist $m_2, \ldots, m_n \in M$ and $(m_i, \sigma_i)\varphi_i(g_i, m_{i+1})$ for all $1 \leq i \leq n$, where $m_1 = m$, $m_{n+1} = m'$, and $s = \sigma_1 \ldots \sigma_n$, $g = g_1 \ldots g_n$. The relation corresponding to the empty path λ is $(m, \lambda)\langle \lambda \rangle(\lambda, m)$, for any $m \in M$.

In the case of an SX-machine the path relation becomes a path function.

Given an SX-machine $\mathcal{M}$, we define the relation

$$f : \Sigma^* \leftrightarrow \Gamma^*$$

by $g \in f(s)$ if and only if there are $q_0 \in I, q \in T, m' \in M$, and a path $q_0 \xrightarrow{x} q$ with $(m_0, s)\langle x \rangle(g, m')$.

We say that f is the relation computed by $\mathcal{M}$. For a subset $Y \subseteq \Sigma^*$, we denote by $f_{|Y}$ *the restriction* of f to Y, that is $g \in f_{|Y}(s)$ if and only if $g \in f(s)$ and $s \in Y$.

In a similar way one defines the *relation computed* by an SRX-machine and by a GSX-machine.

In [2] it is proved that any P system with string–objects processed by replicated rewriting (with an arbitrary replication factor) can be simulated by an SX-machine. Then, *communicating* SX-machines are considered, as well as communicating SX-machines *with replicated rewriting*, which can simulate a P system with replicated rewriting in a much easier way than standard stream X-machines. The comparison of complexity (size, time, communication) of a P system and of an equivalent SX-machine, with routing and broadcast operations, is also addressed in [2].

8.6.4 Membrane Systems and Ambient Calculus

Let us now briefly consider the connection with ambient calculus, introduced in [38] as an abstract model of the basic features of distribution and mobility. The central notion is that of a *mobile ambient*, which, in short, is a named, bounded place where a computation is performed. An ambient can have a collection of subambients of the same type. The ambients can move inside or outside other ambients, can open (dissolve) other ambients, can be opened by other ambients. A model of large networks, such as the Internet, is obtained in this way, with a formal framework for addressing important issues such as mobility and security.

The similarity between many notions of ambient calculus and of membrane computing is apparent, and they were examined in some detail in [211]. First, it is shown that a P system with symbol–objects, using the dissolving operation, can be expressed in ambient calculus. Then, one introduces the notion of a prioritized choice operator into ambient calculus, making possible also the simulation of P systems with a priority among rules. In the other way round, one brings the notion of secure communication from ambient calculus to the P systems area. This notion is implemented in membrane systems by creating new membranes, which transport objects to specified addresses, an idea also considered, with a biological motivation, in [23], and then investigated in [25] and [26] – we will return to this latter idea in Section 9.2.

For instance, given a P system Π, we can construct an ambient Λ to simulate Π in ambient calculus as follows. The ambient Λ has basically the same structure as Π, the same nested sequence of membrane–ambients, with the same names, but inside Λ, we create some other ambients to model the objects evolving in Π. They are empty ambients named as the objects that they stand for, and they are manipulated during the calculus, similarly as in Π. Obviously, we can arrange to have in the beginning of the simulation an initialization phase, in which each ambient creates its local data, according to the initial content of the corresponding region of Π.

Consider a region i of Π and an arbitrary rule r of i,

$$r : x \to (x_1 \dots x_m, here)(y_1 \dots y_n, out)(z_1, in_{k_1}) \dots (z_p, in_{k_p})\delta,$$

with $m \geq 0$, $n \geq 0$, $p \geq 0$, and possibly without the final δ. For this rule, we consider the following expression in ambient calculus:

$$E_r = (x, k_1, \dots, k_p \Rightarrow P_r),$$

saying that in the presence of ambients $x, k_1, \dots, k_p$ in the local ambient, the process P_r will be allowed to run (we will call it active). These are precisely the conditions in which the rule r is applicable, ignoring for the moment the priority relations among rules. The process P_r which simulates the rule r in the ambient i of Λ is the following:

$$\begin{aligned} P_r = !(\ open\ x.\ (\ (\ x_1[\ ] \mid \dots \mid x_m[\ ]) \mid \\ (\ y_1[\ ].mv\ out_i\ y_1 \mid \dots \mid y_n[\ ].mv\ out_i\ y_n\ ) \mid \\ (\ z_1[\ ].mv\ in_{k_1}\ z_1 \mid \dots \mid z_p[\ ].mv\ in_{k_p}\ z_p\ ) \mid \\ (\ acid)\ \ )\ ), \end{aligned}$$

where the process *acid* is in P_r only if r contains the symbol δ. The effect of P_r is that for each ambient named x, the following procedure is launched: the ambient x is open and then, in parallel, it sends the new ambients $y_1, \dots, y_n$ outside the ambient i, and the new ambients $z_1, \dots, z_p$ to their destinations $k_1, \dots, k_p$, according to the rule r of Π. Finally, it opens the ambient i if r is a dissolving rule. Obviously, there may be several processes $P_{r_1}, \dots, P_{r_j}$ attempting to open the same ambient x. In this case, the operator *open* succeeds for only one process, chosen randomly among $P_{r_1}, \dots, P_{r_j}$, and the other processes are cancelled. This is in accordance with the principles of membrane systems: if an object x can evolve according to several rules, then the system will choose non-deterministically the rule to be applied to x. Hence, the result of the process P_r is precisely the same as the effect of the rule r on Π.

Here is also the way in which in [211] priorities are introduced in ambient calculus by means of a *prioritized choice operator* $\oplus$, similar to the non-deterministic choice operator $+$. The result of an operation

$$m_1 \Rightarrow P_1 \oplus m_2 \Rightarrow P_2 \oplus \dots \oplus m_k \Rightarrow P_k$$

is that the process to be executed will be P_l, where l is minimal, $1 \leq l \leq k$, such that there currently exists an ambient named m_l in the ambient i of Λ. In other words, the process P_l will be executed with a higher priority than the processes $P_{l+1}, \dots, P_k$, provided that there is an ambient m_l in i.

Assume now that the priorities among the rules of the membrane i are $r_{j,1} > r_{j,2} > \dots > r_{j,l_j}$, for some $1 \leq j \leq k$ and $l_1, \dots, l_k \geq 1$. Then, the process running in the ambient i is constructed as follows:

$$
\begin{aligned}
P_i = (rec\ T)\ (\ & (\nu r) r[out\ i.in\ i.open\ r]\ . \\
& (\ E_{r_{11}} \oplus E_{r_{12}} \oplus \ldots \oplus E_{r_{1l_1}} \mid \ldots \mid \\
& \quad E_{r_{k1}} \oplus E_{r_{k2}} \oplus \ldots \oplus E_{r_{kl_k}}\ )\ ). \\
& (\ E_{r_{11}} \oplus \ldots \oplus E_{r_{1l_1}} \mid \ldots \mid E_{r_{k1}} \oplus \ldots \oplus E_{r_{kl_k}}\ )\ ).
\end{aligned}
$$

The first part of the recursion in P_i ensures that the process will run only as long as the ambient i is not open (in P systems the rules are lost after dissolving a membrane). Then, in parallel, for each priority chain, the active process with the highest priority is chosen. The procedure is then iterated in parallel in each ambient of the system, thus fully simulating the computation of Π.

As we have mentioned above, we will discuss more details later about P systems with gemmation (the possibility of sending membranes to a precise address, thus securing the contents against the rules acting in the compartments through which the membrane passes), which is a type of systems similar to those considered in [211] as a way to ensure secure communication in P systems in a way imported from ambient calculus.

We now pass to another very important (theoretical) computer science topic: constructing *universal computing devices.*

8.6.5 A Direct Construction of a Universal System

From the proofs of all the universality results given in the first chapters of the book one can obtain universal P systems of various types, that is systems Π_u with all components fixed, with the exception of the contents of one specified membrane, and which are able to simulate any particular P system Π if a code of Π is introduced as a multiset (or a set of initial strings, in the case of string–object systems) into one specified membrane of Π_u. This is done in an indirect way: from a universal Turing machine or a universal Chomsky grammar we have first to pass to a universal matrix grammar with appearance checking, and then to follow a proof of a universality theorem. For sure, even when starting from small universal Turing machines (such as those from [216], for instance, with the warning that they are language *recognizing* machines, not language *generating* machines, hence some preliminary modifications are necessary), the universal P system Π_u obtained will be huge in terms of the number of symbols and rules used. From this point of view, a direct way towards a universal system looks more promising, and such an attempt has been carried out in [70], for membrane systems with string–objects processed by means of splicing rules.

Specifically, a system Π_u, of degree 3, with the membrane structure of the form $[_1[_2\ ]_2[_3\ ]_3]_1$, is constructed, which is universal for all systems corresponding to the family $LOP_*(spl, in)$: if a set of strings $code(\Pi)$, related to any given system Π, is introduced in the system Π_u, then $L(\Pi_u) = L(\Pi)$.

Actually, a codification–decodification phase is necessary here, in order to deal with languages over arbitrary alphabets (and with systems generating

such languages), in the bounded framework provided by the components of the universal system. Because this is an important and non-trivial part of the construction, we recall some details from [70].

The arbitrary alphabets of an arbitrary P system Π are encoded into the alphabet $W = \{0, 1, 2\}$ as follows. Assume that the input system has $V = \{v_1, v_2, \ldots, v_p\}, T = \{v_1, v_2, \ldots, v_q\}$ with $q \leq p$. Then the input coding $f : V^* \to W^*$ is the morphism defined as

$$f(v_j) = \begin{cases} 20^j, & \text{for } j \leq q, \\ 10^j, & \text{for } q+1 \leq j \leq p. \end{cases}$$

The universal system will be $\Pi_u = (V_u, T_u, \mu_u, M_{1_u}, M_{2_u}, M_{3_u}, R_{1_u}, R_{2_u}, R_{3_u})$, where $\mu_u = [_1[_2\ ]_2[_3\ ]_3]_1$, relative to the coding functions

$$\begin{aligned} C_M &: V^* \longrightarrow V_u^*, \\ C_R &: V^* \natural V^* \$ V^* \natural V^* \times TAR \times TAR \longrightarrow V_u^*, \\ D &: T_u^* \longrightarrow T^*, \end{aligned}$$

where $TAR = \{0, 1, \ldots, m\}$. For an arbitrary splicing P system $\Pi = (V, T, \mu, M_1, \ldots, M_m, R_1, \ldots, R_m)$, where μ has m membranes, the function C_M transforms strings in $M_1, \ldots, M_m$ into strings in M_{1_u}, C_R transforms evolution rules of $R_1, \ldots, R_m$ into strings in M_{1_u}, and D transforms final strings of the universal system into final strings of the simulated one.

Formally, the fact that Π_u is universal is expressed as follows: for each splicing P system Π, $D(L(\Pi_u(\Pi))) = L(\Pi)$, where

$$\begin{aligned} \Pi_u(\Pi) = (&V_u, T_u, \mu_u, M_{1_u} \cup C_M(M_1 \cup \cdots \cup M_m) \cup C_R(R_1 \cup \cdots \cup R_m), \\ &M_{2_u}, M_{3_u}, R_{1_u}, R_{2_u}, R_{3_u}). \end{aligned}$$

The functions C_M, C_R, and D are constructed as follows. Let us consider an alphabet V so that $V = F \cup B, F \cap B = \emptyset$, where symbols in B are of the type h_l, t_d (for *head* and *tail* respectively) with several subscripts l and d. A string $q \in V^*$ is a *basic string* if $q = h_l e t_d$, where $e \in F^*, h_l, t_d \in B$. Moreover, q is said to be in *state* l, d if $l \neq d$, and in state l otherwise; e is called the *information* of q.

To *rotate* a basic string means to move symbols between h and t (with whatever subscript) from left to right or from right to left. More generally, by *rotation* we also mean the circular permutation of substrings present in the information of a basic string. In this context special symbols are used to keep track of the beginning of the rotated substring. One says that a splicing P system can rotate a string if, using its axioms, it can create basic strings which have the the same state but the information rotated (clockwise or anti-clockwise) with respect to other basic strings present in the system.

With *input-axioms* we refer to the elements of $C_M(M_1 \cup \cdots \cup M_m)$ and to their information; with *input-rules* we refer to the elements of $C_R(R_1 \cup \cdots \cup R_m)$ and to their information.

Input-axioms are $C_M(x) = h_1 o^i s_2 o^i b f(x) s_1 t_1$, where $x \in M_i$, $1 \leq i \leq m$.

The information of an input-axiom is composed of two parts: $o^i s_2 o^i$, $1 \leq i \leq m$, indicates (twice for technical reasons) the language M_i of which the string x is a member; the other part, $bf(x)s_1$, represents the axiom of the system to simulate. The symbol b indicates the beginning of the coding; s_1 and s_2 are used as separators.

Let us define the set $\bar{W} = \{\bar{a} \mid a \in W\}$ and the morphism $bar : (W \cup \{\natural\})^* \to (\bar{W} \cup \{\bar{\natural}\})^*$ by $bar\,(a) = \bar{a}$.

Let $r = (u_1 \natural u_2 \$ u_3 \natural u_4; tar_1, tar_2)$ be an evolution rule present in membrane j. Then a string of the following form will be added to the universal system:

$$C_R(r) = h_3 s_1 bar(mi(f(u_1)\natural f(u_2)))o^{tar_1} s_2 o^{tar_1} \bar{\$} o^{tar_2}$$
$$s_2 o^{tar_2} bar(mi(f(u_3)\natural f(u_4)))o^j t_3.$$

The decoding function D is defined as

$$D(h_1 vbws_1 t_1) = f^{-1}(wv),$$

where f is the morphism for the encoding of the alphabets used above.

As the language generated by the universal system consists of strings in state 1, with a marker b indicating the start of the rotated string, the decoding is easily understood. (The decoding is only defined for strings in state 1.) Apart from undoing the coding into the alphabet W, D also recovers the output by performing a cyclic permutation.

The universal system constructed in [70] is based on a series of modules, which make possible the simulation in Π_u of the work of Π; this means that one has to provide such modules both for "programming" the mode of working of a P system (choosing non-deterministically the rules to apply and the strings to which they are applied, applying the rules, moving the results of the splicing operations according to the target indications associated with the splicing rules, sending out strings), and for the effective use of the rules, for splicing. We do not recall here other details of this complex construction, but only mention that the universal splicing P system obtained in this way contains 594 rules. This does not look too much, but, of course, is far from any hope of being handled in a laboratory.

8.6.6 Further Research Topics

In the previous chapters of the book (as well as in the present chapter) we have seen that many problems are still open, and many research topics remain to be addressed. At first sight, it is rather easy to formulate such research topics, because membrane computing is a young area, hence many things are still waiting to be clarified. However, the formulation of a research topic (of interest, if possible, of *lasting* interest) is in itself a matter of research. That is

why we merely suggest here some directions of research, without being very precise in the formulation.

First, two issues of a technical type.

So far, the membrane systems were always considered in the *generative sense* (one starts from an initial configuration and one collects all outputs, numbers or strings, which the system produces), and as *decidability devices* (one encodes a problem in the initial configuration and one looks for a specified object, *yes*, in order to have an answer to the problem). The generative approach corresponds to grammars, the decidability is automata-like, but not completely. What about using P systems as *accepting* devices? (Problem Q32.) The idea is simple: consider a system Π (membranes, rules, objects), and introduce in its initial configuration a further multiset, M, of objects (we now consider the symbol–object case). We say that M is accepted if, starting from this augmented configuration, the system halts (or halts and produces a special object *yes*, as in the decidability case). Then, the set $N(\Pi)$ consists of all numbers describing the cardinality of all multisets M "recognized" in this way by Π.

This is a classic problem in formal language theory. For grammars the question is not of interest (by reversing the rules of a grammar G, we recognize, that is we reduce to a special symbol S, exactly the strings generated by G), but this is not the case for grammar systems (see, for example, [28]), where interesting differences exist between the generative and the accepting way of using a cooperating distributed grammar system. Membrane systems are distributed computing devices, hence a similarly non-trivial situation is expected.

A second technical – but general – research topic, somewhat suggested already in the book, is to systematically investigate the case of systems with a small number of membranes (Problem Q33). This means systems that are not necessarily universal, hence their properties are not known, as direct consequences of the equality with NRE or RE. The closure properties of such families have already been mentioned, but other classic questions can be raised: comparison with known families from Chomsky and Lindenmayer hierarchies, decidability properties, descriptional complexity (do the numbers of symbols or of rules, the lenght of rules, and other parameters of this type introduce infinite hierarchies in the family of number sets/languages generated by systems by a given type, with a given number of membranes?). Suggestions from the well-developed area of descriptional complexity for Chomsky grammars and languages (see [81]) can be useful.

Let us now consider now a problem with a "practical" motivation, but with an important technical content. When we introduced the systems with symport/antiport we mentioned that they observe the conservation law: nothing is created and nothing is destroyed, all objects are moved only from one compartment to another compartment. This property can be easily extended to all types of systems dealing with symbol–objects provided that coopera-

tive rules are allowed, by adding dummy objects to the rules, just to balance the number of objects on the two sides of rules (the dummy objects never evolve and are not considered when defining the output). We do not enter into further details here.

Not so easy to handle seems to be another important biochemical suggestion, the *reversibility* (Problem Q34). Many reactions are reversible, and can go in either direction (possibly depending on a change of reaction conditions, for instance temperature – hence energy availability). In dynamic systems, reversibility is somewhat opposed to non-determinism: we have to uniquely find the previous configuration of the system starting from the present configuration. Strictly chemically speaking, reversibility means reversing the direction of an equation. What about reversible P systems in the dynamic systems sense, what about "local reversible P systems", with the rules allowed to be used in both directions (for each $u \to v$ we have also to add $v \to u$ to the same region)? (By the way, it has been known for many years, [21], that reversible Turing machines are computationally universal. Can reversibility be brought "for free" from the Turing machines area to the P systems area?)

A further research topic, both mathematically and practically interesting, is related to the fact that membrane computing comes from biochemistry, where most processes are non-deterministic, and the results are only approximately/probabilistically known/true. Up to now, only "crisp" mathematics has been used in the P systems area. What about "approximate" mathematical approaches, using probabilities, fuzzy sets, or rough set theory? (We refer to [204] for the rough set theory; the basic idea is to approximate a set from the interior and from the exterior, by converging unions of equivalence classes.) What about "approximate" computing, whatever this can mean? (Problem Q35.)

