

Artiom Alhazov  
Svetlana Cojocaru  
Marian Gheorghe  
Yurii Rogozhin (Eds.)

# 14th International Conference on Membrane Computing

CMC14, Chişinău, Moldova, August 20-23, 2013

## Proceedings

*The Conference is dedicated to the 50<sup>th</sup>  
anniversary of the Institute of Mathematics  
and Computer Science*

Institute of Mathematics and Computer Science  
Academy of Sciences of Moldova

**CZU 004+519.7(082)**

**I-58**

Copyright © Institute of Mathematics and Computer Science,  
Academy of Sciences of Moldova, 2013.  
All rights reserved.

INSTITUTE OF MATHEMATICS AND COMPUTER SCIENCE  
5, Academiei Str., Chişinău, Republic of Moldova, MD 2028  
Tel: (373 22) 72-59-82, Fax: (373 22) 73-80-27,  
E-mail: [imam \(at\) math.md](mailto:imam@math.md)  
WEB address: <http://www.math.md>

Editors: Dr. Artiomi Alhazov<sup>1</sup>, Prof. Svetlana Cojocaru<sup>1</sup>, Dr. Marian Gheorghe<sup>2</sup>, Prof. Yurii Rogozhin<sup>1</sup>.

<sup>1</sup> Institute of Mathematics and Computer Science  
Academy of Sciences of Moldova, Chişinău, Moldova

<sup>2</sup> Department of Computer Science  
The University of Sheffield, Sheffield, UK

Authors are fully responsible for the content of their papers.

**Descrierea CIP a Camerei Naţionale a Cărţii**

**"International Conference on Membrane Computing"**, conf. intern. (14; 2013; Chişinău). 14th International Conference on Membrane Computing: Proceedings: The Conference is dedicated to the 50th anniversary of the Institute of Mathematics and Computer Science, CMC14, 20-23 aug. 2013, Chisinau, Moldova/ed.: Artiomi Alhazov [et al.]. – Chişinău: Institute of Mathematics and Computer Science, 2013 (Tipogr. "Valinex SRL"). – 333 p.

Antetit.: Inst. of Mathematics and Computer Science, Acad. of Sciences of Moldova. – Referinţe bibliogr. la sfârşitul art. – 50 ex.

ISBN 978-9975-4237-2-4.

004+519.7(082)

**ISBN 978-9975-4237-2-4**

## Preface

This volume contains the papers presented at the 14th International Conference on Membrane Computing, CMC14, (<http://www.math.md/cmc14/>), which took place in Chişinău, Moldova, in the period of August 20–23, 2013.

The CMC series was initiated by Gheorghe Păun as the Workshop on Multiset Processing in the year 2000. Then two workshops on Membrane Computing were organized in Curtea de Argeş, Romania, in 2001 and 2002. A selection of papers of these three meetings were published as volume 2235 of the Lecture Notes in Computer Science series, as a special issue of *Fundamenta Informaticae* (volume 49, numbers 1–3, 2002), and as volume 2597 of *Lecture Notes in Computer Science*, respectively. The next six workshops were organized in Taragona, Spain (in July 2003), Milan, Italy (in June 2004), Vienna, Austria (in July 2005), Leiden, The Netherlands (in July 2006), Thessaloniki, Greece (in June 2007), and Edinburgh, UK (in July 2008), with the proceedings published in *Lecture Notes in Computer Science* as volumes 2933, 3365, 3850, 4361, 4860, and 5391, respectively. The 10th workshop returned to Curtea de Argeş in August 2009 (LNCS volume 5957). From the year 2010, the series of meetings on membrane computing continued as the Conference on Membrane Computing with the 2010, 2011 and 2012 editions held in Jena, Germany (LNCS volume 6501), in Fontainebleau, France (LNCS volume 7184), and Budapest, Hungary (LNCS volume 7762).

A Steering Committee takes care of the continuation of the CMC series which is organized under the auspices of the European Molecular Computing Consortium (EMCC). In 2013, also a regional version of CMC, the Asian Conference on Membrane Computing, ACMC, takes place in Chengdu, China.

The Steering Committee of the CMC series consists of Gabriel Ciobanu (Iaşi, Romania), Erzsébet Csuhaj-Varjú (Budapest, Hungary), Rudolf Freund (Vienna, Austria), Pierluigi Frisco (Edinburgh, UK), Marian Gheorghe (Sheffield, UK) - chair, Vincenzo Manca (Verona, Italy), Maurice Margenstern (Metz, France), Giancarlo Mauri (Milan, Italy), Linqiang Pan (Wuhan, China), Gheorghe Păun (Bucharest, Romania/Seville, Spain), Mario J. Pérez-Jiménez (Seville, Spain), Petr Sosik (Opava, Czech Republic) and Sergey Verlan (Paris, France).

The CMC14 conference was organized by Institute of Mathematics and Computer Science of the Academy of Sciences of Moldova Republic.

The Program Committee invited lectures from Jozef Gruska (Brno, Czech Republic), Gheorghe Păun (Bucharest, Romania/Seville, Spain), Alberto Leporati (Milano, Italy), Marian Gheorghe (Sheffield, UK), Petr Sosik (Opava, Czech Republic) and Sergey Verlan (Paris, France).

In addition to the texts or the abstracts of the invited talks, this volume contains 19 full papers and an extended abstract, each of which was subject to

at least two referee reports. In addition the volume contains 4 short papers in a progress.

The Program Committee of CMC13 consisted of Artiom Alhazov (Chişinău, Moldova) co-chair, Gabriel Ciobanu (Iaşi, Romania), Alexandru Colesnicov (Chişinău, Moldova), Erzsébet Csuhaj-Varjú (Budapest, Hungary), Giuditta Franco (Verona, Italy), Rudolf Freund (Vienna, Austria), Marian Gheorghe (Sheffield, UK) co-chair, Tomas Hinze (Jena, Germany), Florentin Ipate (Iaşi, Romania), Alberto Leporati (Milano, Italy), Vincenzo Manca (Verona, Italy), Maurice Margenstern (Metz, France), Giancarlo Mauri (Milan, Italy), Radu Nicolescu (Auckland, New Zealand), Linqiang Pan (Wuhan, China), Gheorghe Păun (Bucharest, Romania and Seville, Spain), Dario Pescini (Milan, Italy), Agustin Riscos-Núñez (Seville, Spain), Yurii Rogozhin (Chişinău, Moldova) co-chair, Francisco J. Romero-Campero (Seville, Spain), Petr Sosik (Opava, Czech Republic), György Vazsil (Debrecen, Hungary), Sergey Verlan (Paris, France) and Claudio Zandron (Milan, Italy).

The Organizing Committee consisted of Artiom Alhazov, Lyudmila Burtseva, Svetlana Cojocaru, Alexandru Colesnicov, Ludmila Malahov, Olga Popcova and Yurii Rogozhin.

The editors warmly thank the Program Committee, the invited speakers, the authors of the papers, the reviewers, and all the participants, as well as all who contributed to the success of CMC14.

Chişinău, July 2013

Artiom Alhazov  
Svetlana Cojocaru  
Marian Gheorghe  
Yurii Rogozhin

## Contents

### **Preface** 3

### **Invited Papers** 7

|   |    |
|---|----|
| Marian Gheorghe, Florentin Ipate: Kernel P Systems - A Survey . . .                                     | 9  |
| Jozef Gruska: New Vision and Future of Informatics. Extended abstract                                   | 11 |
| Alberto Leporati: Computational Complexity of P Systems with Active<br>Membranes . . . . .              | 15 |
| Gheorghe Păun: Some Open Problems About Catalytic, Numerical,<br>and Spiking Neural P Systems . . . . . | 25 |
| Petr Sosík: Computational Complexity in Membrane Computing: Is-<br>sues and Challenges . . . . .        | 35 |
| Sergey Verlan: Using the Formal Framework for P Systems . . . . .                                       | 37 |

### **Regular Papers** 39

|   |     |
|---|-----|
| Artiom Alhazov, Svetlana Cojocaru, Alexandru Colesnicov, Ludmila<br>Malahov, Mircea Petic: A P System for Annotation of Romanian<br>Affixes . . . . .       | 41  |
| Bogdan Aman, Gabriel Ciobanu: Behavioural Equivalences in Real-<br>Time P Systems . . . . .   | 49  |
| I.T. Banu-Demergian, G. Stefanescu: The Geometric Membrane Struc-<br>ture of Finite Interactive Systems Scenarios . . . . .                                 | 63  |
| Luděk Cienciala, Lucie Ciencialová, Miroslav Langer: Modelling of<br>Surface Runoff using 2D P colonies . . . . .   | 81  |
| Alex Ciobanu, Florentin Ipate: Implementation of P Systems by using<br>Big Data Technologies . . . . .  | 95  |
| Erzsébet Csuhaj-Varjú, György Vaszil: On Counter Machines versus<br>dP Automata . . . . .   | 117 |
| Ciprian Dragomir, Florentin Ipate, Savas Konur, Raluca Lefticaru,<br>Laurentiu Mierla: Model Checking Kernel P Systems . . . . .                            | 131 |
| Rudolf Freund: Purely Catalytic P Systems: Two Catalysts Can Be<br>Sufficient for Computational Completeness . . . . .                                      | 153 |
| Zsolt Gazdag: Solving SAT by P Systems with Active Membranes in<br>Linear Time in the Number of Variables . . . . .   | 167 |
| Nestine Hope S. Hernandez, Richelle Ann B. Juayong, Henry N. Adorna:<br>Solving Hard Problems in Evolution-Communication P systems<br>with Energy . . . . . | 181 |
| Sergiu Ivanov, Sergey Verlan: About One-Sided One-Symbol Insertion-<br>Deletion P Systems . . . . .   | 199 |
| Alberto Leporati, Luca Manzoni, Antonio E. Porreca: Flattening and<br>Simulation of Asynchronous Divisionless P Systems with Active<br>Membranes . . . . .  | 213 |

|   |            |
|---|------------|
| Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, Claudio Zandron: Enzymatic Numerical P Systems Using Elementary Arithmetic Operations . . . . .  | 225        |
| Tamás Mihálydeák, Zoltán Ernő Csajbók, Péter Takács: Communication Rules Working in Generated Membrane Boundaries . . . . .   | 241        |
| Radu Nicolescu, Florentin Ipate, Huiling Wu: Towards High-level P Systems Programming using Complex Objects . . . . .   | 255        |
| Adam Obtułowicz: In Search of a Structure of Fractals by using Membranes as Hyperedges . . . . .  | 277        |
| M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, L. Valencia-Cabrera: The Relevance of the Environment on the Efficiency of Tissue P Systems . . . . .  | 283        |
| <b>Short Papers</b>   | <b>297</b> |
| Bogdan Aman, Gabriel Ciobanu: Expressing Active Membranes by using Priorities, Catalysts and Cooperation . . . . .  | 299        |
| Gabriel Ciobanu, Dragoş Sburlan: Causal Sequences and Indexed Multisets . . . . .   | 303        |
| Henning Fernau, Rudolf Freund, Sergiu Ivanov, Marion Oswald, Markus L. Schmid, K.G. Subramanian: Undecidability and Computational Completeness for P Systems with One- and Two-dimensional Array Insertion and Deletion Rules . . . . . | 309        |
| Miklós Fésüs, György Vaszil: Chemical Programming and Membrane Systems . . . . .  | 313        |
| Rudolf Freund, Marion Oswald, Gheorghe Păun: Catalytic and Purely Catalytic P Systems and P Automata: Control Mechanisms for Obtaining Computational Completeness . . . . .   | 317        |
| Rudolf Freund, Yurii Rogozhin, Sergey Verlan: Computational Completeness with Generating and Accepting P Systems Using Minimal Left and Right Insertion and Deletion . . . . .  | 321        |
| Alberto Leporati, Lyudmila Burtseva: A Quantum Inspired UREM P System for Solving a Linguistic Problem . . . . .  | 325        |
| Petr Sosík: A Catalytic P System with Two Catalysts Generating a Non-Semilinear Set . . . . .   | 329        |
| <b>Author Index</b>   | <b>333</b> |

# Invited Papers





# Kernel P Systems - A Survey

Marian Gheorghe<sup>1</sup> and Florentin Ipate<sup>2</sup>

<sup>1</sup> Department of Computer Science  
University of Sheffield  
Portobello Street, Regent Court, Sheffield, S1 4DP, UK  
`m.gheorghe@sheffield.ac.uk`

<sup>2</sup> Department of Computer Science  
University of Bucarest  
Str Academiei, 14, Bucarest, Romania  
`florentin.ipate@ifsoft.ro`

Different classes of P systems have been introduced and studied for their computational power or for modelling various problems, like solving simple algorithms, NP-complete problems, or various applications in biology, graphics, linguistics etc. More recently various distributed algorithms and problems have been studied with a new variant of P systems. In many cases the specification of the system investigated requires features, constraints or types of behaviour which are not always provided by the model in its initial definition. It helps in many cases to have some flexibility with modelling approaches, especially in the early stages of investigating a problem, as it might simplify the model, shorten associated processes and clarify more complex or unknown aspects of the system. The downside of this is the lack of a coherent and well-defined framework that allows us to analyse, verify and test this behaviour and simulate the system. In this respect the concept of *kernel P system* (*kP system*) has been introduced in order to include the most used concepts from P systems with the aim of modelling a large class of problems and systems.

This new class of P systems use a graph-like structure (so called, *tissue P systems*) with a set of symbols, labels of membranes, and rules of various types. A broad range of strategies to run the rules against the multiset of objects available in each compartment is provided. The rules in each compartment will be of two types: (i) *object processing rules* which transform and transport objects between compartments or exchange objects between compartments and environment and (ii) *structure changing rules* responsible for changing the system's topology. Each rule has a guard resembling activators and inhibitors associated with certain variants of P systems. We consider rewriting and communication rules, membrane division, dissolution, bond creation and destruction.

The kP system discussed in this work represents a low level modelling language which are complex enough to describe various problems in a relatively efficient and adequate way. Its key features will be illustrated by examples. Connections of various other classes of P systems with kernel P systems will be discussed and the efficient way of solving various problems within this context will be presented. Its syntax and informal semantics will be introduced and analysed. Finally, some implementation issues will be identified and references to the case of a specific implementation of the entire kP systems language, allow-

ing simulation and formal verification will be made and a discussion regarding efficient implementations of its backend on a parallel hardware platform will be presented.

The work reported here includes developments undertaken by other groups and future plans for further steps in this field.

**Acknowledgement.** The work of MG and FI was partially supported by the MuVet project, Romanian National Authority for Scientific Research (CNCS – UEFISCDI), grant number PN-II-ID-PCE-2011-3-0688.

# New Vision and Future of Informatics.

## Extended abstract

Jozef Gruska

Faculty of Informatics, Masaryk University  
Brno, Czech Republic

Currently dominating perception of computer science has its origin in a very cleverly written, and much influential, paper of *Newel, Simon and Perlis*, published in Science in 1967, that well captured the perception of the field at that time.

The basic ideas presented in their paper were:

*"Whenever there are phenomena there can be a science dealing with these phenomena. Phenomena breed sciences. Since there are computers, there is computer science. The phenomena surrounding computers are varied, complex and rich."*

Since that time there have been numerous attempts to modernize such a view of computer science. However, such a computer-centric view of computer science still dominates.

There are nowadays a variety of reasons why such a computer-centric view of the field should be seen as very obsolete and actually damaging the development of the field. They will be discussed in details in the talk. Here are some of them.

- An understanding starts to be developed that information processing plays key role both in physical and biological nature. For example, quantum, DNA, molecular information processing.
- Natural sciences are increasingly seen as being to a large extent information processing driven.
- It starts to be clear that in the future any very significant innovation will use advanced informatics tools, methods and paradigms.

All that requires that a much broader and deeper view of the field should be developed.

A new perception of the informatics presented in the talk will see the field as consisting of four much interleaved components: (a) scientific informatics; (b) technological informatics; (c) new methodology; (d) application informatics. As a scientific discipline of a very broad scope and deep nature, Informatics has many goals. Its main task is Main tasks of scientific informatics are to discover, explore and exploit in depth, the laws, limitations, paradigms, concepts, models, theories, phenomena, structures and processes of both natural and virtual information processing worlds.

To achieve its tasks, scientific Informatics concentrates on new, information processing based, understanding of universe, evolution, nature, life (both natural and artificial), brain and mind processes, intelligence, creativity, information

storing, processing and transmission systems and tools, complexity, security, and other basic phenomena of information processing worlds.

One way to illustrate such a broad and deep perception of scientific informatics will be in the talk through presentation and analysis of its grand challenges. The same will be for technological informatics and applied informatics.

Of a key importance for a new perception of informatics is an understanding that informatics, as a symbiosis of a scientific and a technology discipline, develops also basic ingredients of a new, in addition to theory and experiments, third basic methodology for all sciences, technologies and society in general.

This new, informatics-based, methodology provides a new way of thinking and a new language for sciences and technologies, extending the Galilean mathematics-based approach to new heights.

Informatics-driven methodology subsumes and extends the role and improves tools mathematics used to play in advising, guiding and serving other scientific and technology disciplines and society in general.

Power of new methodology will be discussed in the paper in details. Here are only few of the reasoning:

- It brings new dimension to both old methodologies;
- It brings into new heights an enormous power of modeling, simulations and visualisation;
- It utilises an enormous exploratory and discovery power of automata, algorithms and complexity considerations.
- It utilizes enormous discovery and exploratory power of the correctness and truth searching considerations and tools.
- It utilizes an enormous potential that the study of virtual worlds brings for understanding of the real world.

Because of its enormous guiding power for practically all areas of science, technology and the whole society and an enormously powerful tools informatics offers, we can see informatics as a new queen and at the same time a new powerful servant for all of society.

In particular informatics is expected to play the key role in dealing with two main megachallenges of current science, technology and society. Namely:

- **To beat natural human intelligence.** More exactly, to create super-powerful non-biological intelligence and its merge with biological intelligence.
- **To beat natural human death.** More exactly, to increase much longevity for human bodies and to achieve uploading for human minds. In more details, to fight natural death as another disease and to find ways to upload human mind to non-biological substrate.

There starts to be enough reasons to see the above megachallenges as being currently realistic enough. Here are some of them.

- Since information processing keeps developing exponentially we can assume to have soon (2045?) laptops with information processing power and capacity larger than of all human brains.

- Exponential scaling up in genetic and nanotechnologies and AI create a basis for making two megachallenges as already feasible ones.
- Exponential developments of information processing technologies are believed to imply enormous speed up developments in science and technology.
- Tools and efforts to reverse engineering brains keep also developing exponentially and so we can assume to have quite soon ways to simulate functionality of human brains.
- Society starts to put enormous effort to develop genome engineering, to model human brains and minds as well as to vastly extend human longevity.
- A vision starts to be accepted to see the development of superintelligent machines as the next stage of evolution.

Some related food for thoughts.

- There is nothing in biology found yet that indicates the inevitability of death.  
*Richard Feynman*
- It seems probable that once the machine thinking method had started, it will not take long to outstrip our feeble power. They would be able to converse with each other to sharpen their wits. At some stage therefore, we should have to expect machine to take control. *Alan M. Turing*
- Let an ultraintelligent machine be defined as a machine that can far surpass all intellectual activities of any man, however clever. Since the design of machines is one of intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be and "intelligent explosion" and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the last invention that man needs ever make. *I. J. Good, 1965, a British mathematician*
- Since there is a real danger that computers will develop intelligence and take over we urgently need to develop direct connections to brains so that computers can add to human intelligence rather than be in opposition. *Stephen Hawking*



# Computational Complexity of P Systems with Active Membranes

Alberto Leporati

Dipartimento di Informatica, Sistemistica e Comunicazione  
Università degli Studi di Milano-Bicocca  
Viale Sarca 336/14, 20126 Milano, Italy

`alberto.leporati@unimib.it`

**Abstract.** P systems with active membranes constitute a very interesting model of computation, defined in the framework of Membrane Computing. Since their appearance, they have been used to solve computationally difficult problems (usually in the classes **NP** and **PSPACE**), due to their ability to generate an exponential size workspace in a polynomial number of time steps. Several computational complexity techniques have thus been applied to study their computing power and efficiency. During the talk, I will survey some of the techniques and the main results which have been obtained in the last few years by the group of Membrane Computing at the University of Milano-Bicocca (also known as the “Milano Team”), sometimes in collaboration with colleagues from the international Membrane Computing community.

## 1 Introduction

P systems with *active membranes* are a very interesting and stimulating model of computation, defined in the framework of *membrane systems* [9]. They were first introduced in [10] to attack **NP**-complete problems. Since then, they have generated several variants; a general survey of these can be found in chapters 11 and 12 of [11].

In this model of P systems, also the membranes play an active role in the computations: they possess an electrical charge that can inhibit or activate the rules that govern the evolution of the system, and they can also increase exponentially in number via division rules. This latter feature makes them extremely efficient from a computational complexity standpoint: using exponentially many membranes that evolve in parallel, they can be used to solve **NP**-complete and even **PSPACE**-complete problems [22, 1] in polynomial time. Surprisingly, polarizations are not even needed (provided that division rules are powerful enough) to solve these kinds of problems, as shown in [28, 4]. On the other hand, when the ability of dividing membranes is limited the efficiency apparently decreases: the so-called Milano theorem [27] tells us that no **NP**-complete problem can be solved in polynomial time without using division rules, unless  $\mathbf{P} = \mathbf{NP}$  holds.

Needless to say, several computational complexity techniques have been applied to investigate the computational power and efficiency of P systems with

active membranes. During my talk, I will survey some of these techniques as well as the main results which have been obtained in the last few years by the group of Membrane Computing at the University of Milano-Bicocca (the so-called “Milano team”), sometimes in collaboration with colleagues from the international Membrane Computing community.