A promising research topic (Problem Q36), again with a practical flavour (see also the discussion from Section 9.5), is to "translate" the idea of an eco-grammar system, as introduced in [45], in terms of membrane systems. In short, an eco-grammar system consists of a set of agents, evolving in a common environment. The agents have their own rules for internal evolution, as well as rules according to which they can act on the environment or on other agents. In turn, the environment has its own evolution rules, and, depending on its state, it determines the state of the agents (the set of rules active at a given moment). In [45], both the agents and the environment are described by strings, hence the evolution rules are rewriting rules (used either in the sequential or in the parallel manner). In the framework of membrane computing, the agents can be membranes, placed into a larger membrane which represents the environment. Bearing in mind an eco-system, both the agents and the environment should be finite, but it could be of interest to have one more region around the environment (the "world", the "universe"), infinite, providing raw materials to the environment, but not active otherwise, for instance not in the range of communication of agents. The agents

can exchange objects with the environment (for instance, by means of symport/antiport rules), the environment can send objects into the universe, and can grab objects from the universe (an object which exits the environment is "lost"). For the agents it is probably better to have rules as simple as possible (following somehow the style of *colonies* – see [105]), for instance only renaming rules, of the form $a \to b$, or symport/antiport rules of a minimal weight; it is expected that a careful choice of innocent-looking ingredients will produce an interesting emergent behaviour, as often met in the grammar systems (colonies included) area.

We end this short list with another very general question (Problem Q37), actually a list in itself of "exotic" problems. What about introducing some quantum computing ideas in membrane computing (or conversely)? Imagine, for instance, that the objects b and c produced by a rule $a \to bc$ are *entangled*, each one "senses" the other, irrespective of where they are placed after several steps of a computation. Can this be used in our framework – in particular, can it be used for speeding up the computations? The entanglement is only one quantum notion; many others can be considered: quantum object, superposition, teleportation – see [82].

In the same "exotic" area, what about the reproduction of a membrane system? The problem has a long history for cellular automata (S. Ulam, J. von Neumann, etc.). By the division of a membrane we have directly introduced reproduction at the level of inner membranes, but this is not allowed for the skin membrane. Is there any non-trivial way to approach this issue?

What about also trying to go "beyond Turing"? We have mentioned in the Introduction that in the framework used in this book this is not possible: the systems are always discrete, finite, hence recursively enumerable. However, one can introduce ingredients which are not Turing coverable, such as infinite computations, continuous elements, accelerations (consider a membrane which "learns", so that it works normally the first time unit, but the job for the next time unit is done in half the time – while the other membranes work normally – the job for the third time unit is done in half of the time necessary for the previous step, and so on, hence using $1/2^n$ time units for the nth step of the computation). An easy way to "compute" non-recursively enumerable languages is to work on the complement, in the sense of the so-called "computing by carving" from [184], which is an abstraction of the filtering procedures from DNA computing; can the "computing by carving" also be used in membrane computing?

There are plenty of possibilities. The membrane computing framework is very general, hence very permissive from this point of view.

9. (Attempts to Get) Back to Reality

In the Introduction, we stressed the fact that membrane computing is inspired by the biology of the cell, but it is a theoretical computer science enterprise, aiming to provide new computing paradigms abstracted from the cell structure and functioning, but not (yet) trying to model the cell in such a way as to return relevant conclusions to the biologist. Anyway, this was the style of the developments in this area up to now and this was the style of this book. On the other hand, the mathematical developments in membrane computing already seem to make possible attempts to capture more real-life features of living cells, so that, to try to build *a model of the cell*, starting from *the computing models inspired by the cell.* The usefulness of such a model can be proved by (making possible) computer simulations of various aspects of a cell.

Without going here into such an ambitious project, but supporting it in a fragmented manner, the following sections describe some attempts to capture more specific cell features, to apply membrane systems, and to implement them (especially on the usual computer). The reader should treat these attempts mainly as an invitation to research areas which still await a serious effort – in general, by an interdisciplinary team.

9.1 Getting Closer to the Cell by Energy Accounting

Actually, what we add in this section to the membrane computing framework is only one further feature, fundamental for cell life, but which was completely ignored in the previous chapters of the book – and which was only recently considered in the membrane computing literature – namely the *energy.* Life means a continuous production and consumption of energy, from the small scale to the large scale, from the thermal agitation and ATP, to eco-systems transformations. Most of the trans-membrane transport of chemicals consumes energy, and most of the chemical reactions either consume or produce energy.

In an attempt to take some steps towards a formal approach of this important feature of living cells, we can relate the energy aspect to one of the interesting concepts developed in molecular biology in the last decades, namely that of a *conformon*, which, according to Green and Ji [80], is defined

as "the free energy associated with a localized conformational strain in biological macromolecules". Later, the concept was enlarged, in order to include genetic information: "Conformons are genetically determined local conformational strains of biological macromolecules, each endowed with specific biological functions including ligand binding, catalysis, free energy storage, and free energy transfer", [101, 102].

In a more abstract formulation, a conformon is an information–energy pair, hence it is described by a *name*, for instance a symbol from a given alphabet or a string over a given alphabet, and a *number*, accounting for the energy associated with the name–information. Anyway, this was the manner in which the idea of a conformon was captured and introduced it in the framework of a membrane system in [71]. The model considered in [71], in the form of a P system with the objects being conformons in the above sense and reacting by exchanging energy, was then investigated as a computing model (and shown to be computationally universal), not as a model of what happens in the cell, although several suggestions are immediate. Here we preliminarily explore these suggestions, mainly making them explicit for further investigation.

This attempt can also be seen as a possible step towards "superposing" the mathematically elaborate models from membrane computing onto a molecular model of the living cell called *Bhopalator* (because it was proposed during a meeting held in Bhopal, India, in 1983), see [101, 102]. As the originator of the Bhopalator, puts it in [102], "There are three major concepts that are novel to the Bhopalator: 1) *Conformons*, defined as sequence-specific conformational strains of biopolymers that carry free energy and genetic information to drive all molecular processes in the cell, 2) *intracellular dissipative structures* (IDS's), i.e., the gradients of chemical concentrations and mechanical stresses in the cell that are the final forms of gene expression, and 3) *cell language*, the language that cells employ to communicate and compute, to reason, and to argue, using molecules as signs."

We remain here at the level of the first concept, that of conformons, and refer to the above-mentioned papers for the intriguing speculations about the cell language.

Thus, let us consider a finite alphabet V, of *names* for conformons (the biologists have classified them in ten classes, each class containing a large number of members, from 10^{30} to 10^{60}, see [102]), and a natural number E_a associating to each conformon the *maximal energy* it can contain. We also accept E_a, for $a \in V$, to be zero. The pairs (a, E_a) are called conformons. During its "life" in the cell, a conformon will transfer energy to other conformons, or will simply spend it for trans-membrane transport or for other purposes, hence the current state of a conformon will be of the form (a, e) with $0 \leq e \leq E_a$. We will also call conformon any pair (a, e) with $0 \leq e \leq E_a$, although this is just one possible state of the same conformon. Otherwise formulated, we identify a conformon with its name.

Two conformons can react, by exchanging energy, according to given *rules*. Such a rule should specify the names of the reacting conformons and the energy they exchange, hence we write then in the form

$$r : a \rightarrow_s b,$$

where $a, b \in V$ and $0 \leq s \leq E_a$. The meaning is that the conformon a will pass s "quanta of energy" to conformon b, that is we have the *reaction*

$$(a, e_1) + (b, e_2) \Rightarrow_r (a, e_1 - s) + (b, e_2 + s).$$

Such a reaction is possible only if $e_1 - s \geq 0$ (after reaction the conformon a cannot get negative energy, such a thing does not exist), and $e_2 + s \leq E_b$ (the conformon b will not get "too much" energy, more than it is allowed to carry).

In this way, the energy is passed from one conformon to another one, hence the total balance remains the same. We can also consider "ATP-like conformons", which we write in the form $\langle a, e \rangle$, with the same meaning as for a conformon (a, e), but used in reactions in the following way:

$$\langle a, e_1 \rangle + (b, e_2) \Rightarrow_r \langle a, e_1 \rangle + (b, e_2 + s),$$

provided that the rule $r : a \rightarrow_s b$ exists, and that $e_2 + s \leq E_b$. That is, the ATP conformon a has enriched the conformon b with s quanta of energy, without losing this energy (another possibility is to lose only part of this energy).

By such reactions, the total energy available in a cell or in a region of a cell increases.

We can provide ways to also spend energy, for instance for communication. Let us assume that we are given a membrane structure μ where our conformons evolve, with m membranes labelled with the natural numbers from 1 to m. With each membrane i we associate two numbers: l_i and p_i. The first number is the *minimal energy a conformon should have in order to pass through membrane i*, the second number is the *energy spent by any conformon when passing through membrane i*. That is, if a conformon (a, e) has enough energy, in the sense that $e \geq l_i$, then it can pass through membrane i, from the compartment where it is placed, to the compartment behind membrane i (if the conformon is inside, then it will exit; if it is outside the region bounded by membrane i, then it will enter into this region). However, during this operation, the conformon has "to pay" p_i quanta of energy, hence after passing through membrane i it will have the state $(a, e - p_i)$. Of course, we must have $p_i \leq e$, otherwise the passage is not possible. Observe that we have imposed no relation among p_i and l_i, for instance to have $p_i \leq l_i$. It is possible that a membrane has a low threshold l_i, but asks for a bigger number of energy units when a conformon passes through it. This energy, spent during communications, is lost, we do not count it (for instance for membranes), hence we cannot return it to conformons.

Of course, the previous assumptions can be modified to better fit the reality, or further assumptions can be considered. Here are only a few possibilities: (1) a conformon which gets the state of null energy, that is the state $(a, 0)$, can be removed from the system as being "dead", chemically inactive forever; (2) we can also keep a balance of energy from each region, and associate a "blowing threshold" with each membrane; if at any moment of the evolution of the system, in the region of a given membrane we get conformons which have together more energy than this threshold, then the membrane is broken, dissolved; (3) an interaction with the environment of the cell can be considered, bringing energy "for free", and/or dissipating energy continuously; the interface with the environment is ensured by the skin membrane/region, hence the way of passing this "free energy" to lower compartments can be of interest; (4) during the reactions we have allowed only the exchange of energy, but it is equally possible, at least theoretically, to also consider rules which change at the same time the names of the conformons; a nice variant is to change the names of conformons according to their energy, also consuming energy when the name is changed; (5) the permeability of membranes, that is the thresholds l_i, can be modified during the passage of molecules through them, or, perhaps more appropriate to reality, the thresholds should be assigned to membrane–object pairs: certain objects can pass through a membrane when having a lower energy than others, perhaps "paying" less energy than others; (6) instead of associating only one number with a conformon, a more realistic case would be to associate several numbers, corresponding to several parameters (the energy, the mass, the momentum, etc.) which describe a conformon, and then also the reaction rules and the rules for passing conformons through membranes have to be changed accordingly; and so on, and so forth.

In [71] the rules governing the reactions of conformons are of the form $r : (a, e_1; b, e_2; e_3)$, with the meaning that the conformon with the name a can pass e_3 energy units to the conformon with the name b, provided that before the reaction a has at least e_1 energy units and after the reaction b will not have more than e_3 energy units. Also, certain *predicates* are associated with the membranes, in the form of conditions $\leq n, = n, \geq n$: a conformon can pass through a membrane only if its energy satisfies the associated condition (its energy is smaller than, equal to, or larger than n, respectively). Actually, there are two more important features of the systems considered in [71]: (1) the membrane structure is, in fact, a network of cell-like P systems, with communication allowed also among different cells, and (2) the computations are considered terminated at the moment when a special object (conformon, here) arrives in a specified membrane. This last idea is very interesting from a theoretical point of view, as it provides an alternative to the way of defining successful computations as halting computations. The life of a cell is – if not infinite – arbitrarily long in comparison with the duration of a computation

(a particular process which takes place in the cell), hence it is natural to use such "flag objects" in order to signal the end of a computation.

In this framework, a characterization of recursively enumerable sets of natural numbers is obtained, which is again a result of a "classic" type. However, by implementing such a system on a computer (such implementations exist for other classes of P systems, see Section 9.4), we can address completely different questions related to the "life" of the system, to its evolution in time. The range of such questions is rather large: Is any membrane dissolved during the life of a given cell? Does a region become overpopulated/empty? Are objects leaving the system? Are objects removed because of null energy? What is the balance of two given objects, in given membranes? How many conformons are reacting at a given moment? What about deadlocks, moments where no reaction is possible? What about communication deadlocks, when no communication takes place (but reactions are produced)? How long can be a no-communication interval, bounded by two steps when communication takes place? What is the role of non-determinism? If some probabilities can be introduced into the model (for instance, conformons with a higher energy are more likely to react than conformons with a lower energy), then these questions – and others, too, which can be considered – can become more interesting, more "life-like".

We stop these speculations here (and we leave the study of conformon-based P systems as a *research topic* – Problem Q38), and we pass to a short presentation of another way of considering energy in the framework of membrane systems aiming to simulate real cells. Specifically, we consider the case where the reactions which take place in a region consume or produce energy. This idea can be easily captured by associating a positive or a negative amount of energy with the usual evolution rules from symbol–object or string–object P systems.

For instance, we can consider rules of the form

$$u \to v\langle e\rangle,$$

where u, v are multisets of objects, and e is a positive or negative integer. When such a rule is used for processing a multiset of objects from a region i of a P system, the amount of energy specified by e is handled; if e is a negative number, then the energy is consumed (we have an endogenous reaction), if e is a positive number, then the energy is released during this reaction (it is an exogenous one), and when $e = 0$ no quantum of energy is involved in the reaction. Of course, in order to apply a rule $u \to v\langle e\rangle$ with a negative value for e we must have the necessary energy available in the membrane where the rule is applied (sufficient ATP should be available). More precisely, we have to balance the energy for all rules used in a given membrane at a given step, summing up the energy associated with all rules which we want to apply, as well as the energy inherited from the previous steps of the system evolution, and only if the balance is positive can we use the considered rules. This can

provide a powerful way of controlling the non-determinism of choosing the rules and the objects which can evolve at a given time.

Note the important differences from the case of conformons, as considered above: the objects are only identified by their names, they do not have associated energy, this time the energy is counted at the level of each region, and fluctuates according to the rules which are applied in the region.

As in the case of conformons, we can consider thresholds for dissolving membranes: when a region contains "too much" energy, as the result of the evolution of its objects, it is dissolved, the objects are passed to the upper membrane, while its energy is consumed by this operation.

Associating energy with rules can also provide a way to control their use, stating that a rule which requests less energy (implicitly, produces more energy) can be applied in an easier way, hence it has priority over rules which are energy consuming. However, this would promote the "over-heating" rules, which will end in a high rate of dissolution of membranes. Also, as in the case of conformons, we can involve or not involve the environment in the energy balance, we can add probabilities, and so on and so forth.

A nice possibility (also relevant to the biochemical reality) is to combine the two ways of handling energy, and to consider conformons which carry energy and exchange energy through interactions with other conformons, to consider also reaction rules which change the names of conformons, passing their energy to the new conformons, and consuming or producing energy at the level of the regions of the system, perhaps also making possible the transfer of energy from the level of conformons to the level of the region, and back. The game can be rather sophisticated, which implies both the danger to become a purely mathematical game, and the possibility to become a relevant way to approach the cell – but this latter possibility should be checked in collaboration with a molecular biologist, and this is beyond the scope of the present book.

Note. P systems with energy accounting, both in the case of symbol–objects and of string–objects, were considered in [198], but mainly as computing models; characterizations of NRE are obtained for symbol–objects, for various combinations of features (catalysts, priorities, two membranes, no bound on the energy of each rule, or no catalyst, priority, no bound on the number of membranes, with $-1, 0, +1$ quanta of energy associated with each rule, respectively; the first result also holds without energy accounting, Theorem 3.4.3, but with a much more complex proof, while the second result is obtained without using catalysts, a rather rare case of universality), as well as for string–objects (no bound on the number of membranes, but at most two quanta of energy – positive or negative – associated with each rule).

9.2 Getting Closer to the Cell by Gemmation

In Subsection 8.6.4 we mentioned the class of P systems considered in [211], where multisets of objects are communicated from one region to another one encapsulated in a membrane which has only this role, of transporting objects in a protected manner, not to be used by the evolution rules of the regions through which one passes. This corresponds very much to the communication based on vesicles, as described in Subsection 2.1.2. Especially, Golgi apparatus makes continuous use of vesicle communication, when sending "mature" proteins to various destinations by first covering them in a vesicle.

A possible way to model this phenomenon is the following one (we recall some ideas from [23, 25, 26, 130]).

Consider a P system with string–objects, with the alphabet V, and a membrane structure μ with m membranes labelled with the numbers from 0 to m. The membrane structures which we allow have two levels only (the skin membrane contains a number of elementary membranes inside), and, because the skin membrane has a special position also in the gemmation of mobile membranes, we label it with 0, while the elementary membranes are labelled with $1, 2, \ldots, m$. For each $i \in \{0, 1, 2, \ldots, m\}$ we consider a special symbol, $@_i$, not in V. It will be used as an address for the destination of the vesicles which can be created inside the system.

One uses two types of rules: the usual rules for processing string–objects (rewriting, possibly also with replication, splitting, etc. – in [25, 26] the operation used are point mutation, replication, and splitting), and also so-called *pre-dynamical* rules.

A pre-dynamical rule is a classic rule which introduces one or several symbols $@_i$. For instance, in a region i of the system, other than the skin region, we can have the rewriting and replicating rule

$$a \to (u_1@_j)||(u_2, tar).$$

When such a rule is applied to a string w_1aw_2 we get two strings, $w_1u@_jw_2$ and $w_1u_2v_2$. The second string will immediately go to the skin region if $tar = out$ and will remain in the same region i if $tar = here$, as usual in P systems. The first string will produce a new membrane around it, which will be moved to the membrane with label j, as indicated by the symbol $@_j$.

The gemmation of mobile membranes is possible only in the inner membranes, and not in the skin region. However, the address of a vesicle can be the skin region, a case which is interpreted as sending the contents of the vesicle out of the system.

Let us describe in some detail the process of creating and moving membranes.

First, we assume that the rules which do not introduce any symbol $@_j$ have a higher priority – this is meant to model the fact that only "mature objects" can be transported by means of vesicles. However, if a symbol $@_j$ has been introduced, at that moment we start the communication.

We represent the gemmation of a mobile membrane in the following way:

$$\begin{aligned} &[_0 \cdots [_i \cdots, w_1 @_j w_2, \cdots]_i \cdots]_0 \\ \Rightarrow\ &[_0 \cdots [_i \cdots]_i (_{i,j}\ w_1 w_2\ )_{i,j} \cdots]_0, \end{aligned}$$

for $i \in \{1, \ldots, m\}, j \in \{0, 1, \ldots, m\}, j \neq i$, and $w_1, w_2 \in V^*$.

That is, the symbol $@_j$ is removed from the string $w_1 @_j w_2$, and a mobile membrane is created, outside membrane i; we have represented it by parentheses, with two subscripts. The first subscript is the label of the originating membrane, the second one is the label of the target membrane, that indicated by the symbol $@_j$.