This extended abstract is organized as follows. In Section 2 I recall the formal definition and operation of P systems with active membranes, as well as the definition of *time* and *space* complexity. Section 3 summarizes the results on the complexity of P systems with active membranes that will be illustrated during the talk.

## 2 P Systems with Active Membranes

We start by recalling the definition of P systems with active membranes that will be used in the rest of this paper (and during the talk as well). For a more formal definition we refer the reader to chapter 12 of [11].

**Definition 1.** A P system with active membranes of initial degree  $d \geq 1$  is a tuple  $\Pi = (\Gamma, \Lambda, \mu, w_1, \dots, w_d, R)$ , where:

- $\Gamma$  is a finite alphabet of symbols (the objects);
- $\Lambda$  is a finite set of labels for the membranes;
- $\mu$  is a membrane structure (i.e., a rooted unordered tree) consisting of  $d$  membranes, enumerated by  $1, \dots, d$ ; furthermore, each membrane is labeled by an element of  $\Lambda$ , not necessarily in a one-to-one way;
- $w_1, \dots, w_d$  are strings over  $\Gamma$ , describing the initial multisets of objects placed in the  $d$  regions of  $\mu$ ;
- $R$  is a finite set of rules.

As usual in Membrane Computing, the membrane structure of a P system is represented symbolically as a string of balanced nested brackets, where each pair of corresponding open/close ones represents an individual membrane. The nesting of brackets corresponds to the ancestor-descendant relation of nodes in the tree; brackets at the same nesting levels can be listed in any order.

Each membrane possesses, besides its label and position in  $\mu$ , another attribute called *electrical charge* (or polarization), which can be either neutral (0), positive (+) or negative (−) and is always neutral before the beginning of the computation.

The rules are of the following kinds:

- *Object evolution rules*, of the form  $[a \rightarrow w]_h^\alpha$   
They can be applied inside a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the object  $a$  is rewritten into the multiset  $w$  (i.e.,  $a$  is removed from the multiset in  $h$  and replaced by every object in  $w$ ).



- *Send-in communication rules*, of the form  $a [ ]_h^\alpha \rightarrow [b]_h^\beta$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and such that the external region contains an occurrence of the object  $a$ ; the object  $a$  is sent into  $h$  becoming  $b$  and, simultaneously, the charge of  $h$  is changed to  $\beta$ .
- *Send-out communication rules*, of the form  $[a]_h^\alpha \rightarrow [ ]_h^\beta b$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the object  $a$  is sent out from  $h$  to the outside region becoming  $b$  and, simultaneously, the charge of  $h$  is changed to  $\beta$ .
- *Dissolution rules*, of the form  $[a]_h^\alpha \rightarrow b$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the membrane  $h$  is dissolved and its contents are left in the surrounding region unaltered, except that an occurrence of  $a$  becomes  $b$ .
- *Elementary division rules*, of the form  $[a]_h^\alpha \rightarrow [b]_h^\beta [c]_h^\gamma$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$ , containing an occurrence of the object  $a$  but having no other membrane inside (an *elementary membrane*); the membrane is divided into two membranes having label  $h$  and charges  $\beta$  and  $\gamma$ ; the object  $a$  is replaced, respectively, by  $b$  and  $c$  while the other objects in the initial multiset are copied to both membranes.
- *Nonelementary division rules*, of the form

$$[[ ]_{h_1}^+ \cdots [ ]_{h_k}^+ [ ]_{h_{k+1}}^- \cdots [ ]_{h_n}^- ]_h^\alpha \rightarrow [[ ]_{h_1}^\delta \cdots [ ]_{h_k}^\delta ]_h^\beta [[ ]_{h_{k+1}}^\epsilon \cdots [ ]_{h_n}^\epsilon ]_h^\gamma$$

They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$ , containing the positively charged membranes  $h_1, \dots, h_k$ , the negatively charged membranes  $h_{k+1}, \dots, h_n$ , and possibly some neutral membranes. The membrane  $h$  is divided into two copies having charge  $\beta$  and  $\gamma$ , respectively; the positive children are placed inside the former, their charge changed to  $\delta$ , while the negative ones are placed inside the latter, their charges changed to  $\epsilon$ . Any neutral membrane inside  $h$  is duplicated and placed inside both copies.

Each instantaneous configuration of a P system with active membranes is described by the current membrane structure, including the electrical charges, together with the multisets located in the corresponding regions. A computation step changes the current configuration according to the following set of principles:

- Each object and membrane can be subject to at most one rule per step, except for object evolution rules (inside each membrane any number of evolution rules can be applied simultaneously).
- The application of rules is *maximally parallel*: each object appearing on the left-hand side of evolution, communication, dissolution or elementary division must be subject to exactly one of them (unless the current charge of the membrane prohibits it). The same reasoning applies to each membrane

that can be involved to communication, dissolution, elementary or nonelementary division rules. In other words, the only objects and membranes that do not evolve are those associated with no rule, or only to rules that are not applicable due to the electrical charges.

- When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step.
- While all the chosen rules are considered to be applied simultaneously during each computation step, they are logically applied in a bottom-up fashion: first, all evolution rules are applied to the elementary membranes, then all communication, dissolution and division rules; then we proceed towards the root of the membrane structure. In other words, each membrane evolves only after its internal configuration has been updated.
- The outermost membrane cannot be divided or dissolved, and any object sent out from it cannot re-enter the system again.

A *halting computation* of a P system is a finite sequence of configurations  $\mathcal{C} = (C_0, \dots, C_k)$ , where  $C_0$  is the initial configuration, every  $C_{i+1}$  is reachable by  $C_i$  via a single computation step, and no rules can be applied anymore in  $C_k$ . A *non-halting computation* consists of infinitely many successive configurations  $\mathcal{C} = (C_i : i \in \mathbb{N})$ .

P systems can be used as *recognisers* by employing two specified objects YES and NO; exactly one of these must be sent out from the outermost membrane during each computation, in order to signal acceptance or rejection respectively; we also assume that all computations are halting. If all computations starting from the same initial configuration are accepting, or all are rejecting, the P system is said to be *confluent*. If this is not necessarily the case, we have a *non-confluent* P system, and the overall result is established as for nondeterministic Turing machines: it is acceptance iff an accepting computation exists.

In order to solve decision problems (i.e., decide languages), we use *families* of recogniser P systems  $\Pi = \{\Pi_x : x \in \Sigma^*\}$  for some finite alphabet  $\Sigma$ . Each input  $x$  is associated with a P system  $\Pi_x$  that decides the membership of  $x$  in the language  $L \subseteq \Sigma^*$  by accepting or rejecting. The mapping  $x \mapsto \Pi_x$  is restricted, in order to be computable efficiently; usually one of the following *uniformity conditions* is imposed.

**Definition 2.** A family of P systems  $\Pi = \{\Pi_x : x \in \Sigma^*\}$  is said to be *semi-uniform* if the mapping  $x \mapsto \Pi_x$  can be computed in polynomial time by a deterministic Turing machine.

The Turing machine can encode its output  $\Pi_x$  by describing the membrane structure with brackets, the multisets as strings of symbols (in unary notation) and listing the rules one by one. However, any explicit encoding of  $\Pi_x$  is allowed as output, as long as the number of membranes and objects represented by it does not exceed the length of the whole description, and the rules are listed one by one. We pose this restriction in order to enforce the initial membranes, initial

objects and rules to be at most polynomial in number, as they can be super-polynomial if more compact representations (e.g., binary numbers) are used; this mimics a (hypothetical) realistic process of construction of the P systems, where membranes and objects are presumably placed one by one, and require actual physical space in proportion to their number (see also how the size of a configuration is defined in the following, and [7]).

**Definition 3.** A family of P systems  $\Pi = \{\Pi_x : x \in \Sigma^*\}$  is said to be uniform if the mapping  $x \mapsto \Pi_x$  can be computed by two deterministic polynomial-time Turing machines  $M_1$  and  $M_2$  as follows:

- The machine  $M_1$ , taking as input the length  $n$  of  $x$  in unary notation, constructs a P system  $\Pi_n$  with a distinguished input membrane (the P system  $\Pi_n$  is common for all inputs of length  $n$ ).
- The machine  $M_2$ , on input  $x$ , outputs a multiset  $w_x$  (an encoding of the specific input  $x$ ).
- Finally,  $\Pi_x$  is simply  $\Pi_n$  with  $w_x$  added to the multiset placed inside its input membrane.

Notice how the uniform construction is just a restricted case of semi-uniform construction. The relations between the two kinds of uniformity have not completely been clarified yet; see [11, 7] for further details on uniformity conditions (including even weaker constructions).

Finally, we describe how time and space complexities for families of recogniser P systems are measured.

**Definition 4.** A uniform or semi-uniform family of P systems  $\Pi = \{\Pi_x : x \in \Sigma^*\}$  is said to decide the language  $L \subseteq \Sigma^*$  in time  $f: \mathbb{N} \rightarrow \mathbb{N}$  iff, for each  $x \in \Sigma^*$ ,

- the system  $\Pi_x$  accepts if  $x \in L$ , and rejects if  $x \notin L$ ;
- each computation of  $\Pi_x$  halts within  $f(|x|)$  computation steps.

The notion of space complexity has been formally introduced in the Membrane Computing setting in [14], in order to analyse the time/space trade-off that is common when solving computationally hard problems via P systems. The size  $|\mathcal{C}|$  of a configuration  $\mathcal{C}$  of a P system is given by the sum of the number of objects and the number of membranes; this definition assumes that every component of the system requires some fixed amount of physical space, thus approximating (up to a polynomial) the size of a real cell. The space required by a halting computation  $\mathcal{C} = (\mathcal{C}_0, \dots, \mathcal{C}_k)$  is then given by  $|\mathcal{C}| = \max\{|\mathcal{C}_0|, \dots, |\mathcal{C}_k|\}$ , and the space required by a P system  $\Pi$  is

$$|\Pi| = \max\{|\mathcal{C}| : \mathcal{C} \text{ is a computation of } \Pi\}.$$

We can finally give the following definition.

**Definition 5.** A uniform or semi-uniform family of P systems  $\Pi = \{\Pi_x : x \in \Sigma^*\}$  operates in space  $f: \mathbb{N} \rightarrow \mathbb{N}$  if  $|\Pi_x| \leq f(|x|)$  for all  $x \in \Sigma^*$ .

Several complexity classes can be defined referring to the languages recognized by P systems with active membranes (possibly with restrictions on their rules), when a polynomial, exponential, logarithmic (or other) bound is fixed on the amount of time or space allowed in computations. Here we do not recall them, so as not to burden the exposition. For precise definitions, we refer the reader to the cited papers.

### 3 The Complexity of P Systems with Active Membranes

During the talk I will focus on techniques and results concerning the time and space complexity of P systems with active membranes, that is, the amount of time and/or space needed to solve a given problem. The leading question will be: “When we bound the amount of time and/or space by this quantity, what is the class of decision problems (resp., languages) we can solve (resp., recognize)?”.

Referring to [26], I will first show that a deterministic single-tape Turing machine, operating in polynomial space with respect to the input length, can be efficiently simulated (both in terms of time and space) by a semi-uniform family of P systems with active membranes and three polarizations, using only communication rules. Basing upon this simulation, a result similar to the *space hierarchy theorem* [21] can be obtained for P systems with active membranes: the larger the amount of space we can use during the computations, the harder the problems we are able to solve.

We then continue to consider the case in which only communication rules and nonelementary division rules (which apply to membranes containing other membranes) are allowed. It turns out that the resulting P systems are not computationally universal, neither in the uniform nor in the semi-uniform setting; nonetheless, they are very powerful, as they characterize the class of languages decidable by Turing machines using time (or, equivalently, space) bounded by an exponential function iterated polynomially many times (known as *tetration*) [13].

The computing power of polynomial-time P systems with division rules operating only on *elementary* membranes (that is, membranes not containing other membranes) is possibly the most interesting case. It is a known fact that elementary division rules suffice to efficiently solve **NP**-complete problems (and, due to closure under complement, also **coNP**-complete ones). This result dates back to 2000 in the semi-uniform case [27], and to 2003 in the uniform case [12]. Since these results do not require membrane dissolution rules, they hold also for the so-called *P systems with restricted elementary active membranes* [1], where dissolution is avoided. Although a **PSPACE** upper bound was proved in 2007 [23], no significant improvement on the  $\mathbf{NP} \cup \mathbf{coNP}$  lower bound for these P systems has been found until 2010. Following [16] and [17], I will show that there exists a uniform family of P systems with restricted elementary active membranes that solves the **PP**-complete problem **SQRT-3SAT**: given a Boolean formula of  $m$  variables in 3CNF, do at least  $\sqrt{2^m}$  among the  $2^m$  possible truth assignments satisfy it? The ability to solve all decision problems in the complexity class **PP** will follow by an analogous solution of the **NP**-hard problem **THRESHOLD-3SAT**:

Given a Boolean formula of  $m$  variables in 3CNF, and a non-negative integer  $k < 2^m$ , do more than  $k$  assignments (out of  $2^m$ ) satisfy it? Note that the complexity class **PP** appears to be larger than **NP**, since it contains **NP** as a subset and it is closed under complement: thus  $\mathbf{NP} \cup \mathbf{coNP} \subseteq \mathbf{PP}$ . However, neither the upper bound proved in [23] nor the lower bound proved in [16] are known to be strict.

The existence of the uniform family of P systems with restricted elementary active membranes shown in [16, 17] has an interesting consequence. As shown in [19], it is possible to use the P systems that solve THRESHOLD-3SAT (presented in [17]) as modules inside a larger P system; this allows us to simulate subroutines or oracles. In this way, the class **PPP** turns out to be solvable in polynomial time by P systems, without requiring nonelementary division or dissolution rules. This result, together with Toda's theorem [25], allows us to conclude that P systems with restricted elementary active membranes are able to solve all the decision problems residing in the *polynomial hierarchy* **PH** [24].

I will then pass to consider the characterization of P systems with active membranes occurring in *polynomial*, *exponential* and *logarithmic* space.

Concerning polynomial space, we will focus our attention on recognizer P systems with active membranes (that, in this context, means associating three polarizations to the membranes, whereas division and dissolution rules are forbidden). Following [15], I will show that these P systems are able to efficiently simulate deterministic register machines, using only communication and evolution rules. Such a simulation will then be used to illustrate the following result: recognizer P systems with active membranes are able to solve, in a uniform way, the **PSPACE**-complete problem QUANTIFIED-3SAT, using a polynomial amount of space (and an arbitrary amount of time — in a sense, we are here trading time for space). This means that the complexity class **PSPACE** is contained into the class of decision problems which can be solved in polynomial space by the above kind of recognizer P systems; furthermore, such P systems can solve in arbitrary time (and polynomial space) problems which cannot be solved in polynomial time unless  $\mathbf{P} = \mathbf{PSPACE}$ . On the contrary, in [18] it has been proved that P systems with active membranes can be simulated by Turing machines with only a polynomial increase in space complexity. By combining this result with the above stated ability of P systems to solve **PSPACE**-complete problems in polynomial space, we obtain a *characterization* of **PSPACE** in terms of membrane systems. An interesting aspect is that this result holds for both confluent and non-confluent systems, and even when strong features such as division rules are used.

A similar result can be obtained for P systems with active membranes working in *exponential* space. In particular, in [2] it is shown that exponential-space P systems with active membranes *characterise* the complexity class **EXSPACE**. This result is proved by simulating Turing machines working in exponential space via uniform families of P systems with restricted elementary active membranes; the simulation is efficient, in the sense that the time and space required are at most polynomial with respect to the resources employed by the simulated Turing

machine. In fact, it should be noted that the simulation technique used in [18] does not seem to be applicable when the space bound is exponential (or even super-exponential). Indeed, we would need to use P systems with an exponential number of membranes with distinct labels, and such systems cannot be built in a polynomial number of steps by a deterministic Turing machine — as required by the notion of polynomial-time uniformity usually employed in the literature [12].

Finally, investigation on P systems with active membranes working in *logarithmic* space is currently in progress. Here a new notion of uniformity is needed, which is weaker than the P systems themselves, otherwise one could cheat by letting the Turing machine that builds the P systems solve the problem directly. Inspired by Boolean circuits complexity [6] we have thus introduced **DLOGTIME**-uniformity [20], and we have proved that **DLOGTIME**-uniform families of P systems with active membranes working in logarithmic space (not counting their input) can simulate logarithmic-space deterministic Turing machines. It remains to be established whether these P systems may or not characterize the class **L** of problems solvable in logarithmic space by deterministic Turing machines, or maybe solve harder problems like, for instance, those in **NL**.

**Acknowledgements** I warmly thank the organizers of the 14<sup>th</sup> International Conference on Membrane Computing for inviting me. The research here described was partially supported by Università degli Studi di Milano-Bicocca, Fondo di Ateneo (FA) 2011.

## References

1. Alhazov, A., Martín-Vide, C., Pan, L.: Solving a PSPACE-complete problem by recognizing P systems with restricted active membranes. *Fundamenta Informaticae* 58(2), 67–77 (2003)
2. Alhazov, A., Leporati, A., Mauri, G., Porreca, A.E., Zandron, C.: The computational power of exponential-space P systems with active membranes. In: Martínez-del-Amor, M.A. et al (eds.), *Proceedings of the Tenth Brainstorming Week on Membrane Computing*, Sevilla, January 30 – February 3, 2012, Volume I, pp. 35–60. Research Group on Natural Computing, Seville (2012)
3. Garey, M.R., Johnson, D.S.: *Computers and intractability: a guide to the theory of NP-completeness*. W.H. Freeman & Co. (1979)
4. Leporati, A., Ferretti, C., Mauri, G., Pérez-Jiménez, M.J., Zandron, C.: Complexity aspects of polarizationless membrane systems. *Natural Computing* 8, 703–717 (2009)
5. Mauri, G., Leporati, A., Porreca, A.E., Zandron, C.: Recent complexity-theoretic results on P systems with active membranes. *Journal of Logic and Computation*, online first, doi: 10.1093/logcom/exs077 (2013)
6. Mix Barrington, D.A., Immerman, N., Straubing, H.: On uniformity within  $NC^1$ . *Journal of Computer and System Sciences* 41(3), 274–306 (1990)
7. Murphy N., Woods D.: The computational power of membrane systems under tight uniformity conditions. *Natural Computing* 10(1), 613–632 (2011)
8. Papadimitriou, C.H.: *Computational complexity*. Addison-Wesley (1993)

9. Păun, Gh.: Computing with membranes. *Journal of Computer and System Sciences* 1(61), 108–143 (2000)
10. Păun, Gh.: P systems with active membranes: attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics* 6(1), 75–90 (2001)
11. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): *The Oxford handbook of membrane computing*. Oxford University Press (2010)
12. Pérez-Jiménez, M.J., Romero Jiménez, A., Sancho Caparrini, F.: Complexity classes in models of cellular computing with membranes. *Natural Computing* 2(3), 265–285 (2003)
13. Porreca, A.E., Leporati, A., Zandron, C.: On a powerful class of non-universal P systems with active membranes. In: Gao, Y. et al. (eds.), *Developments in Language Theory, DLT 2010, London, ON, Canada, August 17–20, 2010. Lecture Notes in Computer Science*, vol. 6224, pp. 364–375. Springer (2010)
14. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: Introducing a space complexity measure for P systems. *International Journal of Computers, Communications & Control* 4(3), 301–310 (2009)
15. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: P systems with active membranes: trading time for space. *Natural Computing* 10(1), 167–182 (2011)
16. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: P systems with elementary active membranes: beyond NP and coNP. In: Gheorghe, M. et al. (eds.), *Membrane Computing, Eleventh International Conference, CMC 2010, Jena, Germany, August 24–27, 2010, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 6501, pp. 383–392. Springer (2010)
17. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: Elementary active membranes have the power of counting. *International Journal of Natural Computing Research* 2(3), 35–48 (2011)
18. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: P systems with active membranes working in polynomial space. *International Journal of Foundations of Computer Science* 2(1), 65–73 (2011)
19. Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: P systems simulating oracle computations. In: Gheorghe M. et al. (eds.), *Membrane Computing: Twelfth International Conference, CMC 2011, Fontainebleau, France, August 23–26, 2011, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 7184, pp. 346–358. Springer (2012)
20. Porreca, A.E., Zandron, C., Leporati, A., Mauri, G.: Sublinear-space P systems with active membranes. In: Csuhaj-Varjú E. et al. (eds.), *Membrane Computing: Thirteenth International Conference, CMC 2012, Budapest, Hungary, August 28–31, 2012, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 7762, pp. 342–357. Springer (2012)
21. Sipser M.: *Introduction to the theory of computation*. Third edition. Cengage Learning (2012)
22. Sosík, P.: The computational power of cell division in P systems: beating down parallel computers? *Natural Computing* 2(3), 287–298 (2003)
23. Sosík, P., Rodríguez-Patón, A.: Membrane computing and complexity theory: a characterization of PSPACE. *Journal of Computer and System Sciences* 73(1), 137–152 (2007)
24. Stockmeyer, L.J.: The polynomial hierarchy. *Theoretical Computer Science* 3, 1–22 (1976)
25. Toda, S.: PP is as hard as the polynomial-time hierarchy. *SIAM Journal on Computing* 20(5), 865–877 (1991)

26. Valsecchi, A., Porreca, A.E., Leporati, A., Mauri, G., Zandron, C.: An efficient simulation of polynomial-space Turing machines by P systems with active membranes. In: Păun, Gh. et al (eds.) *Membrane Computing, 10<sup>th</sup> International Workshop, WMC 10, Revised Selected and Invited Papers*. Lecture Notes in Computer Science, vol. 5957, pp. 461–478. Springer (2010)
27. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-complete problems using P systems with active membranes. In: Antoniou, I. et al. (eds.), *Unconventional Models of Computation, UMC2K: Proceedings of the Second International Conference*, pp. 289–301. Discrete Mathematics and Theoretical Computer Science. Springer (2001)
28. Zandron, C., Leporati, A., Ferretti, C., Mauri, G., Pérez-Jiménez, M.J.: On the Computational Efficiency of Polarizationless Recognizer P Systems with Strong Division and Dissolution. *Fundamenta Informaticae* 87(1), 79–91 (2008)



# Some Open Problems About Catalytic, Numerical, and Spiking Neural P Systems

Gheorghe Păun

Institute of Mathematics of the Romanian Academy  
PO Box 1-764, 014700 București, Romania, and

Department of Computer Science and Artificial Intelligence  
University of Sevilla

Avda. Reina Mercedes s/n, 41012 Sevilla, Spain  
gpaun@us.es, ghpaun@gmail.com

**Abstract.** Some open problems and research topics are pointed, about three classes of P systems: catalytic, numerical, and spiking neural P systems. In each case, several issues are briefly discussed, in general, related to questions already formulated as open problems in the literature and also related to recent results dealing with these questions. With respect to spiking neural P systems, a new variant is proposed, systems with the evolution rules associated with the synapses, not with the neurons; the universality of this new class of SN P systems is proved.