If at the same time, in the same region i, we have several strings containing the same symbol $@_j$ (this can happen as the result of the application of several rules, or of only one rule which replicates the string) then all of them will be encapsulated in the same new membrane. On the other hand, if inside membrane i there are several strings containing symbols $@_j$ with different subscripts, then as many mobile membranes are created as there are many different destinations.

After gemmating a vesicle, it will transport its contents to the destination, and will fuse with the indicated membrane. This last operation can be formally written as follows:

$$[_0 \cdots (_{i,j} w)_{i,j} [_j \cdots]_j \cdots]_0 \Rightarrow [_0 \cdots [_j \cdots, w, \cdots]_j \cdots]_0,$$

for some $i \in \{1, \ldots, m\}, j \in \{1, \ldots, m\}, j \neq i, w \in V^*$.

In this way, the content of the mobile membrane becomes a part of the content of the target membrane.

If the target was the skin membrane, that is $j = 0$, then the mobile membrane fuses with the skin membrane and the objects exit the system. In this way we can simulate the biological process of exocytosis (and we can send strings outside the system, which is useful when dealing with computations). This operation can be described as follows:

$$[_0 \cdots (_{i,0} w)_{i,0} \cdots]_0 \Rightarrow [_0 \cdots]_0 w.$$

The movement of the vesicle from the originating membrane i to the target membrane j is supposed to take place immediately, although this is an unrealistic assumption. A simple way to introduce some reasonable duration for the gemmation–fusing process is to say that it takes a time proportional to the "weight" of the transported objects. In turn, the weight can be given by the length of the strings, or by using a valuation-like mapping.

A much more complicated situation appears when we want to consider gemmation-like operations for systems having a membrane structure of an arbitrary form (this case was considered in [130]), because a vesicle can be created in a membrane i and directed towards a membrane j which can be placed on a different level of the membrane structure, possibly with several

membranes lying in between the two ends of the path. In such a framework, the duration of the process can be related to the number of membranes which have to be crossed (and perhaps also to the weight of the transported objects).

Especially in this general case the role of the transporting vesicle is clear: protection for the transported objects against the evolution rules from each region which is passed through – note that the moving vesicles have no evolution rule associated with them, hence the objects remain unchanged during the communication. This aspect is essential in the case of mobile ambients from mobile calculus, as emphasized in [211].

The operation of creating mobile membranes is very useful as a "programming technique": P systems with string–objects, processed by context-free rules, also using gemmation (with priority of the usual rewriting rules over the pre-dynamical rules, but not with priority among rules from each of the two categories), characterize the family of recursively enumerable languages. Seven membranes are used in the extended case, and eight in the non-extended case – which looks a little bit too many and probably can be improved, but the lack of powerful features, such as priorities and/or actions δ, τ, may be a reason that in this case the hierarchy induced by the number of membranes does not collapse at a very low level. This might be an "explanation" why the cell is so complex: it uses gemmation, but not powerful computer-science-inspired features.... Anyway, we also believe that gemmation – and related topics, in connection with ambient calculus as well – deserves further investigation (Problem Q39).

9.3 Getting Closer to the Cell: Bilayer Membranes

In Section 2.1 we omitted to discuss several particular details about biomembranes. Two of them, leading to the same formal modification in a membrane system, are briefly considered here, and they refer to the fact that what we call "membrane" is actually a double wall, with an internal compartment, which can be used as a region with specific properties in the framework of P systems.

First, each membrane is a bilayer of phospholipid molecules, as described in Section 2.1, hence in between the two layers we may consider an inner region (let us call it *inter-region*), of a small size in comparison with the region delimited by the membrane (we reserve the term *region* for the compartment enclosed by the membrane as a whole, hence, rigorously, by the inner layer of the membrane). The inter-region is strongly hydrophobic, because of the fatty acids which face this space from both sides. From P systems' point of view, if we take into account the inter-regions, we have to consider not only one further region for each membrane, but a more sophisticated manner of communicating objects among regions: we can send objects from the inner region to the inter-region or directly outside the membrane, from the inter-region to the inner or to the outer regions, and from the outer region to

the inter-region or directly to the internal region. The communication in two steps, with a stop in the inter-region, can be *unprotected*, or *protected*. In the first case, if there are rules in the inter-region which can be applied to the transported objects, then, because of the non-determinism of using the rules, it is possible to evolve these travelling objects in such a way that they can change their destination. In the protected transport, we may consider the objects encapsulated in vesicles which make the objects inactive until passing through the whole membrane (in the same way as we have proceeded in the previous section). Figure 9.1 illustrates this variety of possible transport across a double membrane.

A communication in two steps through a membrane is the only one which can be considered in the case of certain bacteria, which have a double bilayer membrane (so, counting all phospholipid layers we may consider that we have four membranes in total, with three inter-regions around the inner region of the double membrane). This is the case with the so-called *Gram-negative* bacteria, which have two membranes, an external one and an internal cell membrane; in contrast, the bacteria having only one cell membrane are called *Gram-positive*. An examples of the first type is *Escherichia coli*, while an example of Gram-positive bacteria is *Micrococcus species*. The space between the two membranes of Gram-negative bacteria is called the *periplasmic space* and contains proteins involved in the transport of molecules across membranes, as well as other organic or inorganic components. The two membranes have the same fluid–mosaic structure, but different compositions and functions.

There are differences between the external and the internal membranes in what concerns the transport of ions and molecules. For example, inorganic ions can pass through the external membrane, but their passage through the internal membrane is controlled by proteins embedded in it. Other molecules, larger than ions, such as certain antibiotics, cannot cross the external membrane, and this provides resistance to certain antibiotics to Gram-negative bacteria.

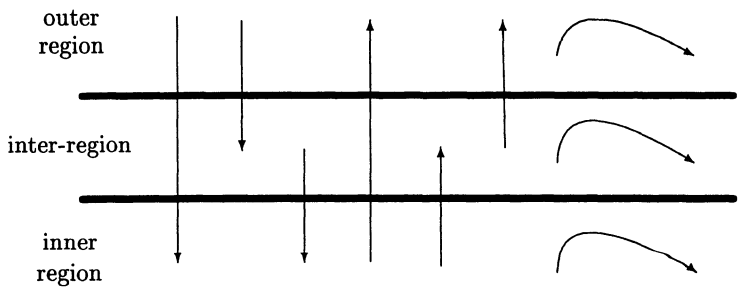


Fig. 9.1. Transport across a double membrane

The inter-region placed between the two layers of any membrane as well as the region delimited by the two membranes of Gram-negative bacteria can be used in terms of P systems (the case of the inter-region was explicitly considered in [169], for the case of symbol–objects). At first sight, from a formal point of view, this is not of much interest, as already systems with a small number of membranes are universal. For instance, for all results dealing with one membrane, considering the inter-region is useless (some small problems appear in the communication with the environment, but we can do it in two steps), while for results about systems with two membranes we can use the inter-region instead of one of these membranes, so we can obtain the result for systems with only one double-layer membrane.

However, there is an important detail here. If we want to be realistic, we have to avoid to have "too many" objects at the same time in the inter-region, as this space is considerably smaller than the internal region. Thus, we have to be careful with the role we assign to the inter-region. On the other hand, the inter-region is not an elementary region, hence it cannot be the output region – unless we allow this by the definition of P systems we consider (this is just a technical convention), or we ensure that the inner layer of the membrane is dissolved during the computation. This latter idea raises the question whether or not we are allowed to use such a half-membrane dissolution – an operation without any biological counterpart.

We do not examine from these points of view (especially for bounding the number of objects staying in the inter-region at any time) the universality results which use P systems with two membranes, in order to convert them to the case of considering the inter-region (this remains as a research topic – somewhat routine, unless we do not also impose restrictions on the rules which are allowed in the inter–region), but we consider only one example, of a "difficult case": that of systems with symport/antiport rules, as in Theorem 4.2.1, where it is proved that $NOP_2(sym_2, anti_2) = NRE$. The proof uses the two regions of the system in a way which is completely inadequate for a direct adaptation to the situation met in the inter-region case: the whole work is done in the skin region, where one continuously keeps all symbols of the sentential forms of the simulated matrix grammar, while the inner membrane is the output one, hence able to store in the end an arbitrarily large number of copies of the terminal object a. Thus, none of the two regions can be directly considered the inter-region of a membrane. Still, we can change the construction in the proof of Theorem 4.2.1 so that the skin membrane is used only as a "transit room", at each moment holding only a bounded number of objects. As we will immediately see, this number is rather low, namely six.

We start, as in the proof of Theorem 4.2.1, from a matrix grammar with appearance checking in the Z-binary normal form, $G = (N, \{a\}, S, M, F)$. Consider a new symbol, d, and replace each matrix of type 2, $m_i : (X \to Y, A \to x)$ with $|x| < 2$ by $m_i : (X \to Y, A \to xd^s)$, where $s \in \{1, 2\}$ such that $|xd^s| = 2$. We construct the following P system with symport/antiport rules

(of degree 1, but with the membrane considered double, with an internal layer denoted by $[_{1i}\ ]_{1i}$, and an external layer denoted by $[_{1e}\ ]_{1e}$):

$$
\begin{aligned}
\Pi &= (O, [_{1e}[_{1i}\ ]_{1i}]_{1e}, w_{1e}, w_{1i}, E, R_{1e}, R_{1i}, 1i),\\
O &= N_1 \cup N_2 \cup \{a, c, d, f, g, h, Z, \#\} \cup \{c_i, c_i', c_i'', d_i \mid 1 \le i \le n\},\\
w_{1e} &= \#\#,\\
w_{1i} &= cX_{init}A_{init},\\
E &= N_1 \cup N_2 \cup \{a, d, f, g, h, Z\} \cup \{c_i, c_i', c_i'', d_i \mid 1 \le i \le n\},\\
R_{1e} &= \{(cX, out; c_i, in),\ (c_iA, out; c_i'c_i'', in),\\
&\qquad (c_i', out; c\alpha_1, in),\ (c_i'', out; Y\alpha_2, in) \mid\\
&\qquad \text{for all } m_i : (X \to Y, A \to \alpha_1\alpha_2), 1 \le i \le k,\\
&\qquad \text{and } \alpha_1, \alpha_2 \in N_2 \cup \{a, d\}\}\\
&\quad \cup \{(d, out),\ (cZ, out)\}\\
&\quad \cup \{(cX, out; c_id_i, in),\ (d_i, out; Yh, in),\\
&\qquad (h, out; cg, in),\ (c_ig, out) \mid\\
&\qquad \text{for all } m_i : (X \to Y, A \to \#), k+1 \le i \le n\},\\
R_{1i} &= \{(cX, out),\ (c_i, in),\ (c_iA, out),\\
&\qquad (c\alpha_1, in),\ (Y\alpha_2, in),\ (c_i, out; \#, in) \mid\\
&\qquad \text{for all } m_i : (X \to Y, A \to \alpha_1\alpha_2), 1 \le i \le k,\\
&\qquad \text{and } \alpha_1, \alpha_2 \in N_2 \cup \{a, d\}\}\\
&\quad \cup \{(c, out; \#, in),\ (\#, out; \#, in),\ (d, out),\ (cZ, out)\}\\
&\quad \cup \{(cX, out),\ (c_id_i, in),\ (c_i\#, in),\\
&\qquad (c_iA, out; \#, in),\ (d_i, out),\ (Yh, in),\\
&\qquad (h, out),\ (cg, in),\ (c_ig, out) \mid\\
&\qquad \text{for all } m_i : (X \to Y, A \to \#), k+1 \le i \le n\}.
\end{aligned}
$$

The reader should compare this construction with that from the proof of Theorem 4.2.1. This time all the work is done by the inner region, which holds all symbols of the sentential forms of G, while the inter-region always holds two copies of the trap object $\#$, needed to control the correct simulation of derivations with respect to G, and at most other four objects. This maximum of four objects is reached in two situations: when simulating a matrix of the form $m_i : (X \to Y, A \to \alpha_1\alpha_2)$, one brings at the same time from the environment the couples $c\alpha_1$ and $Y\alpha_2$; if both α_1 and α_2 are equal to d, then, after introducing them into the central region, they leave membrane 2 at the same time with a couple cX, which starts the simulation of another matrix. However, all these objects immediately leave the skin region (in exchange for one or two objects from the environment).

We do not enter into details concerning the way the system Π simulates the derivations with respect to G; after the many similar proofs from the previous chapters, this should already be a routine task for the reader.

In Section 9.6 we will use a membrane system with two membranes, but also taking into account the inter-region of the internal membrane, as a basis for simulating photosynthesis and photoinhibition in plants.

9.4 *In Silico* Implementations

Membrane computing is part of the general intellectual adventure of searching for progress in computer science, at any level, from "purely" theoretical to "purely" practical, starting from biological processes – and their underlying "hardware" – polished by nature over billions of years. The hope is that *nature knows what it selects*, sometimes even without making it clear to us (part of nature, but *sapiens*, trying to understand–explain–generalize–formalize, possibly use in our benefit, irrespective of whether this benefit is "natural" or not). The success of genetic algorithms is just a convincing illustration of this situation: they work "unexpectedly well" in many applications. Membrane computing is somehow simultaneous with DNA computing (with a decade's delay if we count the theoretical beginnings and we consider paper [86], where the splicing operation was introduced, as the beginning of – theoretical – DNA computing, and with nobody-knows-how-many decades' delay from an experimental point of view). However, as we have also argued in the Introduction, it is not clear whether membrane computing should try to imitate DNA computing, and stay in the laboratory, or if it is better to imitate genetic algorithms and go to the electronic computer for implementations.

Because we do not have an answer to this basic question, but we do have illustrations of the second possibility, that of implementing/simulating P systems on a normal computer, we will briefly present here some attempts in this respect. It is important to underline the fact that "implementing" a membrane system on an existing electronic computer cannot be a real implementation, it is merely a simulation. As long as we do not have genuinely parallel hardware on which the parallelism (let aside the non-determinism) of membrane systems could be realized, what we obtain cannot be more than simulations, thus losing the main, good features of membrane systems. However, the implementations/simulations already circulated in this area have at least two important merits: they have a didactic value, and they can be used for simulating membrane systems which model certain real-life phenomena, with relevance for other domains than computer science itself, such as artificial life, ecology, and biochemistry. The next sections will discuss two applications of this type of membrane computing via simulation programs.

Among the first programs simulating P systems, we mention [137] and [64]. Let us call them *of the first generation*: dealing with symbol–objects, priorities, the dissolving action, not having a fancy screen appearance, but

fast enough for simulations. From the *second generation* one can consider the program(s) realized by Suzuki and Tanaka, see, for instance, [235], mainly meant to be used for long simulations of the evolution of systems with a small number of membranes, possibly only one, and producing statistical graphs and tables. All these programs are pretty modular, flexible concerning the inputs (membrane structures, multisets of objects, rules to process them), but the modularity is the explicit goal of the attempt reported in [167], which provides a "library" of modules meant to facilitate the implementation/simulation of a P system (both with active membranes and with "non-active" membranes).

Of the *third generation* we consider the program realized by "the Madrid team (Spain)", see [10, 11, 18, 19, 20], and that realized by "the Iaşi team (Romania)", see [41].

The Madrid program starts from a rather elaborate formalization, with detailed efforts paid to the data structures used for representing the ingredients of a P system, and with generalizations towards bio-languages inspired by this work. We refer to the above-mentioned papers for details, and we describe the Iaşi program here, mainly because it is available on the web page whose address was given in the Preface of the book. We will closely follow the presentation in [41].

The program, explicitly called a *membrane simulator*, is intended to be a tool for people working with P systems. It provides a graphical simulation of two types of P systems: *catalytic systems*, with symbol–objects, catalysts, priorities, and dissolving actions (in short, of type (cat, tar, pri, δ)), and systems *with active membranes*, hence with the possibility to divide membranes.

The main functions of the program are: (1) the interactive definition of a membrane system, (2) the visualization of a membrane system, (3) a graphical representation of the result of a computation, (4) saving and (re)loading a defined membrane system.

Apart from these functions the simulator has a debug module allowing step-by-step observations of the membrane system behaviour in order to detect possible errors in the definition of the user P system.

The internal representation of membrane structures is the tree.

The main screen of the application is divided into two windows, as shown in Figure 9.2.

The left window gives a tree representation of the membrane system and includes all the elements of such a system. We distinguish here objects (usual objects or catalysts in catalytic systems), and membranes (containing objects, rules, and priorities associated with them, as well as other membranes).

The right window is larger, and it provides the graphical representation of the membrane system described by the left window. In a Venn-like diagram, it also indicates the objects from each region.

In order to define a membrane system, the user interacts with the left window. A menu allows the specification of a membrane system by using

Fig. 9.2. The main screen of the Iaşi P systems simulator

Insert actions. The basic actions add new objects, membranes, rules, and priorities. For each new object or membrane, the user can also indicate a colour associated uniquely with that object or membrane, in order to identify the corresponding elements of the graphical representation. The tree representation from the left window and the graphical representation from the right window are synchronized and each action on the tree representation is reflected almost simultaneously in the graphical representation. The new defined elements are checked according to the definition of a membrane system. This prevents the user from defining "wrong rules" and keeps the membrane system consistent.

The *Edit* button of the menu allows the user to modify existing elements or their properties (for instance, one can change the name of objects, remove objects, etc.).

The actions of both the *Insert* and *Edit* buttons are context-sensitive; they can be used only when they are allowed to act on the selected element in the left window. This is a very useful property. When an element of the tree representation is selected, the menu bar will activate only the possible operations on that element. The user can easily see that the displayed dialog windows differ according to the selected element.

The *File* menu allows the user to save and reload a membrane system. From a technical point of view, these operations are realized by object serialization – a feature included in the system.

The simulation of a computation performed by a membrane system is started by using the *Run* function from the *Build* menu. After the user watches the graphical evolution and interprets the result, (s)he can return to the initial system by using the *Reload* function from the same *Build* menu. The *Build* menu also includes a *Debug* module. By using its three functions, *Start, Next,* and *Stop*, the user can observe the system evolution step-by-step.

The implementation described in [41] is written in MS Visual C++ 6.0 using the class hierarchy indicated in Figure 9.3.

Here is a short description of the main classes.

- The CSymbol class is used to represent the objects of the systems.
- The CNormalRule class was designed to represent all the rules of the membrane systems without the communication and division rules for the active membranes. These last two kinds of rules are represented by the CComRule class (for communication rules), the CElemDivRule class (for the division rules of the elementary membranes), and CCompDivRule class (for the division rules of a non-elementary membrane).
- The CPriority class is used to represent a priority relation between the rules defined in a membrane and it is used only for catalytic membrane systems.
- The CMembrane is a virtual class encapsulating the common functionality of a membrane. The following two classes are derived from this class: CCatalyticMembrane is used for membranes of the catalytic hierarchical systems and CActiveMembrane is used for membranes of the active membrane systems.
- CPSystem is a virtual class encapsulating the common functionality of a system. The following two classes are derived from this class: CCatalyticSystem encapsulates the operations for the catalytic systems and CActiveSystem encapsulates operations for the active membrane systems.