## 1 Introduction

In spite of the large bibliography accumulated in the almost fifteen years since this research area was initiated, [26], membrane computing still exhibits a lot of open problems and research topics, some of them “going back to basics”, others being related to more recent branches of the theory. We recall here three sets of such problems, from both categories mentioned above.

First, we start from the already “classic” question whether or not catalytic P systems with one catalyst, or purely catalytic P systems with two catalysts are computationally universal, and we add to this basic issue three related research topics: (i) give an example of a P system with two catalysts computing a non-trivial (e.g., non-semi-linear) set of numbers, find additional features to be added to (ii) P systems with one catalyst or to (iii) purely catalytic P systems with two catalysts in order to get universality. Recent results in this respect were reported – see, e.g., [11], [8].

Then, we consider the numerical P systems. Besides the basic question, of constructing a complexity theory for these systems, especially related to and important for applications, open problems related to a recent important progress concerning the power of enzymatic numerical P systems ([20]) are formulated. In particular, an interesting question concerns of the computing power of numerical P systems with a small number of enzymes.

Finally, with respect to the spiking neural (SN) P systems, we mention two problems already mentioned elsewhere (e.g., in [15]) and we also introduce a new

class of SN P systems, where the spiking and the forgetting rules are associated with the synapses, not with the neurons. The universality of this class of SN P systems is proved – and the move of rules on synapses seems to be useful, the proof is simpler than in the case of rules placed in neurons.

In view of the assumed non-Turing computing power/behavior of the brain, an interesting issue would be to find SN P systems able to compute beyond Turing barrier; suggestions from the hypercomputation area could be useful.

The reader is assumed to be familiar with membrane computing (e.g., from [28], [33], [40]), hence we recall no prerequisites. Instead, for the use of the reader, we indicate a series of primary references, without being complete from this point of view; further references can be found in the above mentioned comprehensive sources of information in membrane computing.

## 2 Catalytic P Systems

P systems with catalytic rules were already introduced in [26], with their computing power left open.

We denote by  $NP_m(cat_r)$  the family of sets of numbers computed (generated) by P systems with at most  $m$  membranes, using catalytic or non-cooperative rules, containing at most  $r$  catalysts. When all the rules of a system are catalytic, we say that the system is *purely catalytic*, and the corresponding families of sets of numbers are denoted by  $NP_m(pcat_r)$ . When the number of membranes is not bounded by a specified  $m$  (it can be arbitrarily large), then the subscript  $m$  is replaced with  $*$ .

The following fundamental results are known:

**Theorem 1.** (i)  $NP_2(cat_2) = NRE$ , [10];  
(ii)  $NREG = NP_*(pcat_1) \subseteq NP_*(pcat_2) \subseteq NP_2(pcat_3) = NRE$ , [13], [14].

Two intriguing open problems appear here, related to the borderline between universality and non-universality: (1) are catalytic P systems with only one catalyst universal? (2) are purely catalytic P systems with two catalysts universal? The conjecture is that both these questions have a negative answer, but it is also felt that “one catalyst is almost universal”: many features which look “innocent” at the first sight are enough to lead P systems with one catalyst to universality (see [11]) – and similar results were obtained also for purely catalytic P systems with two catalysts (see [8]).

Here we briefly recall the universality results for one catalyst P systems with additional ingredients:

- Introducing a *priority* relation among rules, [26].
- Using *promoters* and *inhibitors* associated with the rules.
- Controlling the computation by means of controlling the *membrane permeability*, by actions  $\delta$  (decreasing the permeability) and  $\tau$  (increasing the permeability), [27].
- Besides catalytic and non-cooperating rules, also using rules for *membrane creation*, [22].

- Considering, instead of usual catalysts, *bi-stable catalysts*, [34], or *mobile catalysts*, [18].
- Imposing *target restrictions* on the used rules, [11]; the universality was obtained for P systems with 7 membranes, and it is an open problem whether or not the number of membranes can be diminished).
- Imposing to P systems the idea from *time-varying* grammars and splicing systems, [11]; the universality of time-varying P systems is obtained for one catalyst P systems with only one membrane, having the period equal to 6, and it is open the question whether the period can be decreased.
- Using in a transition only (labeled) rules with the same label – so-called *label restricted* P systems, [19].

Several of these results were extended in [8] to purely catalytic P systems with two catalysts. It remains open to do this for all the previous results, as well as to look for further ingredients which, added to one catalyst P systems or to purely catalytic P systems with two catalysts, can lead to universality. It would be interesting to find such ingredients which work for one catalyst systems and not for purely catalytic systems with two catalysts, and conversely. Suggestions from the regulated rewriting area [6] or the grammar systems area [3] in formal language theory can be useful.

We end this section with a somewhat surprising issue: we know that  $NP_2(cat_2) = NRE$ , but no example of a P system with two catalysts which generates a non-trivial set of numbers (for instance,  $\{2^n \mid n \geq 1\}, \{n^2 \mid n \geq 1\}$ ) is known. In fact, the problem is to find a system of this kind as simple as possible (otherwise, just repeating the construction in the proof from [10], starting from a register machine computing a set as above, we get an example, but of a large size).<sup>1</sup>

### 3 Numerical P Systems

Numerical P systems form an “eccentric” class of P systems, because of their “non-syntactic” character, far from language and automata theory, but closer to numerical analysis. This is probably one of the reasons for which only a few papers were accumulated in this area. However, because of the economic motivations, [31], and of the recent applications in robot control, [24], [25], [38], [37], the subject started to call the attention. There are many questions to be investigated in this framework (see a list of such questions in [30]).

Two recent papers, making important steps ahead in the study of numerical P systems are [20] and [21]. The first paper considerably improves the universality results for enzymatic numerical P systems. We do not recall here the definitions, but we only mention that one deals with enzymatic numerical P systems working in the so-called *all-parallel* or *one-parallel* modes introduced in [39].

---

<sup>1</sup> Petr Sosík told me recently that he is progressing in finding such a simple/readable example.

Thus, two immediate questions are to consider also the case of (i) numerical P systems without enzymes and (ii) of sequential numerical P systems (with or without enzymes).

Then, let us remember that the enzyme variables behave like catalysts (actually, they are closer to promoters) in catalytic P systems. This suggests the problem of considering numerical P systems with a small number of enzymes. Which is the smallest number of enzyme variables for which enzymatic numerical P systems (working in a specified manner: sequential, all-parallel, or one-parallel) is universal?

## 4 Spiking Neural P Systems

The SN P systems area contains many open problems and research topics. We have mentioned in the Introduction the paper [15]. Three main problems are recalled there:

- To further investigate the power and the properties of SN dP systems, that is, to combine the idea of distributed P systems introduced in [6] with that of spiking neural P systems from [17]. SN dP systems were introduced in [16], but only briefly investigated.
- To investigate the possibility of using SN P systems as pattern recognition devices, in general, in handling 2D patterns. One of the ideas is to consider a layer of input neurons which can read an array line by line and the array is recognized if and only if the computation halts.
- In some sense, the SN P systems is the only class of P systems which have only a few and somewhat metaphoric applications in the study of the “real” brain, of interest for biologists, and this is an important issue: should we change the definition of an SN P system in order to have such applications, or we simply failed to find them in the present setup?

Actually, several modifications in the initial definition of SN P systems were considered already. We only mention the SN P systems with astrocytes ([29], [1]), the SN P systems with *request rules* ([5]), the SN P systems with anti-spikes ([23]), and the axon computing systems ([4]).

Here we introduce one further modification in the initial definition, namely, we move the firing rules (they can be both spiking and forgetting rules, of the standard forms in SN P systems) on the synapses. The neurons contain spikes; when the number of spikes in a given neuron is “recognized” by a rule on a synapse leaving from that neuron, then the rule is fired, a number of spikes are consumed and a number of spikes are sent to the neuron at the end of the synapse. Precise details will be given immediately. Using one rule per synapse, with all synapses firing in parallel, we get computations, in the usual style of SN P systems.

In what follows, we prove the universality of SN P systems with rules on synapses (with the result of a computation being the number of spikes stored in a designated neuron, the output one, in the end of the computation).

Formally, an SN P system with rules on synapses is a construct

$$\Pi = (O, \sigma_1, \dots, \sigma_m, syn, i_o),$$

where  $O = \{a\}$  contains the spike symbol,  $\sigma_i, 1 \leq i \leq m$ , are the neurons, of the form  $\sigma_i = (n_i)$  (the number of spikes initially present in neuron  $\sigma_i$ ),  $i_o$  is the output neuron (usually labeled by *out*), and  $syn$  is the set of synapses, i.e., pairs of the form  $((i, j), R_{(i,j)})$ , where  $i, j \in \{1, 2, \dots, m\}$ , with  $i \neq j$ , and  $R_{(i,j)}$  is a finite set of rules of the following forms:

1.  $E/a^c \rightarrow a^p; d$ , with  $E$  a regular expression over  $O, c \geq p \geq 1, d \geq 0$ ;
2.  $a^s \rightarrow \lambda$ , for  $s \geq 1$ .

The rules of the first form are spiking rules: if  $E/a^c \rightarrow a^p; d \in R_{(i,j)}$  and the neuron  $\sigma_i$  contains  $k$  spikes such that  $a^k \in L(E), k \geq c$ , then the rule is enabled,  $k$  spikes from  $\sigma_i$  are consumed and  $p$  spikes are sent to neuron  $\sigma_j$  after a delay of  $d$  steps. The rules of the latter form are usual forgetting rules;  $a^s \rightarrow \lambda \in R_{(i,j)}$  is applied only if  $\sigma_i$  contains  $s$  spikes, which are removed by the use of this rule.