The architecture is designed such that all the operations over a system are realized by using the CCatalyticSystem and CActiveSystem classes. These classes represent a sort of interface between the display part and the working part of the application. This architecture gives flexibility and portability to the program.

As the authors of the program say, researchers working in P systems and membrane computing can use this software tool in order to simulate and to improve various membrane systems for solving various problems. The program may also be used for teaching, the graphical representation supporting the intuition of the way a P system works.

We conclude this section by pointing out that a promising strategy is to simulate – maybe to really *implement* – membrane systems on dedicated electronic hardware, as advocated in [136, 229], and [230], but no practical attempt in this respect has been reported so far.

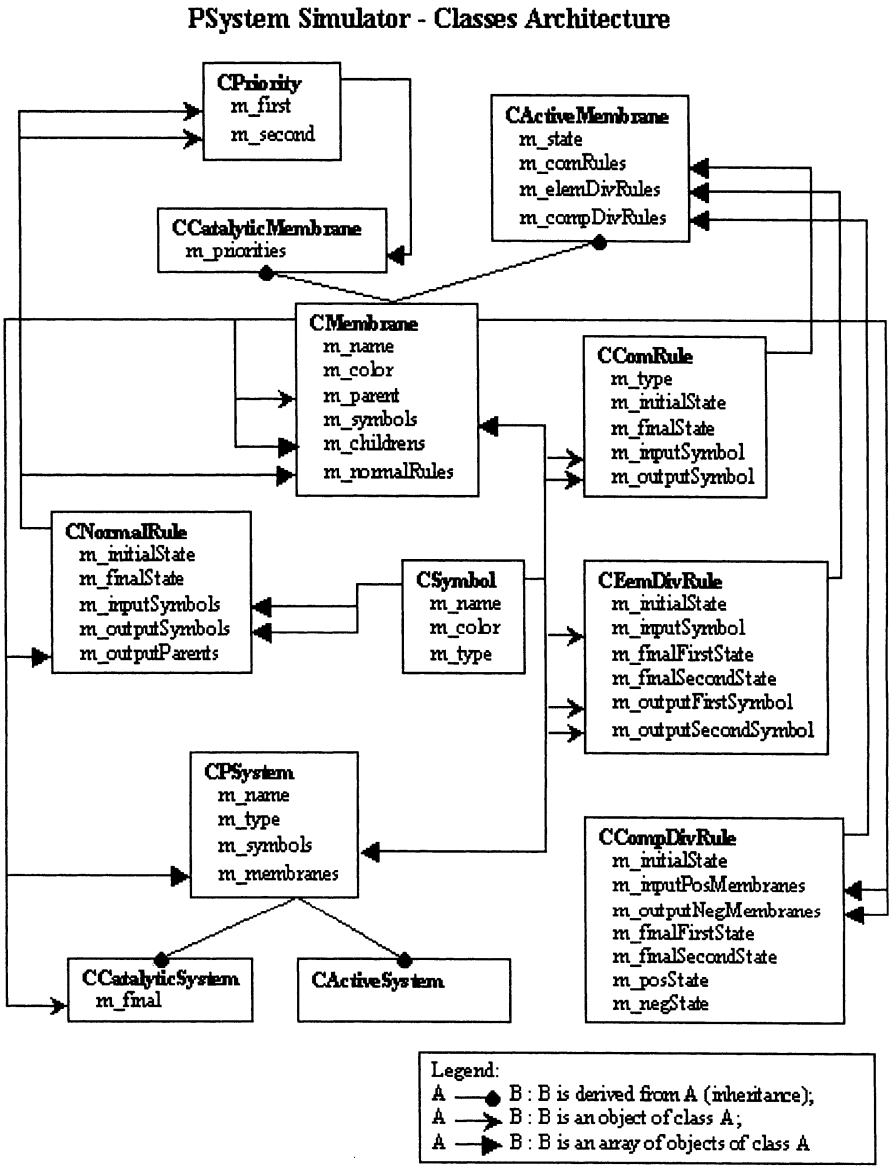


Fig. 9.3. The C++ class hierarchy of the Iaşi simulator

9.5 Artificial Life Applications

In the words of C. Langton, one of the founders of the domain, "Artificial life is the study of man-made systems that exhibit behaviors characteristic of natural living systems. It complements the traditional biological sciences concerned with the *analysis* of living organisms by attempting to *synthesize* life-like behaviors within computers and other artificial media.[...] By extending the empirical foundation upon which biology is based *beyond* the carbon-chain life that has evolved on Earth, Artificial life can contribute to theoretical biology by locating *life-as-we-know-it* within the larger picture of *life-as-it-could-be*", [124].

Oversimplifying, the main topics of artificial life are the *definition* of life, and the *simulation* of life on non-biological supports, such as computers. We do not enter into details about any of these topics, but we only point out that both of them essentially deal with *changes*, with the dynamics of complex systems, with interactions between the parts of a system and between systems and their environment. This almost directly links the domain to membrane computing, where we find tools able to capture many features related to the evolution of complex systems. The idea has been systematically explored by Suzuki and his collaborators, in a series of papers, [232, 233, 234], etc. We present here a few results reported in these papers.

Actually, Suzuki started his researches independently, using a sort of one-membrane P system, with multisets of objects placed inside, evolving by multiset processing rules. His objective was not to get a computing model, but systematic and general support for simulating interactions among agents evolving in a common space. A typical example is that dealt with in [233], where one considers the interaction of plants, herbivores, and carnivorous animals, living together in the same eco-system. We will return to this example with more details. In this framework, one does not look for theoretical results about the models, but one directly goes to computer implementations in order to perform simulations. This requires specific limitations on the size of the multisets considered, hence for applications where the number of objects present in the system is not larger than the considered limitations – implicitly, mainly questions about the appearance of cycles, the shape of cycles, and their succession in time, deadlocks which can appear in the range of the limits imposed on the system size, and other similar issues can be addressed. However, even in this somewhat restricted framework one can otreat rather interesting situations, some of them with a famous history – this is the case for the Brusselator, a model of chemical oscillations in the Belousov–Zabotinsky reaction, investigated in several places, such as [57, 168].

We start by briefly presenting the study of this model, following [236, 233], because of the almost direct possibility to formalize it in terms of membrane systems.

The abstract chemical reaction equations of the Brusselator are usually written in the form

$$A \to X,$$
$$B + X \to Y + D,$$
$$2X + Y \to 3X,$$
$$X \to E,$$

which is nothing more than the multiset processing rules (of a cooperative form)

$$r_1 : A \to X,$$
$$r_2 : BX \to YD,$$
$$r_3 : XXY \to XXX,$$
$$r_4 : X \to E.$$

All resulting objects remain in the same membrane. The alphabet used is

$$O = \{A, B, C, D, X, Y\}.$$

The problem is to investigate the concentration of X and Y, in dependence on the frequency of rule application (reaction rates), assuming that we get inputs continuously from the environment in the form of "objects" A and B. The simulation program used in [232, 233] allows a maximum size of the internal multiset of 5000 objects.

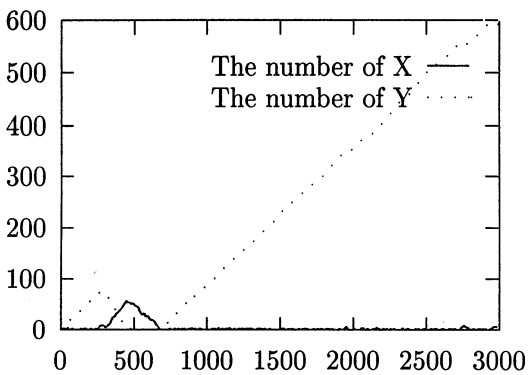


Fig. 9.4. Example of divergence

The results of the computer simulations have confirmed the findings of previous investigations of the same model, based on much more sophisticated analytical techniques, in general of differential equations type, see, for example, [57] and [168]: oscillations between the number of X and Y emerge, of three different types:

1. divergence and convergence (Figure 9.4, Figure 9.5),

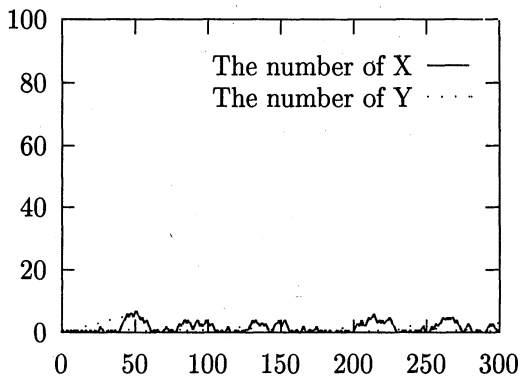


Fig. 9.5. Example of convergence

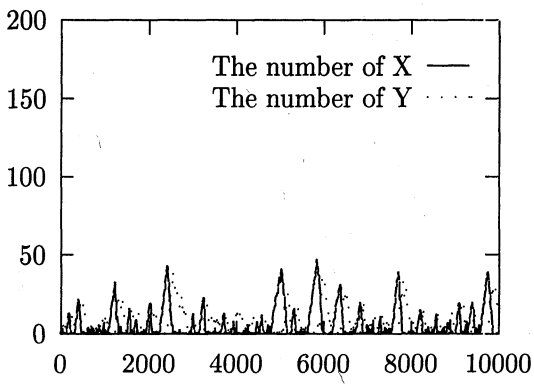


Fig. 9.6. Example of (quasi-) stable oscillation

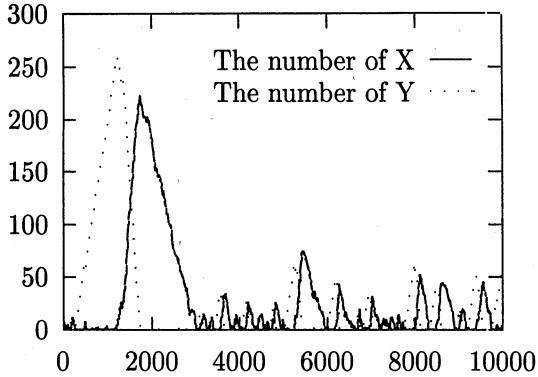
2. quasi-stable oscillations (Figure 9.6), and
3. unstable oscillations (Figure 9.7).

Here are, from [233], some comments about the results of the simulations carried out by Suzuki and his collaborators.

Divergence. When the frequency of the application of rule r_2 is much larger than that of r_1, for example 0.1 for r_1 and 0.45 for r_2, the system becomes unstable and in many cases it diverges.

In this case, a system generates X–Y oscillations (Figure 9.4). The system diverges easily on perturbation and it is difficult to predict whether we get oscillations or divergence.

Convergence. When the frequency of the application of r_1 is much larger than that of r_2, for instance 0.1 versus 0.04, the system converges (Figure 9.5). In this case almost all oscillations are X or Y oscillations, and only a few X–Y oscillations can be found.



(Quasi-) Stable oscillations. When the frequency of the applications of r_1 is slightly larger than that of r_2, for example 0.1 versus 0.15, the system is able to generate X–Y oscillations. Even if we introduce a perturbation to the system, it maintains this type of oscillation (Figure 9.6).

Unstable oscillation. When the frequency of r_2 is slightly larger than that of r_1, for instance 0.1 versus 0.35, the system exhibits an unstable oscillation. In this case, there are a few X or Y oscillations and almost all the oscillations become X–Y oscillations. A great variety of X–Y oscillation can be found (Figure 9.7).

Let us now pass to the investigations by the same team on the above-mentioned tritrophic eco-system, recalling some details from [233] and [232].

Fairly recent observations [51, 220] show that there are plants which defend themselves against herbivorous animals by producing certain volatiles which attract the carnivorous predators of those herbivores – and these volatiles are not the result of mechanical damage in general, but of the damage produced by a herbivore. This kind of game among three different species can be naturally modelled in terms of multisets of objects (four objects: plants, herbivores, carnivores, and volatiles), evolving together in a given closed space (the region of a membrane), according to given rules.

Let us denote by p a plant (this is called a "leaf" in [233]), by h a herbivore, by c a carnivore, and by v a volatile (a well-specified density of a volatile which attracts a carnivore, hence a "quantum" of a volatile); let us also use d to denote a "dead state" of a herbivore or a carnivore (thus, in total, we have five objects). The following interaction rules are considered in [233, 232]:

$r_1 : p \to pp$ (increase of plants),
$r_2 : ph \to hhv$ (a herbivore eats a plant, develops, and a volatile is released),
$r_3 : chv \to cc$ (a carnivore, attracted by a volatile, eats a herbivore and develops),

$$r_4 : c \to d \text{ (a carnivore dies)},$$
$$r_5 : h \to d \text{ (a herbivore dies)}.$$

Let us denote this set of rules by R_1.

As in the Brusselator case, *reaction rates* are associated with these rules, denoted by k_1, k_2, k_3, k_4, k_5, respectively, and governing the frequency of applications of rules.

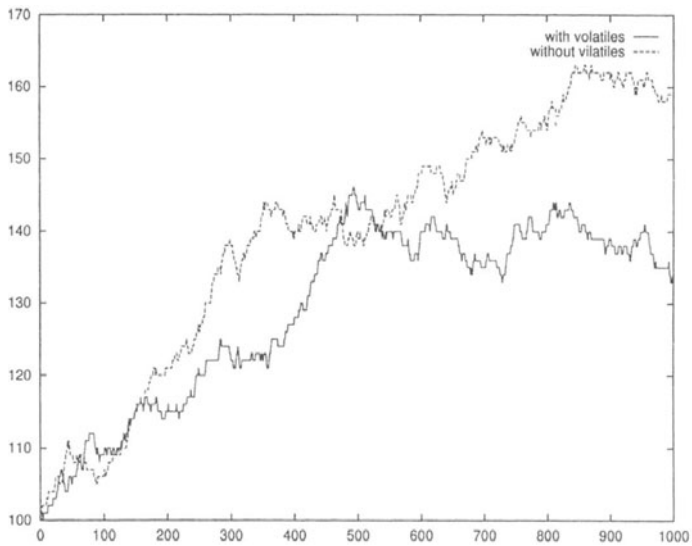


Fig. 9.8. The number of herbivores of R_1 and R_2 for $k_3 = 0.1$

All these reaction rates are inputs to the simulation, with the exception of k_2, which is computed according to the state of the eco-system, as follows:

$$k_2 = \frac{M(h)}{M(p) + M(h) + M(v) + M(c) + M(d)},$$

where by $M(\alpha)$ we have denoted the multiplicity of α in the system, $\alpha \in \{p, h, v, c, d\}$. One can also take care of the breeding rate of carnivores by changing the number of copies of c resulting from the application of rule r_3; for instance, we can take $r_3' : chv \to cccc$.

Table 9.1. The average of the number of herbivores after 1000 steps

k_3, k_8	0.1	0.2	0.3	0.4	0.9
average of herbivores (k_3)	141.96	49.79	49.75	49.77	~49.00
average of herbivores (k_8)	163.31	2.43	0.00	0.00	0.00

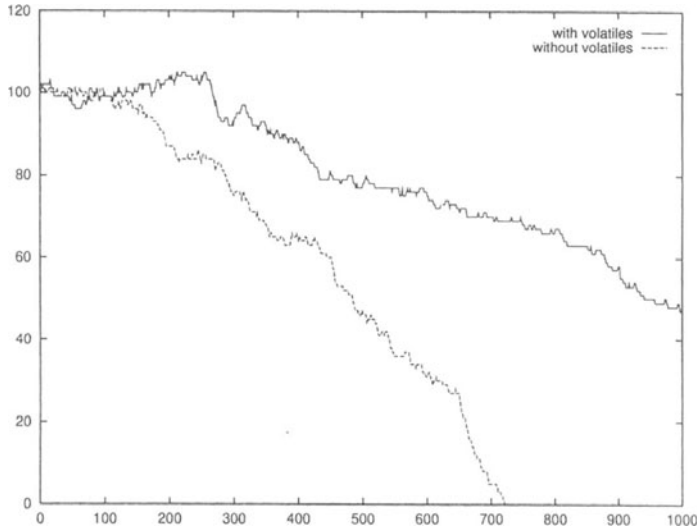


Fig. 9.9. The number of herbivores of R_1 and R_2 for $k_3 = 0.2$

The "life" of the previous model, where the plants produce volatiles, was compared with the "life" of the following model, where there is no volatile:

r_6 : $p \to pp$ (increase of plants),
r_7 : $ph \to hh$ (a herbivore eats a plant and develops, but no volatile is released),
r_8 : $ch \to cc$ (a carnivore eats a herbivore and develops),
r_9 : $c \to d$ (a carnivore dies),
r_{10} : $h \to d$ (a herbivore dies).

This set of rules is denoted by R_2.

The reaction rates for rules r_1, r_4, r_5, r_6, r_9, and r_{10} were set as 0.5, 0.1, 0.1, 0.5, 0.1, and 0.1, respectively.

In the simulation, k_3 and k_8 are changed from 0.1 to 0.9 and in the initial state the number of herbivores is set to 100. Then the correlation between the population size of herbivores and the values of k_3, k_8 is observed. In the case when volatiles are produced, k_3 is regarded as the ability of volatiles to attract carnivores, and in the case without volatiles k_3 corresponds to the ability of carnivores to find herbivores.

When k_3 and k_8 are equal to 0.1 the population of herbivores increases, with the specification that the number of herbivores is larger in the case of R_1 than in the case of R_2 (Figure 9.8). When k_3 and k_8 are equal to 0.1, the averages of the number of herbivores after 1000 steps are 141.96 and 163.31, respectively (Table 9.1, in 100 trials). Hence, when plants produce volatiles, herbivores are caught by carnivores to a larger extent than when the plants do not produce volatiles (Figure 9.9).

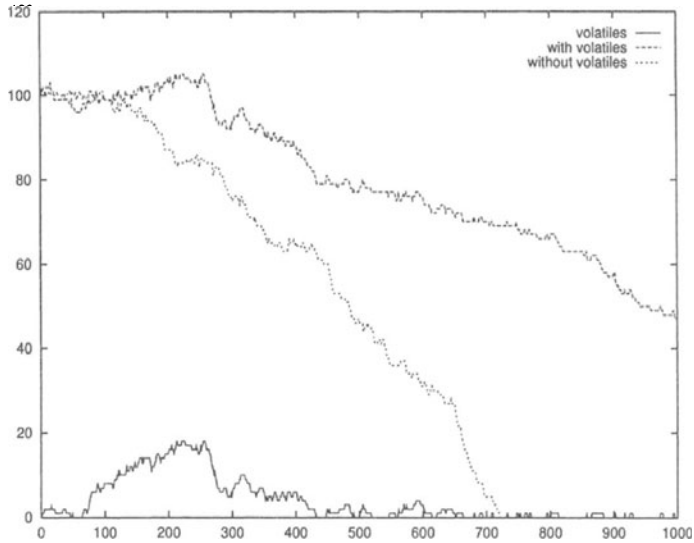


Fig. 9.10. The number of volatiles and of herbivores for $k_3 = k_8 = 0.2$

However, when k_3 and k_8 are greater than 0.2, the number of herbivores decreases. This indicates that when plants do not generate volatiles, the number of herbivores is increased, as expected (Figure 9.10). This tendency is preserved when k_3 and k_8 continue to grow. When k_3 and k_8 exceed 0.2, in the case of R_2 the herbivores become extinct, but in the case of R_1 the herbivores are never extinct – which is no longer as expected (Figure 9.11).

For instance, during simulations, it was found that the herbivores can keep their population for at least 1000 generations in the system where the plants generate volatiles, but the herbivores were exterminated by the carnivores in around 100 generations in the system where plants do not generate volatiles.

This result suggests the somewhat paradoxical possibility that herbivores themselves induce the volatiles for their survival. Here is a real case of this type, as described in [233]: "This may be true in the tritrophic system consisting of plants, two-spotted spider mites (*Tetranychus urticae*) and predatory mites (*Phytoseiulus persimilis*), [51]. Two-spotted spider mite is a tiny (ca. 0.6 mm) herbivore. However, due to their rapid population increase on a plant, they tend to overexploit the plant. A kidney bean leaf infested by the spider mites started emitting volatiles that attract predatory mites *P. persimilis*. Once in the prey colony, *P. persimilis* overexploit the spider mites. However, the volatiles were induced only after the spider mites increased over a certain number per leaf (more than ca. 100-300 females per small plant) and there will be a time lag between the emission of the volatiles and the visitation by the predatory mites. Thus, the spider mites of the next generation that disperse from the current patch before the plant starts emitting the volatiles can be free from the predator. Such spider mites will establish

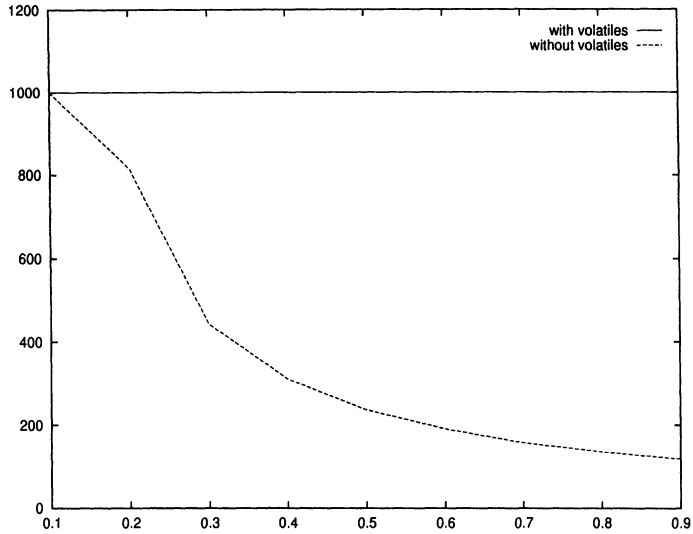


Fig. 9.11. The average value of herbivore extinction for R_1 and R_2. The horizontal axis illustrates the value of k_3 and k_8 and the vertical axis illustrates the average of the number of extinctions in 1000 trials

a new colony nearby. At the same time, the original colony may be exterminated by the predators. This cycle may be one of the defense strategies of the two-spotted spider mites against the predatory mites that search for them with the volatiles."

It is rather interesting that such a sophisticated interaction has been sensed/predicted by the formal model.

Of course, part of these investigations can be considered as only partially dealing with membrane systems in their real form, as one uses only one membrane, with an interaction with the environment in the case of the Brusselator model, and without even interacting with the environment in the tritrophic eco-system case. This can be due both to the computer programs available for simulations and to the specific applications – although the extract quoted above from [233] suggests that in reality there are several "membranes" placed at the same level, interacting and influencing each other, and this interaction is, in fact, the "real explanation" for the counter-intuitive findings of the simulations. Passing to systems with several membranes, communicating among them seems necessary (and possible now, with the programs mentioned in the previous section).

A more ambitious approach, as suggested in Problem Q36 (consider an eco-membrane system), is probably a way to model (and simulate) a large variety of similar eco-system issues.

9.6 A Simulation of Photosynthesis

We recall now, closely following Nishida [169], an approach to photosynthesis in plants, which is essentially based on using a membrane system. As in the applications discussed in the previous section, one aims to obtain a framework for computer simulations, but this time the system involves two membranes, and also the inter-region of one of the membranes, in the sense of Section 9.3. Moreover, one works with multiplicities of objects which are not necessarily natural numbers, but they can be positive real numbers, describing, for instance, concentrations.

Together with Nishida, let us start by recalling some basic information about photosynthesis.

Photosynthesis of higher plants occurs in the so-called *chloroplasts*, where many membrane-surrounded structures called *thylakoids* can be found. The region enclosed by the thylakoid membrane is called the *lumen*. The space between the chloroplast envelope and thylakoids is named the *stroma*.

Photosynthetic reactions are classified into two groups: *light reactions* and *dark reactions*. Light reactions separate water into O_2 and H^+, reduce NADP to NADPH, and synthesize ATP. The enzymes which act as catalysts of light reactions are embedded in the thylakoid membrane. Dark reactions make starch from CO_2 and H_2O using the reduction power of NADPH and chemical energy of ATP. Dark reactions occur in the stroma.

Light reactions are carried out by two photosystems, PSI and PSII, which consist of many proteins and pigments such as chlorophyll and carotenoid. Pigments catch light energy and give the energy to proteins in PSI and PSII. Using the energy, PSII splits water on the lumen side of the thylakoid membrane and transports H^+ from the stroma to the lumen; PSI reduces NADP on the stroma side of the thylakoid membrane (see Figure 9.12).

These light reactions are summarized as follows:

$$2H_2O(L) \overset{PS,\gamma}{\rightarrow} O_2(L) + 4H^+(L), \tag{9.1}$$

$$2NADP(S) + 2H^+(S) \overset{PS,\gamma}{\rightarrow} 2NADPH(S), \tag{9.2}$$

$$4H^+(S) \overset{PS,\gamma}{\rightarrow} 4H^+(L), \tag{9.3}$$

where γ stands for a photon and (L) and (S) stand for molecules in the lumen and stroma, respectively. We omit electrons from the chemical formulae above. The electric potential of the electrons is the motive power of the active transport of H^+. Thus, reactions (9.1)–(9.3) link to each other, and, as a consequence of these reactions, the concentration of H^+ in the lumen increases or the pH in the lumen decreases.

In the thylakoid membrane there is another reaction which synthesizes ATP using the chemical potential of H^+ as follows:

$$ADP(S) + P(S) + nH^+(L) \rightarrow ATP(S) + nH^+(S), \tag{9.4}$$

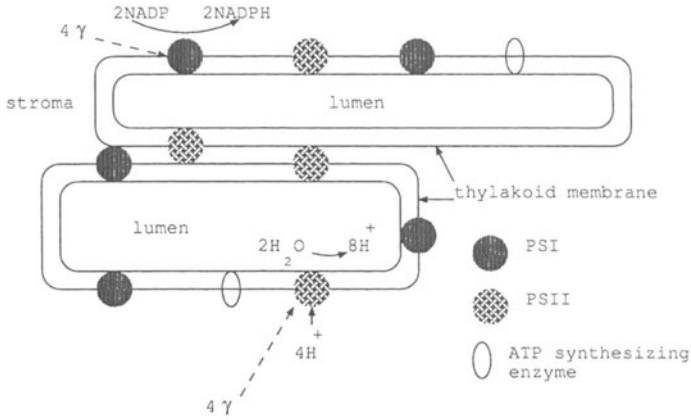


Fig. 9.12. The structures of the thylakoid membrane and light reactions; γ stands for a photon

where P stands for phosphoric acid and n is the number of H^+'s which are required in the ATP synthesis. The number n depends upon the difference of pH between the lumen and the stroma; usually $n = 3$. The dark reactions are essentially the reverse of (9.2)

$$\mathrm{NADPH}(S) \rightarrow \mathrm{NADP}(S) + \mathrm{H}^+(S). \tag{9.5}$$

In a moderate luminosity, the products of light reactions are all consumed by dark reactions. But, if the light is strong or dark reactions stop for some reason, the products, NADPH at the stroma and H^+ at the lumen, become harmful to the structures of chloroplasts; they may destroy proteins, pigments, lipids, and so on. Consequently, the plants have mechanisms to repress light reactions in high luminosity. The mechanism is called *photoinhibition.* The details of photoinhibition are unknown. Here are two suggested photoinhibition mechanisms. The first consists of reactions which occur when there is no NADP in the stroma and the pH in the lumen is less than 5 [88]. The reactions are summarized as follows:

$$2\mathrm{H_2O}(L) \overset{\mathrm{PS},\gamma}{\rightarrow} \mathrm{O_2}(L) + 4\mathrm{H}^+(L) \tag{9.6}$$

$$\mathrm{O_2}(S) + 4\mathrm{H}^+(S) \overset{\mathrm{PS},\gamma}{\rightarrow} 2\mathrm{H_2O}(S) \tag{9.7}$$

$$4\mathrm{H}^+(L) + \mathrm{O_2}(L) \rightarrow 2\mathrm{H_2O}(L), \tag{9.8}$$

$$2\mathrm{NADPH}(S) + 2\mathrm{H}^+(S) + \mathrm{O_2}(S) \rightarrow 2\mathrm{H_2O}(S) + 2\mathrm{NADP}(S). \tag{9.9}$$

The second mechanism decreases the activity of PSII. PSII has many chlorophyll–protein complexes (called LHCII) as antennas to capture photons. PSII releases LHCII in strong light and decreases its light collecting activity [6].

In order to pass from these biochemical facts to a computer simulation, Nishida considered membrane systems with multisets of objects whose multiplicities are positive real numbers. We do not enter into the technical details of the mathematical background, but only mention that in this framework the (non-cooperative) rules are of the form

$$X \to (mul_1 Y_1, tar_1)(mul_2 Y_2, tar_2) \dots (mul_l Y_l, tar_l),$$

where mul_i is a polynomial formula indicating the multiplicity of object Y_i which will be placed in the region indicated by tar_i in the next configuration of the system. If such a rule is applied in a region, then the multiplicity of X in that region goes to 0 (because of the maximal parallelism of using the rule), and the multiplicity of Y_i in the region indicated by tar_i increases by mul_i for every $i = 1, \dots, l$. The formulae which give the multiplicities are computed at each step, according to probabilities which control the application of the rules, depending on the concentrations of objects in the region.

The membrane system constructed in [169], called *Photo*, uses a membrane structure with two membranes $\mu = [_S[_L\ ]_L]_S$, but it also takes into account the inter-region of the inner membrane; hence we may write

$$\mu = [_S[_{Le}[_{Li}\ ]_{Li}]_{Le}]_S.$$

In what follows, we use this three-membrane structure, with the elements of the system (multisets of initial objects and rules) identified by the subscripts of these three membranes, S, Le, and Li ("e" and "i" stand for "external" and "internal", respectively, "S" and "L" stand for "stroma" and "lumen", respectively). Thus, the system is

$$Photo = (O, \mu, w_S, w_{Le}, w_{Li}, R_S, R_{Le}, R_{Li}).$$

Because we are interested in the continuous evolution of the process, there is no output membrane. The external membrane represents the envelope membrane of the chloroplast, the internal membrane represents the thylakoid membrane, while the region delimited by this membrane is the lumen.

Actually, there are many thylakoids in a chloroplast, so we might use a membrane structure of the form $[_S[_L\ ]_L[_L\ ]_L \cdots [_L\ ]_L]_S$, with the corresponding inter-region for membranes $[_L\ ]_L$. Also, there is experimental evidence which suggests that there are differences between stacked thylakoids and unstacked thylakoids. However, in what follows, only the above-mentioned simple membrane structure, μ, is considered.

The multisets w_S, w_{Li}, w_{Le} represent the contents of the stroma, lumen, and thylakoid, respectively, at the beginning of simulations.

The alphabet of objects of *Photo* is

$$O = \{\mathrm{H}^+, \text{ NADP, NADPH, PSI, PSIIh, PSIIl}\}.$$

The sets of evolution rules R_S, R_{Le}, and R_{Li} consist of $\{R_S1, R_S2, R_S3\}$, $\{R_{Le}1, R_{Le}2\}$, and $\{R_{Li}1\}$, respectively, given as follows (as usual, the indication *here* is ommited):

$$\begin{aligned}
R_S1(2) &: \mathrm{NADP} \to (1-2r_2)A_S(\mathrm{NADP})\ \mathrm{NADP} \\
&\qquad 2r_2A_S(\mathrm{NADP})\ \mathrm{NADPH}, \\
R_S2(5,9) &: \mathrm{NADPH} \to (1-r_5-2r_9)A_S(\mathrm{NADPH})\ \mathrm{NADPH} \\
&\qquad (r_5+2r_9)A_S(\mathrm{NADPH})\ \mathrm{NADP}, \\
R_S3(2,7,9) &: \mathrm{H}^+ \to (1-6r_1-4r_7-2r_9)A_S(\mathrm{H}^+)\ \mathrm{H}^+ \\
&\qquad (4r_1A_S(\mathrm{H}^+)\mathrm{H}^+,\ in_{Li}), \\
R_{Le}1 &: \mathrm{PSIIh} \to ((1-r_{PS})A_{Le}(\mathrm{PSIIh})+C_{PS})\ \mathrm{PSIIh} \\
&\qquad r_{PS}A_{Le}(\mathrm{PSIIh})\ \mathrm{PSIIl}, \\
R_{Le}2 &: \mathrm{PSIIl} \to (A_{Le}(\mathrm{PSIIl})-C_{PS})\ \mathrm{PSIIl}, \\
R_{Li}1(1,4,6,8) &: \mathrm{H}^+ \to ((1-r_8)A_{Li}(\mathrm{H}^+)+(r_1+r_4)C_L-r_5C_{ATP})\ \mathrm{H}^+ \\
&\qquad (r_5C_{ATP}\mathrm{H}^+,\ out).
\end{aligned}$$

The figures in parentheses correspond to the reaction numbers given when describing the photosynthesis. The constants C_{PS}, C_L, and C_{ATP} correspond to the "house keeping" PSIIh recovering, the maximum water splitting in the lumen, and the maximum ATP synthesis in the thylakoid membrane, respectively. The "probabilities" r_x are given by the following formulae:

$$\begin{aligned}
r_1 &= 1-\exp(-p_1\nu A_{Le}(\mathrm{PSIIh}))+1-\exp(-p_1'\nu A_{Le}(\mathrm{PSIIl})), \\
r_2 &= 1-\exp(-p_1\nu A_{Le}(\mathrm{PSI})), \\
r_4 = r_6 = r_7 &= \begin{cases} 0, & \text{if } A_S(\mathrm{NADP}) \geq \theta_4, \\ r_2, & \text{otherwise}, \end{cases} \\
r_5 &= 2(1-\exp(-p_1\nu_0 A_{Le}(\mathrm{PSI}))), \\
r_8 = r_9 &= \begin{cases} 0, & \text{if } A_{Li}(\mathrm{H}^+) \leq \theta_8, \\ \frac{1-\exp(-p_6(A_{Li}(\mathrm{H}^+)-A_S(\mathrm{H}^+)))}{4}, & \text{otherwise}, \end{cases} \\
r_{PS} &= 1-\exp(-p_{PS}\nu A_{Le}(\mathrm{PSIIh})),
\end{aligned}$$

where p_1, p_1', p_6, and p_{PS} are probabilities, ν stands for the strength of light, θ_4 and θ_6 are constants which stand for thresholds, and ν_0 stands for the normal strength of light. Photons are treated as if they are molecules. Because reactions (9.1)–(9.3) link to each other, they occur with the same probability p_1. Since PSII catalyses reactions (9.1) and (9.3) and PSIIl has less activity than PSIIh, p_1' should be smaller than p_1.

The behaviour of *Photo* was simulated for the parameter values and the initial multisets (with real number multiplicities) shown in Tables 9.2 and 9.3, respectively. Six cases were examined: {photosystem decreasing activity is disabled ($p_{PS} = 0$), photosystem decreasing activity is enabled ($p_{PS} =$

$10^{-8})\} \times \{$normal light ($\nu = 1000$), strong light ($\nu = 10000$), very strong light ($\nu = 10^5$)$\}$.

Table 9.2. Parameters for the simulation

constants	C_{PS} $1000(1-\exp(-0.01))$	C_L 30000	C_{ATP} 500
probabilities	$p_1 = 10^{-8}$	$p_1' = 10^{-9}$	$p_6 = 10^{-5}$
ν_0	1000		
thresholds	$\theta_4 = 200$	$\theta_6 = 30000$	

Table 9.3. The initial values for the simulation

x	$A_S(x)$	$A_{Le}(x)$	$A_{Li}(x)$
H^+	3000	0	3000
NADP	1000	0	0
NADPH	0	0	0
PSI	0	1000	0
PSIIh	0	1000	0
PSIIl	0	0	0

During simulations under very strong light conditions, multiplicities of some objects go to unrealistic values, for example negative values, but the program resets these values to biologically reasonable values, for example 0.

Table 9.4. The average pH values in the lumen obtained by computer simulations of *Photo* during 100 time steps.

decreasing PS activity	light conditions		
	normal (1)	strong (×10)	very strong (×100)
disabled	5.07	4.91	4.42
enabled	5.07	4.98	4.61

The average pH values in the lumen during 100 time steps are shown in Table 9.4. Figure 9.13 illustrates the changes of A_{Le}(PSIIh) and pH values in the lumen under strong light conditions with a decreasing activity of the PS structure.

Low pH is harmful to plants, and in Table 9.4 we can indeed see that decreasing activity of PS is an effective photoinhibition mechanism under strong and very strong light conditions.