We start from the initial configuration,  $(n_1, n_2, \dots, n_m)$ . As usual in the SN P area, we work sequentially on each synapse (at most one rule from each set  $R_{(i,j)}$  can be used), in parallel at the level of the system (if a synapse has at least one rule enabled, then it has to use a rule).

A delicate problem appears when several synapses starting in the same neuron have rules which can be applied. We work here with the restriction that all rules which are applied consume the same number of spikes from the given neuron. Let us assume that the applied rules on the synapses leaving from  $\sigma_i$  are of the form  $E_u/a^c \rightarrow a^{p_u}; d_u$ ; then  $c$  spikes are removed from  $\sigma_i$  (and not a multiple of  $c$ , according to the number of applied rules). Of course, this restriction can be replaced by another strategy: various rules can consume various numbers of spikes and in this way the sum of these numbers of spikes is removed from the neuron.

Actually, we choose this restriction because this is the case in the proof below, where, furthermore, no delay and no forgetting rule is used.

We denote by  $NsSN_m^n P$  the family of sets of numbers computed (generated) by SN P systems with at most  $m$  neurons and at most  $n$  rules associated with a synapse; as usual, the indices  $n, m$  are replaced with  $*$  when no bound is imposed on the respective parameter.

**Theorem 2.**  $NsSN_*^2 P = NRE$ .

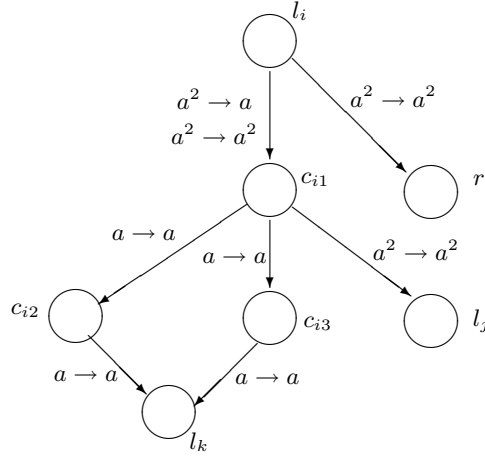
*Proof.* We only prove the inclusion  $NRE \subseteq NsSN_*^2 P$  and to this aim we construct an SN P system with rules on synapses which simulates a register machine  $M = (m, H, l_0, l_h, I)$  (number of registers, set of instruction labels, initial label, halt label, set of instructions). We assume that register 1, the one where the result is obtained, is never decremented.

As usual, for each instruction of  $M$  we construct a module of our SN P system  $\Pi$ . With each register of  $M$  we associate a neuron; if register  $r$  contains the

number  $n$ , then this is encoded in the associated neuron  $\sigma_r$  by means of  $2n$  spikes. With each  $l \in H$  we also associate a neuron  $\sigma_l$ . Further neurons are present in the modules below. There are two distinguished neurons,  $\sigma_g$  (a “garbage collector”) and  $\sigma_{out}$  (the output neuron). Initially, all neurons are empty, with the exception of  $\sigma_{l_0}$ , where we place two spikes. In general, a neuron  $\sigma_l, l \in H$ , is active if it gets two spikes; rules on the synapses leaving  $\sigma_l$  can then be used. When some neuron  $\sigma_l$  is active, then the instruction labeled with  $l$  starts to be simulated.

Here are the mentioned modules (we give them in the graphical form, with the obvious meaning: neurons are represented by circles, with the number of spikes specified inside, and the synapses have the rules written near them):

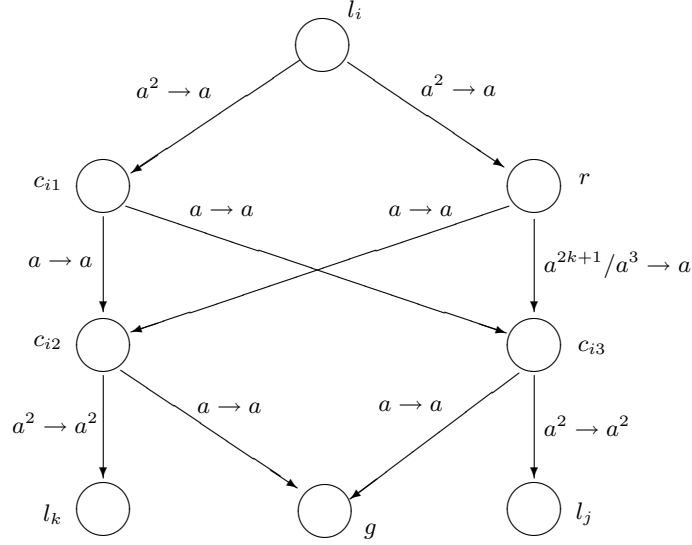
The module associated with an ADD instruction  $l_i : (\text{ADD}(r), l_j, l_k)$  is given in Figure 1. It works as follows. After introducing two spikes in  $\sigma_{l_i}$ , both synapses leaving this neuron fire. One of them sends two spikes to  $\sigma_r$  (and this corresponds to increasing by one the value of this register), and the other one sends one or two spikes to neuron  $c_{i1}$ . Depending on this number, one of the neurons  $\sigma_{l_j}, \sigma_{l_k}$  gets two spikes and in this way the modules associated with those neurons/labels become active.



**Fig. 1.** The ADD module.

The module associated with a SUB instruction  $l_i : (\text{SUB}(r), l_j, l_k)$  is given in Figure 2.

After activating neuron  $\sigma_{l_i}$ , both  $\sigma_r$  and  $\sigma_{c_{i1}}$  receive one spike. In this way,  $\sigma_r$  contains an odd number of spikes and its synapses can fire. If there is only one spike in  $\sigma_r$  (hence the neuron was empty), then a spike is sent to  $\sigma_{c_{i2}}$ , and  $\sigma_{l_k}$  gets two spikes. If the register  $r$  is not empty, then neuron  $\sigma_{l_j}$  gets two spikes, through neuron  $c_{i3}$  (while 3 spikes are removed from  $\sigma_r$ ). In both cases, the continuation of the simulation of the register machine is correct.



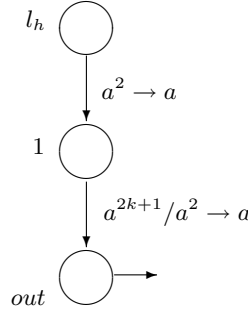
**Fig. 2.** The SUB module.

Note the important detail that if any of the neurons  $c_{i2}, c_{i3}$  receives only one spike, the synapse having associated the rule  $a \rightarrow a$  must be enabled, and in this way the spike is removed (and one spike is added to the “garbage collector”  $\sigma_g$ ). This is useful also in ensuring that the SUB modules do not interfere in an unwanted way: several SUB instructions can send a spike to the same register  $r$ ; if the spike does not come from the neuron  $\sigma_{l_i}$ , then neurons  $c_{i2}, c_{i3}$  will get only one spike, which is immediately moved in the “garbage collector”, hence no neuron  $\sigma_{l_j}, \sigma_{l_k}$  is activated.

The module associated with the HALT instruction  $l_h : \text{HALT}$  is given in Figure 3. If  $\sigma_{l_h}$  receives two spikes, its synapse send one spike to neuron  $\sigma_1$  (which is never decreased during the computation of  $M$ , hence this is the first time when this neuron contains an odd number of spikes). Each pair of spikes will send one spike to neuron  $\sigma_{out}$ , and the process stops when only one spike remains in  $\sigma_1$ . Thus, in the end, the output neuron will contain the number stored in the halting configuration by register 1 of  $M$ .

In conclusion,  $M$  and  $\Pi$  compute the same set of numbers.  $\square$

It is worth mentioning that the maximal number of rules associated with each synapse in the previous proof is two, because of the need of having non-determinism in the functioning of the system. If we will use an SN P system in the accepting mode (start the computation by introducing a number of spikes in a neuron and accept that number if the computation halts), then we can have only one rule associated with each synapse.

**Fig. 3.** The HALT module.

The result of a computation can be also defined as the number of steps elapsed between the first and the second spike sent to the environment by the output neuron (a “pseudo-synapse” should be considered, having rules associated, between the output neuron and the environment); the necessary changes in the HALT module from the previous proof remains as an exercise for the reader.

A natural problem which appears in this framework is to find small universal SN P systems with rules on synapses, a direction of research with many results in terms of usual SN P systems.

We end this section with one further research idea: changing the definition of SN P systems in such a way to obtain hypercomputations, going beyond the Turing barrier. In membrane computing there are, as far as we know, only two papers dealing with this subject (but not with SN P systems), the accelerated P systems with membrane creation from [2], and the lineages of P systems from [35]. Suggestions from the general hypercomputation area could be useful – see, e.g., the survey from [36].

## 5 Final Remarks

We end this note by recalling the attention about the “mega-paper” [12], where a lot of open problems and research topics in membrane computing can be found.

**Acknowledgements.** Work supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200.

## References

1. A. Binder, R. Freund, M. Oswald, L. Vock: Extended spiking neural P systems with excitatory and inhibitory astrocytes. *Proc. Eighth WSEAS Intern. Conf. on Evolutionary Computing*, Vancouver, Canada, 2007, 320-325.
2. C. Calude, Gh. Păun: Bio-steps beyond Turing. *BioSystems*, 77 (2004), 175–194.