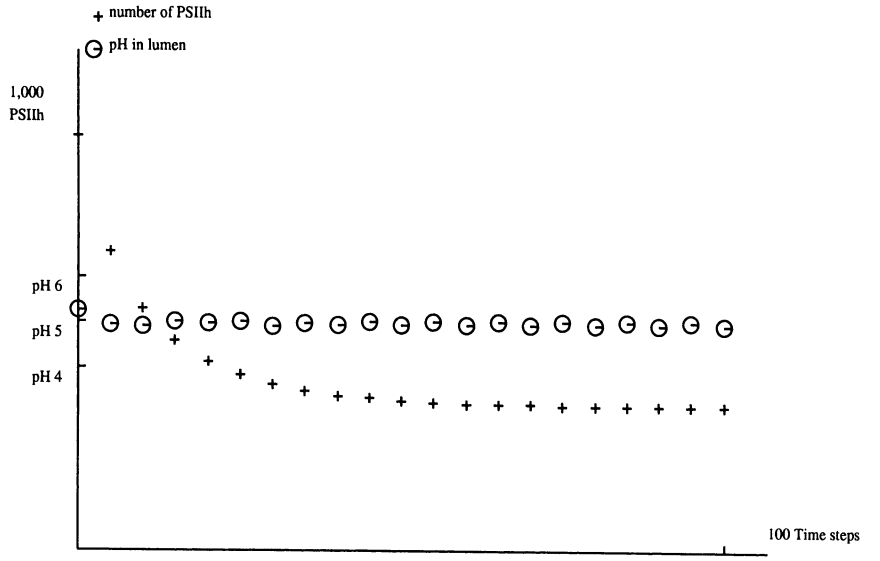


Fig. 9.13. Changes of A_{Le}(PSIIh) and pH values in the lumen under strong light conditions with decreasing activity of PS structure

These simulations were done for parameters with no close experimental checking, but the versatility of the system/program *Photo* makes possible various types of experiments, and various types of questions can be answered by simulations – for instance, oscillations, as encountered in the case of the Brusselator, in the previous section. Further results in this respect are expected in the near future, since the experiments are ongoing.

Open Problems

Problem	Page	Problem	Page
Q1	63 and 70	Q21	248
Q2	74	Q22	261
Q3	77	Q23	273
Q4	102	Q24	301
Q5	105	Q25	313
Q6	123	Q26	320
Q7	127	Q27	325
Q8	140	Q28	341
Q9	141	Q29	341
Q10	144 and 159	Q30	345
Q11	144	Q31	357
Q12	153	Q32	364
Q13	168 and 207	Q33	364
Q14	202	Q34	365
Q15	208	Q35	365
Q16	218	Q36	365
Q17	220	Q37	366
Q18	226	Q38	371
Q19	232	Q39	375
Q20	232	Q40	399

Universality Results

1.	Theorem 3.3.3	:	$NOP_1(coo) = NRE$
2.	Theorem 3.4.3	:	$NOP_2(cat, in, pri) = NRE$
3.	Theorem 3.4.6	:	$NOuP_2(2cat, in, pri) = NRE$
4.	Theorem 3.4.7	:	$NOP_2(2cat, in) = NRE$
5.	Theorem 3.6.1	:	$NOP_4(cat, in, \delta, \tau) = NRE$
6.	Theorem 3.6.2	:	$NOP_2(2cat, con) = NRE$
7.	Theorem 3.6.3	:	$NOP_1(ncoo, rule) = NRE$
8.	Theorem 3.6.4	:	$NOP_2(cat, in, proR) = NRE$
9.	Theorem 3.6.5	:	$NOP_6(ncat, in, proS_2) = NRE$
10.	Theorem 3.6.6	:	$NOP_4(ncoo, in, proS_3)$
			$= NOP_3(ncoo, in, proS_4) = NRE$
11.	Theorem 3.6.7	:	$NOP_1(ncoo, proS_6) = NRE$
12.	Theorem 3.6.8	:	$NOP_3(ncoo, in, inhS_6) = NRE$
13.	Theorem 3.7.3	:	$LOP_1(cat, proR) = RE$
14.	From [119]	:	$NOP_2(cat, in, pri, \textit{time-varying}) = NRE$
15.	From [118]	:	$NOP_*(2cat, simple) = NRE$
16.	Theorem 4.2.1	:	$NOP_2(sym_2, anti_2) = NRE$
17.	Theorem 4.2.2	:	$NOP_5(sym_2, anti_0) = NRE$
18.	Theorem 4.2.3	:	$NOP_2(sym_5, anti_0) = NRE$
19.	Theorem 4.2.4	:	$NOP_3(sym_4, anti_0) = NRE$
20.	Theorem 4.2.5	:	$NOP_3(sym_0, anti_2) = N'RE$
21.	Theorem 4.3.1	:	$NOP_3(psym_2, anti_0)$
			$= NOP_3(fsym_2, anti_0) = NRE$
22.	Theorem 4.4.4	:	$LOP_*(psym_2, panti_*)$
			$= LOP_*(fsym_2, fanti_*) = RE$
23.	Theorem 4.5.1	:	$NOP_2(carr_{3,3}) = NRE$
24.	Corollary 4.5.1	:	$NOP_2(carr_{*,2}) = NRE$
25.	From [180]	:	$NOP_3(sym_2, anti_0, rename, \delta, \tau) = NRE$
26.	Theorem 5.1.6	:	$LSP_2(rw, in, pri) = RE$
27.	Theorem 5.1.9	:	$LSP_4(rw, in, \delta, \tau) = RE$
28.	Theorem 5.2.1	:	$ELSP_1(rw, rule) = RE$
29.	Theorem 5.2.2	:	$LSP_2(rw, in, inh) = RE$
30.	Theorem 5.2.4	:	$LSP_1(rw, cond) = RE$
31.	Theorem 5.2.7	:	$LSP_2(rw, empty, sub_2) = RE$
32.	Theorem 5.2.8	:	$LSP_4(rw, sub_2, symb) = RE$

33.	Theorem 5.2.9	:	$ELSP_5(rw, sub_2, empty) = RE$
34.	Theorem 5.2.10	:	$ELSP_6(rw, symb, empty)$ $= LSP_6(rw, symb, symb) = RE$
35.	Theorem 5.2.12	:	$LSP_6(rw, in, repl_2, halt) = RE$
36.	Theorem 5.2.16	:	$NSP_6(rw, in, repl_2, halt) = NRE$
37.	Theorem 5.2.17	:	$ELSP_4(rw, in, ppar) = RE$
38.	Theorem 5.3.1	:	$ELSP_2(spl, in, (1, 2, 2, 1)) = RE$
39.	Corollary 5.3.1	:	$ELSP_2(spl, in, (2, 1, 1, 2)) = RE$
40.	Theorem 5.3.3	:	$ELSP_1(spl, rule)$ $= LSP_2(spl, in, rule) = RE$
41.	Theorem 5.3.4	:	$ELSP_*(spl, move) = RE$
42.	Theorem 5.4.1	:	$ELSP_2(con_{one}, in, (0, 3; 0, 3)) = RE$
43.	Theorem 5.4.2	:	$ELSP_3(con_{one}, in, (0, 2; 2, 0)) = RE$
44.	Theorem 5.5.2	:	$ELSP_1(insdel, (3, 1; 2, 0)) = RE$
45.	Theorem 5.5.3	:	$ELSP_4(insdel, in, (1, 1; 1, 1)) = RE$
46.	Theorem 5.5.4	:	$ELSP_1(insdel, (*, 0; 2, 0), cond) = RE$
47.	from [56]	:	$LSP_5(rw, in, leftmost)$ $= ELSP_4(rw, in, leftmost) = RE$
48.	From [114]	:	$ELSP_*(rw, in, max\text{-}par) = RE$
49.	From [111]	:	$ELSP_7(rw + cont_{reg}, tar) = RE$
50.	From [161]	:	$ELSP_3(rw, split, s\text{-}p\text{-}concat) = RE$
51.	Theorem 6.1.1	:	$ELStP_3(spl, go, (0, 2, 1, 0))$ $= ELStP_3(spl, go, (1, 0, 0, 2)) = RE$
52.	Corollary 6.2.1	:	$NOtP_{5,5}(sym_2, anti_0) = NRE$
53.	Corollary 6.2.2	:	$1NOtP_{11,11}(sym_2, anti_0) = NRE$
54.	Theorem 6.2.3	:	$1NOtP_{3,3}(sym_2, anti_1) = NRE$
55.	Theorem 6.2.4	:	$NOtP_{4,2}(sym_2, anti_1) = NRE$
56.	Theorem 6.3.1	:	$NOnP_{2,5}(ncoo, min, \beta) = NRE,$ $\beta \in \{repl, one, spread\}$
57.	Theorem 6.3.2	:	$NOnP_{4,4}(ncoo, min, \beta) = NRE,$ $\beta \in \{one, spread\}$
58.	Theorem 6.3.3	:	$NOnP_{2,2}(coo, \alpha, \beta) = NRE,$ $\alpha \in \{min, par, max\}, \beta \in \{repl, one, spread\}$
59.	Theorem 7.2.1	:	$NOP_3(activ, (a, b, c)) = NRE$
60.	Corollary 7.3.1	:	$NOP_{1,2,2}(cat, in, pri, mcre)$ $= NOP_{1,4,4}(cat, in, \delta, \tau, mcre) = NRE$
61.	Theorem 7.3.4	:	$ELSP_{1,3,4}(rw, in, \delta, mcre) = RE$
62.	Theorem 8.4.2	:	$LSP_2(rw_{lr}, valuat) = RE$
63.	Theorem 8.5.1	:	$LSP_{3,*}(rw_{lr}, create, \delta) =$ $LSP_{4,*}(rw_{any}, create, \delta) =$ $LSP_{3,9}(rw_{lr}, create, separate, \delta) = RE.$
64.	From [71]	:	$NOP_*(conformons) = NRE$
65.	From [198]	:	$NOP_*(ncoo, in, pri, energy) = NRE$
66.	From [26]	:	$ELSP_7(gemmation) = RE$

References

1. L.M. Adleman: Molecular Computation of Solutions to Combinatorial Problems. *Science*, 226 (November 1994), 1021–1024.
2. J. Aguado, T. Bălănescu, T. Cowling, M. Gheorghe, M. Holcombe, F. Ipate: P Systems with Replicated Rewriting and Stream X-Machines (Eilenberg Machines). *Fundamenta Informaticae*, 49, 1–3(2002), 17–33.
3. A.V. Aho, J.D. Ullman: *The Theory of Parsing, Translation, and Compiling.* Prentice Hall, Englewood Cliffs, NJ, Vol. I: 1971, Vol. II: 1973.
4. B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, J.D. Watson: *Molecular Biology of the Cell.* 3rd ed., Garland Publishing, New York, 1994.
5. J.A. Anderson: *An Introduction to Neural Networks.* The MIT Press, Cambridge, MA, 1996.
6. B. Andersson: Thylakoid Membrane Dynamics in Relation to Light Stress and Photoinhibition. In *Trends in Photosynthesis Research* (J. Barber, M.G. Guerrero, H. Medran, eds.), Intercept, Andover, 1992, 71–86.
7. M.A. Arbib: *Brains, Machines, and Mathematics.* 2nd ed., Springer, Berlin, 1987.
8. M.A. Arbib: *The Metaphorical Brain: An Introduction to Schemes and Brain Theory.* Wiley Interscience, New York, 1988.
9. I.I. Ardelean: The Relevance of Biomembranes for P Systems. *Fundamenta Informaticae*, 49, 1–3 (2002), 35–43.
10. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. Mingo: A Recursive Algorithm for Describing Evolution in Transition P Systems. In [156], 19–30.
11. F. Arroyo, A.V. Baranda, J. Castellanos, C. Luengo, L.F. Mingo: Structures and Bio-Language to Simulate Transition P Systems on Digital Computers. In [37], 1–16.
12. F. Arroyo, A.V. Baranda, J. Castellanos, Gh. Păun: Membrane Computing: The Power of (Rule) Creation. *Journal of Universal Computer Science*, 8, 3 (2002), 369–381.
13. A. Atanasiu, C. Martín-Vide: Arithmetic with Membranes. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 5–20.
14. A. Atanasiu, C. Martín-Vide: Recursive Calculus with Membranes. *Fundamenta Informaticae*, 49, 1–3 (2002), 45–59.
15. A. Atanasiu, C. Martín-Vide: P Systems and Context-Free Languages. *Actas Primer Congreso Espanol de Algoritmos Evolutivos y Bioinspirados*, Merida, 2002 (E. Alba *et al.*, eds.), 341–346.
16. J.-P. Banâtre, A. Coutant, D. Le Métayer: A Parallel Machine for Multiset Transformation and Its Programming Style. *Future Generation Computer Systems*, 4 (1988), 133–144.

17. J.-P. Banâtre, P. Fradet, D. Le Métayer: Gamma and the Chemical Reaction Model: Fifteen Years After. In [37], 17–44.
18. A.V. Baranda, J. Castellanos, F. Arroyo, R. Gonzalo: Towards an Electronic Implementation of Membrane Computing: A Formal Description of Nondeterministic Evolution in Transition P Systems. *Proc. 7th Int. Meeting on DNA Based Computers* (N. Jonoska, N.C. Seeman, eds.), Tampa, Florida, USA, 2001, 273–282.
19. A.V. Baranda, J. Castellanos, F. Arroyo, C. Luengo: Bio-Language for Computing with Membranes. *Proc. ECAL 2001*, Praga, September 2001 (J. Kelemen, P. Sosik, eds.), *Lecture Notes in Artificial Intelligence*, 2159, Springer, Berlin, 2001, 176–185.
20. A.V. Baranda, J. Castellanos, R. Gonzalo, F. Arroyo, L.F. Mingo: Data Structures for Implementing Transition P Systems in Silico. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 21–32.
21. C. Bennett: Logical Reversibility of Computation. *IBM Journal of Research and Development*, 17 (1973), 525–532.
22. G. Berry, G. Boudol: The Chemical Abstract Machine. *Theoretical Computer Science*, 96 (1992), 217–248.
23. D. Besozzi: *P Systems with Gemmation.* Masters Thesis, Universitá degli Studi dell'Insubria, 2000.
24. D. Besozzi, C. Ferretti, G. Mauri, C. Zandron: Parallel Rewriting P Systems with Deadlock. *Proc. 8th Int. Meeting on DNA Based Computers* (M. Hagiya, A. Ohuchi, eds.), Sapporo, Japan, 2002, 171–183.
25. D. Besozzi, G. Mauri, Gh. Păun, C. Zandron: Gemmating P Systems: Collapsing Hierarchies. *Theoretical Computer Science*, to appear.
26. D. Besozzi, C. Zandron, G. Mauri, N. Sabadini: P Systems with Gemmation of Mobile Membranes. *Proc. ICTCS 2001*, Turin, October 2001, *Lecture Notes in Computer Scienc*, 2202, Springer, Berlin, 2001, 136–153.
27. D.S. Blank *et al.* (24 co-authors): Connectionist Symbol Processing: Dead or Alive? *Neural Computing Surveys*, 2 (1999), 1–40.
28. H. Bordihn, H. Fernau: Accepting Grammars and Systems. An Overview. In *Developments in Language Theory. At the Crossroads of Mathematics, Computer Science, and Biology* (J. Dassow, G. Rozenberg, A. Salomaa, eds.), World Scientific, Singapore, 1996, 199–208.
29. P. Bottoni, A. Labella, C. Martín-Vide, Gh. Păun: Rewriting P Systems with Conditional Communication. In *Formal and Natural Computing. Essays Dedicated to Grzegorz Rozenberg* (W. Brauer, H. Ehrig, J. Karhumäki, A. Salomaa, eds.), *Lecture Notes in Computer Science*, 2300, Springer, Berlin, 2002, 325–353.
30. P. Bottoni, C. Martín-Vide, Gh. Păun, G. Rozenberg: Membrane Systems with Promoters/Inhibitors. *Acta Informatica*, to appear.
31. D. Bray: Protein Molecules as Computational Elements in Living Cells. *Nature*, 376 (July 1995), 307–312.
32. T.A. Brown: *Gene Cloning. An Introduction.* Chapman & Hall, London, 1996.
33. C.S. Calude, M.J. Dinneen, Gh. Păun, eds.: *Pre-Proceedings of Workshop on Multiset Processing*, Curtea de Argeş, Romania, CDMTCS Technical Report 140, University of Auckland, 2000.
34. C. Calude, Gh. Păun: Global Syntax and Semantics for Recursively Enumerable Languages. *Fundamenta Informaticae*, 4, 2 (1981), 245–254.
35. C.S. Calude, Gh. Păun: Computing with Cells and Atoms in a Nutshell. *Complexity*, 6, 1 (2000), 38–48.

36. C.S. Calude, Gh. Păun: *Computing with Cells and Atoms.* Taylor and Francis, London, 2001.
37. C.S. Calude, Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *Multiset Processing. Mathematical, Computer Science, and Molecular Computing Points of View. Lecture Notes in Computer Science*, 2235, Springer, Berlin, 2001.
38. L. Cardelli, A.G. Gordon: Mobile Ambients. *Proc. FoSSaCS'98* (M. Nivat, ed.), *Lecture Notes in Computer Science*, 1378, Springer, Berlin, 1998, 140–155.
39. J. Castellanos, Gh. Păun, A. Rodríguez-Patón: P Systems with Worm-Objects. *IEEE 7th. Int. Conf. on String Processing and Information Retrieval, SPIRE 2000*, La Coruna, Spain, 64–74.
40. C. Choffrut, ed.: *Automata Networks. Lecture Notes in Computer Science*, 316, Springer, Berlin, 1988.
41. G. Ciobanu, D. Paraschiv: Membrane Software. A P System Simulator. *Fundamenta Informaticae*, 49, 1–3 (2002), 61–66.
42. G. Ciobanu, B. Tanasă: Coordination of Gene Expression by Pipes and Signals. *Fundamenta Informaticae*, 49, 1–3 (2002), 67–80.
43. M. Conrad: The Price of Programmability. In *The Universal Turing Machine: A Half-Century Survey* (R. Herken, ed.), Kammerer and Unverzagt, Hamburg, 1988, 285–307.
44. E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun: *Grammar Systems. A Grammatical Approach to Distribution and Cooperation.* Gordon and Breach, London, 1994.
45. E. Csuhaj-Varjú, A. Kelemenova, J. Kelemen, Gh. Păun: Eco-Grammar Systems – A Grammatical Framework for Life-Like Interactions. *Artificial Life*, 3 (1997), 1–28.
46. E. Czeizler: *Attacking NP-Complete Problems by Means of P Systems with Pre-Computed Membrane Structures.* Diploma Thesis, Faculty of Mathematics, University of Bucharest, 2002.
47. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory.* Springer, Berlin, 1989.
48. J. Dassow, Gh. Păun: On the Power of Membrane Computing. *Journal of Universal Computer Science*, 5, 2 (1999), 33–49.
49. J. Dassow, Gh. Păun: Concentration Controlled P Systems. *Acta Cybernetica*, 15, 1 (2001), 9–24.
50. J. Dassow, Gh. Păun, G. Thierrin, S. Yu: Tree-Systems of Morphisms. *Acta Informatica*, 38, 2 (2001), 131–153.
51. M. Dicke, J. Takabayashi, C. Schütte, O.R. Krips: Behavioral Ecology of Plant-Carnivore Interactions: Variation in Response of Phytoseiid Mites to Herbivore-Induced Plant Volatiles. *Experimental and Applied Acarology*, 22 (1997), 595–601.
52. R. Dominé: *Theorem Proving Using Membrane Computing.* Masters Thesis, Dresden University, 2001.
53. A. Dovier, C. Piazza, G. Rossi: Multiset Constraints and P Systems. In [37], 103–122.
54. S. Eilenberg: *Automata, Languages, and Machines.* Academic Press, New York, Vol. A: 1974, Vol. B: 1976.
55. H. Fernau: Closure Properties of Ordered Languages. *Bulletin of the EATCS*, 58 (1996), 159–162.

56. C. Ferretti, G. Mauri, Gh. Păun, C. Zandron: On Three Variants of P Systems with String-Objects. In [156], 63–76.

57. R.J. Field, M. Burger: *Oscillations and Traveling Waves in Chemical Systems.* John Wiley and Sons, 1985.

58. R. Freund: Generalized P Systems. *Proc. Conf. on Fundamentals of Computation Theory, FCT'99*, Iaşi, 1999 (G. Ciobanu, Gh. Păun, eds.), *Lecture Notes in Computer Science*, 1684, Springer, Berlin, 1999, 281–292.

59. R. Freund: Generalized P Systems with Splicing and Cutting/Recombination. *Grammars*, 2, 3 (1999), 189–199.

60. R. Freund: Sequential P Systems. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 77–88.

61. R. Freund: Special Variants of P Systems Inducing an Infinite Hierarchy with Respect to the Number of Membranes. *Bulletin of the EATCS*, 75 (October 2001), 209–219.

62. R. Freund, F. Freund: Molecular Computing with Generalized Homogeneous P Systems. *Proc. 6th Int. Meeting on DNA Based Computers* (A. Condon, G. Rozenberg, eds.), Leiden, 2000, 113–125.

63. R. Freund, C. Martín-Vide, Gh. Păun: From Regulated Rewriting to Computing with Membranes: Collapsing Hierarchies. *Theoretical Computer Science*, to appear.

64. R. Freund, M. Oswald: A Program Simulating P Systems. Manuscript, 1999.

65. R. Freund, M. Oswald: Variants of GP Systems. In [156], 77–88.

66. R. Freund, M. Oswald: GP Systems with Forbidding Context. *Fundamenta Informaticae*, 49, 1–3 (2002), 81–102.

67. R. Freund, Gh. Păun: On the Number of Non-Terminals in Graph-Controlled, Programmed, and Matrix Grammars. *Proc. Third Int. Conf. on Universal Machines and Computations*, Chişinău, Moldova, 2001 (M. Margenstern, Y. Rogozhin, eds.), *Lecture Notes in Computer Science*, 2055, Springer, Berlin, 2001, 214–225.

68. P. Frisco: Membrane Computing Based on Splicing: Improvements: In [33], 100–111.

69. P. Frisco: On Two Variants of Splicing P Systems. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 89–100.

70. P. Frisco, H.J. Hoogeboom, P. Sant: A Direct Construction of a Universal P System. *Fundamenta Informaticae*, 49, 1–3 (2002), 103–122.

71. P. Frisco, S, Ji: Info-Energy P Systems. *Proc. 8th Int. Meeting on DNA Based Computers* (M. Hagiya, A. Ohuchi, eds.), Sapporo, Japan, 2002, 161–170.

72. B.S. Galiukschov: Semicontextual Grammars. *Matematika Logica i Matematika Linguistika*, Talinin University, 1981, 38–50 (in Russian).

73. M.R. Garey, D.J. Johnson: *Computers and Intractability. A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, 1979.

74. V. Geffert: Normal Forms for Phrase-Structure Grammars. *RAIRO. Theoretical Informatics and Applications*, 25 (1991), 473–496.

75. J.-L. Giavitto, O. Michel: MGS – Implementing a Unified View on Four Biologically Inspired Computing Models. In [156], 107–122.

76. J.-L. Giavitto, O. Michel: The Topological Structures of Membrane Computing. *Fundamenta Informaticae*, 49, 1–3 (2002), 123–145.

77. S. Ginsburg: *The Mathematical Theory of Context-Free Languages.* McGraw-Hill, New York, 1966.

78. S. Ginsburg: *Algebraic and Automata-Theoretic Properties of Formal Languages.* North-Holland, Amsterdam, 1975.
79. D.S. Goodsell: *The Machinery of Life.* Springer, New York, 1998.
80. D.E. Green, S. Ji: The Electromechanochemical Model of Mitochondrial Structure and Function. In *The Molecular Basis for Electron Transport* (J. Schultz, B.F. Cameron, eds.), Academic Press, New York, 1972, 1–44.
81. J. Gruska: Descriptional Complexity of Context-Free Languages. *Proc. Symp. on Mathematical Foundations of Computer Science, MFCS*, High Tatras, 1973, 71–83.
82. J. Gruska: *Quantum Computing.* McGraw-Hill, New York, 1999.
83. S.R. Hameroff, J.E. Dayhoff, R. Lahoz-Beltra, A.V. Samsonovich, S. Rasmussen: Models for Molecular Computation: Conformational Automata in the Cytoskeleton. *IEEE Computer*, 25 (1992), 30–39.
84. M. Harrison: *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.
85. D. Hauschild, M. Jantzen: Petri Nets Algorithms in the Theory of Matrix Grammars. *Acta Informatica*, 31 (1994), 719–728.
86. T. Head: Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviors. *Bulletin of Mathematical Biology*, 49 (1987), 737–759.
87. T. Head: Aqueous Simulations of Membrane Computations. *Romanian Journal of Information Science and Technology*, 5, 4 (2002).
88. H.W. Heldt: *Plant Biochemistry and Molecular Biology.* Oxford University Press, Oxford, 1997.
89. G.T. Herman, G. Rozenberg: *Developmental Systems and Languages.* North-Holland, Amsterdam, 1975.
90. J. Hoffmeyer: Semiosis and Living Membranes. *First Seminário Avançado de Comunicaçao e Semiótica. Biosemiótica e Semiótica Cognitiva*, Sao Paolo, August 1998, 9–23.
91. J. Hoffmeyer: Surfaces Inside Surfaces. On the Origin of Agency and Life. *Cybernetics and Human Knowing*, 5, 1 (1998), 33–42.
92. H.J. Hoogeboom: Carriers and Counters. P Systems with Carriers vs. (Blind) Counter Automata. Submitted, 2002.
93. M. Holcombe: X-Machines as a Basis for Dynamic System Specification. *Software Engineering Journal*, 3, 2 (1988), 69–76.
94. M. Holcombe, T. Bălănescu, M. Gheorghe, P. Radovici-Mărculescu: On Testing Generalized Stream X-Machines. In *Recent Topics in Mathematical and Computational Linguistics* (C. Martín-Vide, Gh. Păun, eds.), The Publishing House of the Romanian Academy, Bucharest, 2000.
95. M. Holcombe, F. Ipate: *Correct Systems. Building a Bussiness Process Solution.* Springer, Berlin, 1998.
96. J.E. Hopcroft, J.D. Ullman: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.
97. O.H. Ibarra: Reachability and Safety in Queue Systems with Counters and Pushdown Stack. *Pre-Proceedings 5th Int. Conf. on Implementation and Application of Automata*, 2000, 120–129.
98. M. Ionescu, C. Martín-Vide, A. Păun, Gh. Păun: Membrane Systems with Symport/Antiport: (Unexpected) Universality Results. *Proc. 8th Int. Meeting*

on DNA Based Computers (M. Hagiya, A. Ohuchi, eds.), Sapporo, Japan, 2002, 151–160.

99. M. Ionescu, C. Martín-Vide, Gh. Păun: P Systems with Symport/Antiport Rules: The Traces of Objects. *Grammars*, 5 (2002).
100. M. Ito, C. Martín-Vide, Gh. Păun: A Characterization of Parikh Sets of ET0L Languages in Terms of P Systems. In *Words, Semigroups, and Transducers* (M. Ito, Gh. Păun, S. Yu, eds.), World Scientific, Singapore, 2001, 239–254.
101. S. Ji: The Bhopalator: A Molecular Model of the Living Cell Based on the Concepts of Conformons and Dissipative Structures. *Journal of Theoretical Biology*, 116 (1985), 399–426.
102. S. Ji: The Bhopalator: An Information/Energy Dual Model of the Living Cell. In [156], 123–142.
103. S. Kauffman: *The Origins of Order.* Oxford University Press, Oxford, 1993.
104. S. Kauffman: *At Home in the Universe.* Oxford University Press, Oxford, 1995.
105. J. Kelemen, A. Kelemenova: A Grammar-Theoretic Treatment of Multiagent Systems. *Cybernetics and Systems*, 23 (1992), 621–633.
106. S.C. Kleene: Representation of Events in Nerve Nets and Finite Automata. *Automata Studies*, Princeton University Press, Princeton, NJ, 1956, 3–42.
107. S.N. Krishna: Computing with Simple P Systems. In [33], 124–137.
108. S.N. Krishna: *Languages of P Systems. Computability and Complexity.* PhD Thesis, IIT Madras, 2001.
109. S.N. Krishna: Infinite Hierarchies on Some Variants of P Systems. Submitted, 2002.
110. S.N. Krishna, K. Krithivasan, R. Rama: P Systems with Picture Objects. *Acta Cybernetica*, 15, 1 (2001), 53–74.
111. S.N. Krishna, K. Lakshmanan, R. Rama: Hybrid P Systems. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 111–124.
112. S.N. Krishna, K. Lakshmanan, R. Rama: On the Power of P Systems with One-Sided and Erasing Contextual Rules. *Fundamenta Informaticae*, 49, 1–3 (2002), 167–178.
113. S.N. Krishna, R. Rama: A Variant of P Systems with Active Membranes: Solving NP-Complete Problems. *Romanian Journal of Information Science and Technology*, 2, 4 (1999), 357–367.
114. S.N. Krishna, R. Rama: On the Power of P Systems Based on Sequential/Parallel Rewriting. *International Journal of Computer Mathematics*, 77, 1–2 (2000), 1–14.
115. S.N. Krishna, R. Rama: A Note on Parallel Rewriting in P Systems. *Bulletin of the EATCS*, 73 (February 2001), 147–151.
116. S.N. Krishna, R. Rama: Insertion-Deletion P Systems. *Proc. 7th Int. Meeting on DNA Based Computers* (N. Jonoska, N.C. Seeman, eds.), Tampa, Florida, USA, 2001, 350–359.
117. S.N. Krishna, R. Rama: P Systems with Replicated Rewriting. *Journal of Automata, Languages, and Combinatorics*, 6, 3 (2001), 345–350.
118. S.N. Krishna, R. Rama: On the Power of Simple P Systems. *Journal of Universal Computer Science*, 8 (2002).
119. S.N. Krishna, R. Rama: On Some Classes of Deterministic P Systems. Submitted, 2001.
120. S.N. Krishna, R. Rama: Breaking DES Using P Systems. *Theoretical Computer Science*, to appear.

121. K. Krithivasan, M. Madhu: Generalized Normal Forms for Rewriting P Systems. *Acta Informatica*, to appear.

122. M. Kudlek, C. Martín-Vide, Gh. Păun: Toward FMT (Formal Macroset Theory). In [33], 149–158, and [37], 123–134.

123. R. Lahoz-Beltra: Molecular Automata Assembly: Principles and Simulation of Bacterial Membrane Construction. *BioSystems*, 44 (1997), 209–229.

124. C.G. Langton: Artificial Life. In *Artificial Life, II* (C.G. Langton, C. Taylor, J.D. Farmer, S. Rasmussen, eds.), *Proc. of the Workshop on Artificial Life*, Santa Fe, 1990, Santa Fe Institute Studies in the Science of Complexity, Proc. Vol. X, Addison-Wesley, Reading, MA, 1990, 1–47.

125. A. Lindenmayer: Mathematical Models for Cellular Interaction in Development, Parts I and II. *Journal of Theoretical Biology*, 18 (1968), 280–315.

126. J.Q. Liu, K. Shimohara: Evolutionary Dynamics for Heterogeneous P Systems. *Journal of Xi'an Mining Institute*, 2001.

127. W.R. Loewenstein: *The Touchstone of Life. Molecular Information, Cell Communication, and the Foundations of Life.* Oxford University Press, New York, Oxford, 1999.

128. P.L. Luisi: The Chemical Implementation of Autopoiesis. In *Self-Production of Supramolecular Structures* (G.R. Fleishaker *et al.*, eds.), Kluwer, Dordrecht, 1994.

129. S.S. Mader: *Biology.* 5th ed., McGraw-Hill, Boston, 1996.

130. M. Madhu, K. Krithivasan: Inter-Membrane Communication in P Systems. *Romanian Journal of Information Science and Technology*, 3, 4 (2000), 335–352.

131. M. Madhu, K. Krithivasan: P Systems with Membrane Creation: Universality and Efficiency. *Proc. Third Int. Conf. on Universal Machines and Computations*, Chişinău, Moldova, 2001 (M. Margenstern, Y. Rogozhin, eds.), *Lecture Notes in Computer Science*, 2055, Springer, Berlin, 2001, 276–287.

132. M. Madhu, K. Krithivasan: P Systems with Dynamic Membrane Polarization. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 135–154.

133. M. Madhu, K. Krithivasan: Universality Results for Some Variants of P Systems. In [37], 237–254.

134. M. Madhu, K. Krithivasan: Contextual P Systems. *Fundamenta Informaticae*, 49, 1–3 (2002), 167–178.

135. M. Madhu, K. Krithivasan: Improved Results About the Universality of P Systems. *Bulletin of the EATCS*, 76 (February 2002), 162–168.

136. M. Madhu, V.S. Murty, K. Krithivasan: Hardware Realization of P Systems with Carriers. *Proc. 8th Int. Meeting on DNA Based Computers* (M. Hagyia, ed.), Sapporo, Japan, 2002 (poster).

137. M. Malita: Membrane Computing in Prolog. In [33], 159–175.

138. V. Manca: Monoidal Systems and Membrane Systems. In [33], 176–190.

139. V. Manca: Membrane Algorithms for Propositional Satisfiability. In [156], 181–192.

140. V. Manca: Monoidals for Molecules and Membranes. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 155–170.

141. V. Manca: DNA and Membrane Algorithms for SAT. *Fundamenta Informaticae*, 49, 1–3 (2002), 205–221.

142. V. Manca, C. Martín-Vide, Gh. Păun: On the Power of P Systems with Replicated Rewriting. *Journal of Automata, Languages, and Combinatorics*, 6, 3 (2001), 359–374.

143. S. Marcus: Contextual Grammars. *Revue Roumaine de Mathématiques Pures et Appliquées*, 14 (1969), 1525–1534.

144. S. Marcus: Membranes Versus DNA. *Fundamenta Informaticae*, 49, 1–3(2002), 223–227.

145. M. Margenstern, C. Martín-Vide, Gh. Păun: Computing with Membranes; Variants with an Enhanced Membrane Handling. *Proc. 7th Int. Meeting on DNA Based Computers* (N. Jonoska, N.C. Seeman, eds.), Tampa, Florida, USA, 2001, 53–62.

146. M. Margenstern, Gh. Păun, Y. Rogozhin: On the Power of the (Molecular) Crowd: Set-Conditional String Processing. *10th Int. Conf. on Automata and Formal Languages*, Debrecen, Hungary, 2002.

147. C. Martín-Vide, V. Mitrana: P Systems with Valuations. In *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer, London, 2000, 154–166.

148. C. Martín-Vide, V. Mitrana, Gh. Păun: On the Power of P Systems with Valuations. *Computación y Sistemas*, 5, 2 (2001), 120–127.

149. C. Martín-Vide, A. Păun, Gh. Păun: On the Power of P Systems with Symport Rules. *Journal of Universal Computer Science*, 8, 2 (2002), 317–331.

150. C. Martín–Vide, A. Păun, Gh. Păun, G. Rozenberg: Membrane Systems with Coupled Transport: Universality and Normal Forms. *Fundamenta Informaticae*, 49, 1–3 (2002), 1–15.

151. C. Martín-Vide, Gh. Păun: String Objects in P Systems. *Proc. of Workshop on Algebraic Systems, Formal Languages and Computations*, Kyoto, 2000, RIMS Kokyuroku, Kyoto University, 2000, 161–169.

152. C. Martín-Vide, Gh. Păun: Computing with Membranes. One More Collapsing Hierarchy. *Bulletin of the EATCS*, 72 (October 2000), 183–187.

153. C. Martín-Vide, Gh. Păun: Computing with Membranes (P Systems): Universality Results. *Proc. Third Int. Conf. on Universal Machines and Computations*, Chişinău, Moldova, 2001 (M. Margenstern, Y. Rogozhin, eds.), *Lecture Notes in Computer Science*, 2055, Springer, Berlin, 2001, 82–101.

154. C. Martín-Vide, Gh. Păun: *Elements of Formal Language Theory for Membrane Computing.* Technical Report 21/01 of the Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, 2001.

155. C. Martín-Vide, Gh. Păun: Language Generating by Means of Membrane Systems. *Bulletin of the EATCS*, 75 (October 2001), 199–218.

156. C. Martín-Vide, Gh. Păun, eds.: *Pre-Proceedings of Workshop on Membrane Computing*, Curtea de Argeş, Romania, August 2001, Technical Report 17/01 of Research Group on Mathematical Linguistics, Rovira i Virgili University, Tarragona, 2001.

157. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríguez-Patón: *Tissue P Systems.* Turku Center for Computer Science–TUCS Technical Report 421, September 2001, and *Theoretical Computer Science*, in press.

158. C. Martín–Vide, Gh. Păun, A. Rodríguez-Patón: On P Systems with Membrane Creation. *Computer Science Journal of Moldova*, 9, 2 (2001), 134–145.

159. C. Martín-Vide, Gh. Păun, A. Rodríguez-Patón: P Systems with Immediate Communication. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 171–182.

160. C. Martín-Vide, Gh. Păun, G. Rozenberg: Membrane Systems with Carriers. *Theoretical Computer Science*, 270 (2002), 779–796.

161. J.L. Maté, A. Rodríguez-Patón: On the Power of P Systems with DNA-Worms. In [156], 209–220.

162. J.L. Maté, A. Rodríguez-Patón, A. Silva: On the Power of P Systems with DNA-Worm-Objects. *Fundamenta Informaticae*, 49, 1–3 (2002), 229–239.

163. A. Mateescu, V. Mitrana: Parallel Finite Automata Systems Communicating by States. *International Journal of Foundations of Computer Science*, to appear.

164. J. Matýsek, P. Sosík: Membrane Computing: When Communication is Enough. *Proc. Conf. on Unconventional Models of Computation*, 2002.

165. W.S. McCulloch, W.H. Pitts: A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathematical Biophysics*, 5 (1943), 115–133.

166. M. Minsky: *Computation: Finite and Infinite Machines.* Prentice Hall, 1967.

167. D.V. Nicolau Jr., G. Solana, F. Fulga, D.V. Nicolau Sr.: A "C" Library for Implementing P Systems on the Electronic Computer (Abstract). *Fundamenta Informaticae*, 49, 1-3 (2001), 241–248.