3. E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun: *Grammar Systems. A Grammatical Approach to Distribution and Cooperation*. Gordon and Breach, London, 1994.
4. H. Chen, T.-O. Ishdorj, Gh. Păun: Computing along the axon. *Progress in Natural Science*, 17, 4 (2007), 418–423.
5. H. Chen, T.-O. Ishdorj, Gh. Păun, M.J. Pérez-Jiménez: Spiking neural P systems with extended rules. *Proc. Fourth Brainstorming Week on Membrane Computing*, Sevilla, 2006, RGNC Report 02/2006, 241–265.
6. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer-Verlag, Berlin, 1989.
7. R. Freund: Particular results for variants of P systems with one catalyst in one membrane. *Proc. Fourth Brainstorming Week on Membrane Computing*, Fénix Editora, Sevilla, 2006, vol. II, 41–50.
8. R. Freund: Purely catalytic P systems: Two catalysts can be sufficient for computational completeness. In the present volume.
9. R. Freund, O.H. Ibarra, A. Păun, P. Sosík, H.-C. Yen: Catalytic P systems. Chapter 4 of [33].
10. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science*, 330 (2005), 251–266.
11. R. Freund, Gh. Păun: Universal P systems: One catalyst can be sufficient. *Proc. 11th Brainstorming Week on Membrane Computing*, Sevilla, 4-8 February 2013, Fénix Editora, Sevilla, 2013.
12. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Frontiers of membrane computing: Open problems and research topics, *Intern. J. Found. Computer Sci.*, 2013 (first version in *Proc. Tenth Brainstorming Week on Membrane Computing*, Sevilla, January 30 – February 3, 2012, vol. I, 171–249).
13. O.H. Ibarra, Z. Dang, O. Egecioglu: Catalytic P systems, semilinear sets, and vector addition systems. *Th. Computer Sci.*, 312 (2004), 379–399.
14. O.H. Ibarra, Z. Dang, O. Egecioglu, G. Saxena: Characterizations of catalytic membrane computing systems. *28th Intern. Symp. Math. Found. Computer Sci.*, 2003 (B. Rovan, P. Vojtás, eds.), LNCS 2747, Springer, 2003, 480–489.
15. M. Ionescu, Gh. Păun: Notes about spiking neural P systems. *Proc. Ninth Brainstorming Week on Membrane Computing*, Sevilla, January 31 – February 4, 2011, Fénix Editora, Sevilla, 2011, 169–182.
16. M. Ionescu, Gh. Păun, M.J. Pérez-Jiménez, T. Yokomori: Spiking neural dP systems. *Fundamenta Informaticae*, 11, 4 (2011), 423–436.
17. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.
18. S.N. Krishna, A. Păun: Results on catalytic and evolution-communication P systems. *New Generation Computing*, 22 (2004), 377–394.
19. K. Krithivasan, Gh. Păun, A. Ramanujan: On controlled P systems. *Fundamenta Informaticae*, to appear.
20. A. Leporati, A.E. Porreca, C. Zandron, G. Mauri: Improving universality results on parallel enzymatic numerical P systems. *Proc. 11th Brainstorming Week on Membrane Computing*, Sevilla, 4-8 February 2013, Fénix Editora, Sevilla, 2013.
21. A. Leporati, A.E. Porreca, C. Zandron, G. Mauri: Enzymatic numerical P systems using elementary arithmetic operations. In the present volume.
22. M. Mutyám, K. Krithivasan: P systems with membrane creation: Universality and efficiency. *Proc. MCU 2001* (M. Margenstern, Y. Rogozhin, eds.), LNCS 2055, Springer, Berlin, 2001, 276–287.

23. L. Pan, Gh. Păun: Spiking neural P systems with anti-spikes. *Intern. J. Computers, Comm. Control*, 4, 3 (2009), 273–282.
24. A.B. Pavel, O. Arsene, C. Buiu: Enzymatic numerical P systems – a new class of membrane computing systems. *The IEEE Fifth Intern. Conf. on Bio-Inspired Computing. Theory and applications. BIC-TA 2010*, Liverpool, Sept. 2010, 1331–1336.
25. A.B. Pavel, C.I. Vasile, I. Dumitrache: Robot localization implemented with enzymatic numerical P systems. *Proc. Conf. Living Machines 2012*, LNCS 7375, Springer, 2012, 204–215.
26. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.*, 61 (2000), 108–143 (see also TUCS Report 208, November 1998, [www.tucs.fi](http://www.tucs.fi)).
27. Gh. Păun: Computing with membranes – A variant. *Intern. J. Found. Computer Sci.*, 11, 1 (2000), 167–182.
28. Gh. Păun: *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
29. Gh. Păun: Spiking neural P systems with astrocyte-like control, *JUCS*, 13, 11 (2007), 1707–1721.
30. Gh. Păun: Some open problems about numerical P systems. *Proc. 11th Brainstorming Week on Membrane Computing*, Sevilla, 4–8 February 2013, Fénix Editora, Sevilla, 2013.
31. Gh. Păun, R. Păun: Membrane computing and economics: Numerical P systems. *Fundamenta Informaticae*, 73 (2006), 213–227.
32. Gh. Păun, M.J. Pérez-Jiménez: Solving problems in a distributed way in membrane computing: dP systems. *Int. J. of Computers, Communication and Control*, 5, 2 (2010), 238–252.
33. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
34. Gh. Păun, S. Yu: On synchronization in P systems. *Fundamenta Informaticae*, 38, 4 (1999), 397–410.
35. P. Sosík, O. Valik: On evolutionary lineages of membrane systems. *Membrane Computing, International Workshop, WMC6, Vienna, Austria, 2005, Selected and Invited Papers*, LNCS 3850, Springer, Berlin, 2006, 67–78.
36. A. Syropoulos: *Hypercomputation: Computing Beyond the Church-Turing Barrier*. Springer, Berlin, 2008.
37. C.I. Vasile, A.B. Pavel, J. Kelemen: Implementing obstacle avoidance and follower behaviors on Koala robots using numerical P systems. *Tenth Brainstorming Week on Membrane Computing*, Sevilla, 2012, vol. II, 215–227.
38. C.I. Vasile, A.B. Pavel, I. Dumitrache: Universality of enzymatic numerical P systems. *Intern. J. Computer Math.*, in press.
39. C.I. Vasile, A.B. Pavel, I. Dumitrache, Gh. Păun: On the power of enzymatic numerical P systems. *Acta Informatica*, 49, 6 (2012), 395–412.
40. The P Systems Website: <http://ppage.psystems.eu>.

# Computational Complexity in Membrane Computing: Issues and Challenges

Petr Sosík<sup>1,2</sup>

<sup>1</sup> Departamento de Inteligencia Artificial, Facultad de Informática,  
Universidad Politécnica de Madrid, Campus de Montegancedo s/n,  
Boadilla del Monte, 28660 Madrid, Spain,

<sup>2</sup> Research Institute of the IT4Innovations Centre of Excellence,  
Faculty of Philosophy and Science, Silesian University in Opava  
74601 Opava, Czech Republic, [petr.sosik@fpf.slu.cz](mailto:petr.sosik@fpf.slu.cz)

## Abstract

We resume computational complexity aspects of various models of membrane systems. The scope of studied models include P systems with active membranes, P systems with proteins on membranes, tissue P systems both with membrane separation and membrane division, and spiking neural P systems. A sequence of common types of problems is studied in relation to these P system models.

The first studied problem is a condition guaranteeing the polynomial equivalence of P systems and Turing machines. This problem is not completely trivial as many models of P systems can store information in unary mode, resembling counter machines which, however, are exponentially slower than Turing machines.

Perhaps the most focused problem in this area is the ability of the system to solve NP/co-NP complete problems in polynomial time. Interesting characterizations of the borderline between tractability and intractability, i.e., P/NP, has been recently shown. Many of these models can process, in polynomial time, both problems in NP and co-NP, using the popular strategy of trading space for time. However, their exact relation to these complexity classes remains open.

Similarly important, although less popular, is the relation between NP/co-NP and PSPACE. Several models of P systems has been shown to characterize the class PSPACE, giving an opportunity to characterize the NP/PSPACE borderline. The class PSPACE itself characterizes parallel computations with an unlimited number of processors but a limited propagation of data between them. A relation of P systems to the classes of the polynomial hierarchy would be very interesting. There are also studies investigating the power of P systems working in sublinear time. Finally, a relation between spatial complexity of Turing machines and P systems has been recently studied.

Last, but not least, we question the role of families of P systems, their uniformity conditions and their necessity to solve computationally hard problems in polynomial time.



# Using the Formal Framework for P Systems

Sergey Verlan

<sup>1</sup> Laboratoire d'Algorithmique, Complexité et Logique,  
Université Paris Est – Créteil Val de Marne,  
61, av. gén. de Gaulle, 94010 Créteil, France

<sup>2</sup> Institute of Mathematics and Computer Science,  
Academy of Sciences of Moldova,  
Academiei 5, Chisinau, MD-2028, Moldova  
email: [verlan@u-pec.fr](mailto:verlan@u-pec.fr)

## Abstract

In this presentation we focus on the model called *formal framework for P systems* introduced in [2] and later developed in [1]. It aims to provide a concrete model of P systems that can act as descriptional language powerful enough to represent in a simple way most of the variants of P systems with the goal of better understanding and comparison of different models of P systems.

Informally speaking, a definition of a P system consists of:

- a description of the initial structure (indicating the graph relation between the compartments and any additional information like labels, charges, etc),
- a list of the initial multisets of objects present in each compartment at the beginning of the computation,
- a set of rules, acting over objects and / or over the structure.

A computation of a P system can be defined as a sequence of transitions between configurations ending in some halting configuration. To give a more precise description of the semantics we must define the following 4 notions (functions):

- $Applicable(\Pi, \mathcal{C}, \delta)$  – the set of multisets of rules of  $\Pi$  applicable in the configuration  $\mathcal{C}$ , according to the derivation mode  $\delta$ .
- $Apply(\Pi, \mathcal{C}, R)$  – the configuration obtained by the (parallel) application of the multiset of rules  $R$  to the configuration  $\mathcal{C}$ .
- $Halt(\Pi, \mathcal{C}, \delta)$  – a predicate that yields true if  $\mathcal{C}$  is a halting configuration of the system  $\Pi$  evolving in the derivation mode  $\delta$ .
- $Result(\Pi, \mathcal{C})$  – a function giving the result of the computation of the P system  $\Pi$ , when the halting configuration  $\mathcal{C}$  has been reached. Generally this is an integer function, however it is possible to generalize it, allowing e.g. Boolean or vector functions.

The transition of a P system  $\Pi$  according to the derivation mode  $\delta$  (generally this is the maximally parallel mode) is defined as follows: we pass from a configuration  $\mathcal{C}$  to  $\mathcal{C}'$  (written as  $\mathcal{C} \Rightarrow \mathcal{C}'$ ) iff

$$\mathcal{C}' = Apply(\Pi, \mathcal{C}, R), \text{ for some } R \in Applicable(\Pi, \mathcal{C}, \delta)$$

In general, the result of the computation of a P system is interpreted as the union of the results of all possible computations (in the same way as the language generated by a grammar is defined in formal language theory, gathering all possible derivations). Note that this is a theoretical (non-constructive) definition, since there may exist an infinite number of halting configurations reachable from a single initial configuration  $\mathcal{C}_0$ .

The precise definition of the four functions above depends on the selected model of P systems. The goal of works [1–3] is to provide a concrete variant of P systems (hence with concrete definitions of these functions), called a *formal framework*, such that most of existing models of P systems could be obtained by a restriction (eventually using a simple encoding) of this formal framework with respect to different parameters.

The configuration of the formal framework is a list of multisets corresponding to the contents of membranes of a P system and the rules generalize most kind of rules used in the P systems area. Based on this general form of rules, the applicability and the application of a (group of) rule(s) are defined using an algorithm. This permits to compute the set of all applicable multisets of rules for a concrete configuration  $\mathcal{C}$  ( $Applicable(\Pi, \mathcal{C})$ ). Then this set is restricted according to the derivation mode  $\delta$  ( $Applicable(\Pi, \mathcal{C}, \delta)$ ). For the transition, one of the multisets from this last set is non-deterministically chosen and applied, yielding a new configuration. The result of the computation is collected when the system halts according to the halting condition.

The aim of the presentation is not to present the framework itself, but rather several examples of its application for the description and the comparison of different variants of P systems with static structure, with probabilities and with dynamic structure. We also show how these investigations lead to new research ideas and open problems.