168. G. Nicolis, I. Prigogine: *Exploring Complexity. An Introduction.* W.H. Freeman, San Francisco, 1989.

169. T.Y. Nishida: Simulations of Photosynthesis by a K-Subset Transforming System with Membranes. *Fundamenta Informaticae*, 49, 1–3 (2002), 249–259.

170. A. Obtulowicz: Membrane Computing and One-Way Functions. *International Journal of Foundations of Computer Science*, 12, 4 (2001), 551–558.

171. A. Obtulowicz: Deterministic P Systems for Solving SAT Problem. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 195–202.

172. A. Obtulowicz: On P Systems with Active Membranes Solving Integer Factorizing Problem in a Polynomial Time. In [37], 267–286.

173. A. Obtulowicz: Note on Some Recursive Family of P Systems with Active Membranes. Submitted, 2001.

174. Ch.P. Papadimitriou: *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

175. A. Păun: On P Systems with Membrane Division. In *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer, London, 2000, 187–201.

176. A. Păun: On P Systems with Global Rules. *Proc. 7th Int. Meeting on DNA Based Computers* (N. Jonoska, N.C. Seeman, eds.), Tampa, Florida, USA, 2001, 43–52.

177. A. Păun: On P Systems with Partially Parallel Rewriting. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 203–210.

178. A. Păun: P Systems with String-Objects: Universality Results. In [156], 229–242.

179. A. Păun, Gh. Păun: The Power of Communication: P Systems with Symport/Antiport. *New Generation Computers*, 20, 3 (2002).

180. A. Păun, Gh. Păun, A. Rodríguez-Patón: Further Remarks on P Systems with Symport Rules. *Annals of Al.I. Cuza University, Iaşi, Mathematics-Informatics Series*, to appear.

181. A. Păun, Gh. Păun, G. Rozenberg: Computing by Communication in Networks of Membranes. Submitted, 2001.

182. A. Păun, M. Păun: On Membrane Computing Based on Splicing. In *Where Mathematics, Computer Science, Linguistics, and Biology Meet* (C. Martín-Vide, V. Mitrana, eds.), Kluwer, Dordrecht, 2001, 409–422.

183. Gh. Păun: *Marcus Contextual Grammars.* Kluwer, Dordrecht, 1997.

184. Gh. Păun: Computing by Carving. *Soft Computing*, 3, 1 (1999), 30–36.

185. Gh. Păun: Computing with Membranes. An Introduction. *Bulletin of the EATCS*, 67 (February 1999), 139–152.

186. Gh. Păun: Computing with Membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143, and Turku Center for Computer Science-TUCS Report No. 208, 1998 (www.tucs.fi).

187. Gh. Păun: Computing with Membranes – A Variant. *International Journal of Foundations of Computer Science*, 11, 1 (2000), 167–182.

188. Gh. Păun: Computing with Membranes: Attacking NP-Complete Problems. In *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer, London, 2000, 94–115.

189. Gh. Păun: *Computing with Membranes (P Systems): Twenty Six Research Topics.* CDMTCS Technical Report 119, University of Auckland, 2000.

190. Gh. Păun: From Cells to Computers: Computing with Membranes (P Systems). *BioSystems*, 59, 3 (2001), 139–158.

191. Gh. Păun: Further Research Topics About P Systems. In [156], 243–250.

192. Gh. Păun: P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal of Automata, Languages and Combinatorics*, 6, 1 (2001), 75–90.

193. Gh. Păun, M. Pérez-Jiménez, F. Sancho-Caparrini: On the Reachability Problem for P Systems with Symport/Antiport Rules. *10th Int. Conf. on Automata and Formal Languages*, Debrecen, Hungary, 2002.

194. Gh. Păun, G. Rozenberg: A Guide to Membrane Computing. *Theoretical Computer Science*, to appear.

195. Gh. Păun, G. Rozenberg, A. Salomaa: *DNA Computing. New Computing Paradigms.* Springer, Berlin, 1998; Tokyo, 1999; Moscow, 2002.

196. Gh. Păun, G. Rozenberg, A. Salomaa: Membrane Computing with External Output. *Fundamenta Informaticae*, 41, 3 (2000), 259–266.

197. Gh. Păun, Y. Sakakibara, T. Yokomori: P Systems on Graphs of Restricted Forms. *Publicationes Mathematicae*, 60 (2002).

198. Gh. Păun, Y. Suzuki, H. Tanaka: P Systems with Energy Accounting. *International Journal of Computer Mathematics*, 78, 3 (2001), 343–364.

199. Gh. Păun, Y. Suzuki, H. Tanaka, T. Yokomori: On the Power of Membrane Division in P Systems. *Theoretical Computer Science*, to appear.

200. Gh. Păun, G. Thierrin: Multiset Processing by Means of Systems of Finite State Transducers. *Pre-Proceedings of Workshop on Implementing Automata WIA99*, Potsdam, August 1999, Preprint 5/1999 of University Potsdam (O. Boldt, H. Jürgensen, L. Robbins, eds.), XV, 1–17.

201. Gh. Păun, T. Yokomori: Membrane Computing Based on Splicing. *Preliminary Proc. of Fifth Int. Meeting on DNA Based Computers* (E. Winfree, D. Gifford, eds.), MIT, June 1999, 213–227.

202. Gh. Păun, T. Yokomori: Simulating H Systems by P Systems. *Journal of Universal Computer Science*, 6, 1 (2000), 178–193.

203. Gh. Păun, S. Yu: On Synchronization in P Systems. *Fundamenta Informaticae*, 38, 4 (1999), 397–410.

204. Z. Pawlak: *Rough Sets. Theoretical Aspects of Reasoning about Data.* Kluwer, Dordrecht, 1992.

205. A. Pérez-Jiménez, M.J. Pérez-Jiménez, F. Sancho-Caparrini: Computing a Partial Mapping by a P System: Design and Verification. *Proc. Conf. Unconventional Models of Computation*, 2002.

206. M.J. Pérez-Jiménez, F. Sancho-Caparrini: A Formalization of Basic P Systems. *Fundamenta Informaticae*, 49, 1–3 (2002), 261–272.

207. M.J. Pérez-Jiménez, F. Sancho-Caparrini: Verifying a P System Generating Squares. *Romanian Journal of Information Science and Technology*, 5, 1–2 (2002), 181–191.

208. M.J. Pérez-Jiménez, F. Sancho-Caparrini: Verification of a Non-deterministic Transition P System Solving SAT Problem. *8th Int. Meeting on DNA Based Computers*, Sapporo, Japan, 2002 (poster).

209. M.J. Pérez-Jiménez, F. Sancho-Caparrini: Computing a Partial Mapping by P Systems: Design and Verification. Submitted, 2001.

210. I. Petre: A Normal Form for P Systems. *Bulletin of the EATCS*, 67 (February 1999), 165–172.

211. I. Petre, L. Petre: Mobile Ambients and P Systems. *Journal of Universal Computer Science*, 5, 9 (1999), 588–598.

212. D. Pixton: Regularity of Splicing Languages. *Discrete Applied Mathematics*, 69 (1996), 101–124.

213. D. Pixton: Splicing in Abstract Families of Languages. *Theoretical Computer Science*, 234 (2000), 135–166.

214. R. Rama: Computing with P Systems. In [33], 218–235.

215. A. Rodríguez-Patón: On the Universality of P Systems with Membrane Creation. *Bulletin of the EATCS*, 74 (June 2001), 229–234.

216. Y. Rogozhin: Small Universal Turing Machines. *Theoretical Computer Science*, 168 (1996), 215–240.

217. A. Romero-Jiménez, M.J. Pérez-Jiménez: Simulating Turing Machines by P Systems with External Output. *Fundamenta Informaticae*, 49, 1–3 (2002), 273–287.

218. G. Rozenberg, A. Salomaa: *The Mathematical Theory of L Systems.* Academic Press, New York, 1980.

219. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages* (3 volumes). Springer, Berlin, 1997.

220. M.W. Sabelis, C.M. de Jong: Should All Plants Recruit Bodyguards? Conditions for a Polymorphic ESS Synomone Production in Plants. *Oikos*, 53 (1988), 247–252.

221. A. Salomaa: *Formal Languages.* Academic Press, New York, 1973.

222. A. Salomaa: *Public-Key Cryptography.* 2nd enlarged ed., Springer, Berlin, 1996.

223. F. Sancho-Caparrini: *Formal Verification of DNA Computing and Membrane Computing Algorithms.* PhD Thesis, University of Sevilla, 2002.

224. S.G. Singer, G.L. Nicolson: The Fluid Mosaic Model of the Structure of Cell Membranes. *Science*, 175 (1972), 720–731.

225. M. Sipper: Studying Artificial Life Using a Simple, General Cellular Model. *Artificial Life Journal*, 2, 1 (1995), 1–35.

226. G. Siromoney, R. Siromoney, K. Krithivasan: Abstract Families of Matrices and Picture Languages. *Computer Graphics and Image Processing*, 1 (1972), 284–307.

227. P. Sosik, R. Freund: String Rewriting Sequential P Systems and Regulated Rewriting. *Proc. Conf. on Developments in Language Theory*, Vienna, 2001 *Lecture Notes in Computer Science*, 2295 (W. Kuich, G. Rozenberg, A. Salomaa, eds.) Springer, Berlin, 2002, 379–388.

228. P. Speroni di Fenizio: Artificial Chemistries. *Bulletin of the EATCS*, 76 (February 2002), 128–141.

229. Gh. Ştefan: Membrane Computation with Cellular Automata on a Dedicated Hardware. *Workshop on Multiset Processing*, Curtea de Argeş, Romania, August, 2000.

230. Gh. Ştefan: Chaotic Membrane Computation with Cellular Automata. Manuscript, 2002.

231. K.G. Subramanian, D.G. Thomas, M.H. Begum, P.H. Chandra: A Note on Self Crossover and Splicing P Systems. Submitted, 2001.

232. Y. Suzuki, Y. Fujiwara, H. Tanaka, J. Takabayashi: Artificial Life Applications of a Class of P Systems: Abstract Rewriting Systems on Multisets. In [37], 299–346.

233. Y. Suzuki, J. Takabayashi, H. Tanaka: Investigation of an Ecological System by Using an Abstract Rewriting System on Multisets. In *Recent Topics in Mathematical and Computational Linguistics* (C. Martín-Vide, Gh. Păun, eds.), The Publishing House of the Romanian Academy, Bucharest, 2000, 300–309.

234. Y. Suzuki, H. Tanaka: Artificial Life and P Systems. In [33], 265–285.

235. Y. Suzuki, H. Tanaka: On a LISP Implementation of a Class of P Systems. *Romanian Journal of Information Science and Technology*, 3, 2 (2000), 173–186.

236. Y. Suzuki, S. Tsumoto, H. Tanaka: Analysis of Cycles in Symbolic Chemical Systems Based on an Abstract Rewriting System on Multisets. *Artificial Life*, 5 (1996), 522–528.

237. C. Tarba: *Biomembranes, Transport and Cell Energetics.* The Publishing House of the Romanian Academy, Bucharest, 1996 (in Romanian).

238. A.M. Turing: On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, Ser. 2, 42 (1936), 230–265; a correction, 43 (1936), 544–546.

239. J. van Leeuwen: *Handbook of Theoretical Computer Science* (Vols. A, B). Elsevier, Amsterdam, 1990.

240. D. Wood: *Theory of Computation.* Harper and Row, New York, 1987.

241. R.R. Yager: On the Theory of Bags. *International Journal of General Systems*, 13 (1986), 23–37.

242. C. Zandron: *A Model for Molecular Computing: Membrane Systems.* PhD Thesis, Universitá degli Studi di Milano, 2001.

243. C. Zandron, C. Ferretti, G. Mauri: Priorities and Variable Thickness of Membranes in Rewriting P Systems. Manuscript, 2000.

244. C. Zandron, C. Ferretti, G. Mauri: Solving NP-Complete Problems Using P Systems with Active Membranes. In *Unconventional Models of Computation* (I. Antoniou, C.S. Calude, M.J. Dinneen, eds.), Springer, London, 2000, 289–301.

245. C. Zandron, C. Ferretti, G. Mauri: Two Normal Forms for Rewriting P Systems. *Proc. Third Int. Conf. on Universal Machines and Computations*,

Chişinău, Moldova, 2001 (M. Margenstern, Y. Rogozhin, eds.), *Lecture Notes in Computer Science*, 2055, Springer, 2001, 153–164.

246. C. Zandron, C. Ferretti, G. Mauri: Using Membrane Features in P Systems. *Romanian Journal of Information Science and Technology*, 4, 1–2 (2001), 241–257.

247. C. Zandron, G. Mauri, C. Ferretti: Universality and Normal Forms on Membrane Systems. *Proc. Int. Workshop on Grammar Systems 2000* (R. Freund, A. Kelemenova, eds.), Bad Ischl, Austria, July 2000, 61–74.

Index

0L system 24
2-normal form 171
3-colourability 296

action δ 93
action τ 93
active transport 10
alphabet 17
antiport 11, 130, 238, 377
appearance checking 32
artificial life 384
axiom 19

Bar–Hillel lemma 23
Bhopalator 368
bi-stable catalyst 78
binary normal form 34
Boolean circuit 48
Brusselator 384

carrier 153
carrier index 154
carrier protein 10
catalyst 8
catalytic P system 53
CD grammar system 38, 61
cellular division 14
chemical abstract machine 125
Chomsky grammar 19
Chomsky hierarchy 21
Chomsky normal form 22
closure property 21, 232
coding 18
communication graph 239
complexity class 46, 271
computation 3, 55, 90
computation tree 90
concatenation 17, 18
concentration 99
configuration 3, 54, 88
conformon 368
context-free grammar 20
context-sensitive grammar 20
contextual P system 223
cooperative P system 53

decidability problem 46, 72, 329
depth of a membrane structure 85
deterministic finite automaton 26
deterministic L system 24
deterministic P system 127
deterministic Turing machine 27
Dyck language 24, 85, 119

electrical charge 92
elementary membrane 2, 85, 86
empty string 17
endocytosis 12
endoplasmic reticulum 8
energy 368
equivalent grammars 19
ET0L language 25
eukaryotic 7
evolution rule 3, 53, 87
exocytosis 12
extended configuration 335
extended L system 25
extended P system 163
external output 115

finite automaton 26, 148
fluid–mosaic model 8
forbidding condition 187
forbidding context 141

Gamma language 125
gap junction 13
Geffert normal form 23
gemmation 373
generalized P system 354
global rules 170
Golgi apparatus 7, 12
Gram-negative bacteria 376
Gram-positive bacteria 376

grammar scheme 41
grammar system 38

H scheme 43
H system 43
halting computation 3
halting configuration 90
Hamiltonian path problem (HPP) 46, 268, 286, 321

immediate communication 220
independent set 296
inhibitor 104, 106, 181
initial configuration 54, 88
insertion–deletion 226
insertion–deletion system 44
inter-cellular communication 14
inter-region 375

k-closure 296
Kleene closure 18
Kuroda normal form 23

λ-free morphism 18
language 17
left derivative 18
left-linear grammar 20
length of a string 17
length set 17, 21
length-increasing grammar 20
Lindenmayer system 24
linear grammar 20
linearly bounded automaton 28

Marcus contextual grammar 44
matrix grammar 30
membrane 2
membrane creation 301
membrane dissolution 64
membrane division 273
membrane structure 2, 51, 85, 86
minimal linear grammar 23
mirror image 18
mitosis 14
mobile ambient 359
monochromatic triangle 297
morphism 18
multi-tape Turing machine 28
multiset 3, 49

neural-like P systems 249
neuron 15
non-cooperative P system 53
non-synchronized P system 77
non-terminal symbol 19
normal form 22
NP-complete problem 48
nucleolus 7
nucleus 7

object 3
ordered grammar 37
osmosis 10
output membrane 3, 53

$\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ problem 47
P system with active membranes 274
P system with symport/antiport 130
P systems with global rules 127
parallel time 49
Parikh theorem 23
Parikh vector/mapping 17
passive transport 10
Penttonen normal form 23
permeability 93, 159
permitting condition 187
permitting context 141
phagocytosis 12
phospholipid molecule 9
photosynthesis 392
pinocytosis 12
plasma membrane 7
polarization 92
prefix 17
priority 70, 168
programmed grammar 37
projection 18
prokaryotic 7
promoter 104, 106, 181, 229
promoter diversity 107
propagating L system 24
protein channel 10
pumping lemma 23
pure grammar 19

radius 53
random context grammars 38
reachability 335
recursively enumerable language 21
reduced context-free grammar 21
region 2
register machine 29
regular grammar 20
regular language 18
replicated rewriting 199, 320
replication index 199
rewriting P systems 162

rewriting rule/production 19
rewriting system 19
ribosomes 8
right derivative 18
right-linear grammar 20
rule creation 181, 218

satisfiability (SAT) 281, 292, 311, 321
semi-conditional grammar 38
semilinear language 18
semilinear set 18
sentential form 19
simple P system 127
skin membrane 2, 86
splicing 211
splicing P system 211, 236
splicing rule 43
stream X-machine 357
strictly bounded language 121
string 17
strong binary normal form 35
substring 17
successful computation 55
suffix 17
symport 11, 130, 238, 377
synapse 16

T0L system 25
target command 53
terminal symbol 19
tissue-like P system 237
trace 145
transition 3, 54
Turing machine 27, 298
Turing–Church thesis 29
type-0 grammar 20
type-1 grammar 20
type-2 grammar 20
type-3 grammar 20

unary P system 341
uniport 11
universal Turing machine 40
universal type-0 grammar 41

valuation 349
vertex cover 295
vesicle-mediated transport 10

weak coding 18

X-machine 357

Z-binary normal form 35

Natural Computing Series

W.M. Spears: Evolutionary Algorithms. The Role of Mutation and Recombination. XIV, 222 pages, 55 figs., 23 tables. 2000

H.-G. Beyer: The Theory of Evolution Strategies. XIX, 380 pages, 52 figs., 9 tables. 2001

L. Kallel, B. Naudts, A. Rogers (Eds.): Theoretical Aspects of Evolutionary Computing. X, 497 pages. 2001

M. Hirvensalo: Quantum Computing. XI, 190 pages. 2001

G. Păun: Membrane Computing. An Introduction. XI, 429 pages, 37 figs., 5 tables. 2002

A.A. Freitas: Data Mining and Knowledge Discovery with Evolutionary Algorithms. XIV, 264 pages, 74 figs., 10 tables. 2002

H.-P. Schwefel, I. Wegener, K. Weinert (Eds.): Advances in Computational Intelligence. VIII, 325 pages. 2002

A. Ghosh, S. Tsutsui (Eds.): Advances in Evolutionary Computing. Approx. 980 pages. 2002

L.F. Landweber, E. Winfree (Eds.): Evolution as Computation. DIMACS Workshop, Princeton, January 1999. Approx. 300 pages. 2002

M. Amos: Theoretical and Experimental DNA Computation. Approx. 200 pages. 2002