## References

1. R. Freund, I. Pérez-Hurtado, A. Riscos-Núñez, S. Verlan (2013), A formalization of membrane systems with dynamically evolving structures, *International Journal of Computer Mathematics*. Vol. 90(4), pp. 801-815.
2. R. Freund, S. Verlan, A Formal Framework for Static (Tissue) P Systems, In *Membrane Computing, 8th International Workshop, WMC 2007, Thessaloniki, Greece, June 25-28, 2007 Revised Selected and Invited Papers*. Lecture Notes in Computer Science, Vol. 4860, pp. 271-284, 2007.
3. S. Verlan. Study of language-theoretic computational paradigms inspired by biology. Habilitation thesis, University of Paris Est, 2010.

# Regular Papers





# A P System for Annotation of Romanian Affixes

Artiom Alhazov, Svetlana Cojocaru, Alexandru Colesnicov, Ludmila Malahov,  
Mircea Petic

Institute of Mathematics and Computer Science,  
Academy of Sciences of Moldova  
Academiei 5, Chişinău, MD-2028, Moldova  
E-mail: {artiom,Svetlana.Cojocaru,  
kae,mal,mirsha}@math.md

**Abstract.** This paper describes membrane computational models parsing affixed Romanian words with prefixes, suffixes, terminations, alterations in the root, and continues previous works on word derivation modeling. An algorithm for Romanian affixes extraction is given, and several models of P systems are proposed.

**Keywords:** affixation, morphemes, parsing, P system models, membrane computing, linguistic resources.

## 1 Introduction

Linguistic resources are necessary to solve different problems in natural language processing (NLP). They can exist as text collections, corpora, or dictionaries containing a lot of information. Processing of big volume of information takes the corresponding computer resources. Many problems in computer linguistics could be solved more effectively using parallel computations. Formal models based on principles of bio-molecular computations have inherent parallelism. Therefore, we found it natural to use these models to solve such problems. Models of membrane systems [10] for Romanian word derivation were proposed in [5, 4, 2, 1].

This paper discusses construction of membrane, or P systems to parse Romanian words with affixes. This is important because it permits to solve the inverse problem of creation of affixed linguistic. This contributes to replenishment of corpora and dictionaries, and to formation of morphological word nest for derivation.

Affixation is the most productive technique to form new Romanian words as the Romanian language possesses 86 prefixes and approximately 600 suffixes [8]. Nevertheless, this process has its peculiarities. Using inflexion, we get a priori correct words and know their morphological categories. Affixation as a mechanism of new word production cannot guarantee their correctness and does not permit to preview their semantic and morphological categories. This characterizes affixation as a totally non-regular process that complicates word generation.

That is why automated affixation is a difficult task as compared with other methods of word derivation.

Using data extracted from accessible lexicographical resources, we developed methods to check affixed words [6, 9]. We found their quantitative and qualitative characteristics. We developed a technique to produce affixed words, and got a set of restrictions that permits to filter inappropriate words.

We proposed in [1] several models of P systems to select affixed Romanian words based on these results. This is a continuation of that work, where we allow a derivation step to have more than one root alternation, addition of a prefix and a suffix, replacement of a termination with another one, as well as all of the above.

The paper consists of introduction, two sections, and conclusion. Sec. 2 gives main definitions of membrane systems and word derivation model. Sec. 3 of this paper discusses automated affix selection inside a word. An algorithm to solve this problem is given. A model of P system with replication is constructed to automatically analyze derived words with affixes. The model uses the matrix of rules taking into account alterations in the root in dependence of fixed prefixes and suffixes that we proposed. Sec. 4 gives examples of the constructed model work at affixed Romanian words parsing.

## 2 Definitions

### 2.1 Word derivation model

Consider a finite alphabet  $V$ . We assume that we are given a finite set of word pairs  $A$  of root alternations and finite languages  $\text{Pref}$  of prefixes,  $\text{RR}$  of roots,  $\text{Suf}$  of suffixes and  $T$  of terminations ( $T$  may include the empty word), all over  $V$ . We also write elements of  $A$  in the form  $x \rightarrow y$ . We use  $\overline{\text{Pref}}$ ,  $\widehat{\text{Suf}}$  to denote the sets  $\text{Pref}$ ,  $\text{Suf}$ , where all symbols of each word have lines or hats over them. These two cases correspond to operations of adding a prefix and adding a suffix. We denote the marked terminations by  $\boxed{T} = \{\boxed{t} \mid t \in T\}$ , and the termination rewriting rules by  $\overline{T} = \{\boxed{t_1} \rightarrow \boxed{t_2} \mid t_1, t_2 \in T\}$ . Let  $Op = \overline{\text{Pref}} \cup \widehat{\text{Suf}} \cup \overline{T} \cup A$ . The fourth case ( $A$ ) corresponds to an operation of performing an alternation.

Moreover, we assume we are given a finite language  $M$  over  $Op$ . A derivation step corresponding to a control word  $s = o_{i_1} \cdots o_{i_k} \in M$  consists of  $k$  operations from a set described above. We now define them more formally, using the syntax  $o(w)$  to denote the result of operation  $o$  over a word  $w$  (note that the result of some operations may be undefined on some words, the corresponding choice not leading to any result):

- $\overline{p}(w) = \overline{p}w$ ,
- $\widehat{s}(w\boxed{t}) = w\widehat{s}\boxed{t}$ ,
- $(\boxed{x} \rightarrow \boxed{y})(w\boxed{x}) = w\boxed{y}$ ,
- $(x \rightarrow y)(w_1xw_2) = w_1yw_2$ ,
- $(o_{i_1} \cdots o_{i_m})(w) = o_{i_1}(\cdots(o_{i_m}(w))\cdots)$ .

We will speak about the problem of **accepting** a language obtained by removing the prefix, suffix and termination marks from the words of the minimal language  $L$ , such that

- if  $wt \in \text{RR}$ , then  $w\boxed{t} \in L$ , and
- if  $w \in L$  and  $s \in M$  then  $s(w) \in L$  is defined.

Moreover, we would like the acceptor to also produce the lexical decomposition of the input.

## 2.2 Computing by P systems

Membrane computing is a recent domain of natural computing started by Gh. Păun in 1998. The components of a membrane system are a cell-like membrane structure, in the regions of which one places multisets of objects which evolve in a synchronous maximally parallel manner according to given evolution rules associated with the membranes. The necessary definitions are given in the following subsection, see also [11] for an overview of the domain and to [12] for the comprehensive bibliography.

Let  $O$  be a finite set of elements called symbols, then the set of words over  $O$  is denoted by  $O^*$ , and the empty word is denoted by  $\lambda$ .

**Definition 1.** *A P system with string-objects and input is a tuple*

- $\Pi = (O, \Sigma, \mu, M_1, \dots, M_p, R_1, \dots, R_p, i_0)$ , where:
- $O$  is the working alphabet of the system whose elements are called objects,
  - $\Sigma \subset O$  is an input alphabet,
  - $\mu$  is a membrane structure (a rooted tree) consisting of  $p$  membranes,
  - $M_i$  is an initial multiset of strings over  $O$  in region  $i$ ,  $1 \leq i \leq p$ ,
  - $R_i$  is a finite set of rules defining the behavior of objects from  $O^*$  in region  $i$ ,  $1 \leq i \leq p$ , as described below,
  - $i_0$  identifies the input region.

In this paper we consider string rewriting with target indications. A rule  $x \rightarrow (y, \text{tar}) \in R_i$  can be applied to a string  $uxv$  in region  $i$ , resulting in a string  $uyv$  in region specified by  $\text{tar} \in \{\text{in}_j \mid 1 \leq j \leq p\} \cup \{\text{here}, \text{out}\}$ . The target *here* may be omitted, together with a comma and parentheses.

We assume the following **computation mode**: whenever there are multiple ways to apply different rules (or the same rule) to a string, all possible results are produced (each possible result is performed on a different copy of the string; the string is either replicated, or assumed to be present in sufficient number of copies to allow this).

In our model of P systems, the membrane structure does not change. A configuration of a P system is its “snapshot”, i.e., the multisets of strings of objects present in regions of the system. While initial configuration is  $C_0 = (M_1, \dots, M_p)$ , each subsequent configuration  $C'$  is obtained from the previous configuration  $C$  by maximally parallel application of rules to objects, denoted by

$C \Rightarrow C'$  (no further rules are applicable together with the rules that transform  $C$  into  $C'$ ). A computation is thus a sequence of configurations starting from  $C_0$ , respecting relation  $\Rightarrow$  and ending in a halting configuration (no rule is applicable).

If  $S$  is a multiset of strings over the input alphabet  $\Sigma \subseteq O$ , then the *initial configuration* of a P system  $\Pi$  with an input  $S$  over alphabet  $\Sigma$  and input region  $i_0$  is

$$(M_1, \dots, M_{i_0-1}, M_{i_0} \cup S, M_{i_0+1}, \dots, M_p).$$

We consider the strings sent out of the skin membrane into the environment as the result of the computation.

### 3 Main construction

We proceed with parsing as the reverse process of the generation. For each possible decomposition of the string, the system sends outside a string, obtained from the input by erasing the endmarkers and inserting hyphens (for technical reasons, letters in prefixes and suffixes are marked, the reverse alternations are performed in both the termination and the rest of the word, and the termination is moved to the left of suffixes). In the notation below, we use  $'$  as a morphism:  $u'$  is a string obtained from  $u$  by priming all its letters.

We construct the following P system for accepting words  $x$  given in form  $\$1x\$2$ . We use an enumeration of elements of  $Op$  and  $T$ :  $Op = \{o_1, \dots, o_k\}$  and  $T = \{t_1, \dots, t_n\}$ . We recall that elements  $o_j$ ,  $1 \leq j \leq k$  are of the following forms:  $\bar{p}$ ,  $\hat{s}$ ,  $\boxed{t_1} \rightarrow \boxed{t_2}$ , and  $x \rightarrow y$ , where  $p \in \text{Pref}$ ,  $s \in \text{Suf}$ ,  $t_1, t_2 \in T$  and  $x, y \in V^*$ . We also define a set  $W = \text{Suf}(M^r)$  of suffixes of the mirror language of  $M$ ; words from  $W$  may appear in angular brackets. This corresponds to operations remaining to be undone at possible points of the parsing process.

$$\begin{aligned} \Pi = (&O, \mu, \Sigma, w_1, w_2, w_{o_1}, \dots, w_{o_k}, w_{t_1}, \dots, w_{t_n}, \\ &R_1, R_2, R_{o_1}, \dots, R_{o_k}, R_{t_1}, \dots, R_{t_n}, i_0 = 2), \\ O = &V \cup \bar{V} \cup \hat{V} \cup \boxed{T} \cup Op \cup \{\$, \$1, \$2, -, \langle, \rangle\}, \quad V = \{a, \dots, z\}, \\ \Sigma = &V \cup \{\$, \$2\}, \quad \bar{V} = \{\bar{a}, \dots, \bar{z}\}, \quad \hat{V} = \{\hat{a}, \dots, \hat{z}\}, \\ \mu = &[[ \ ]_2 [ \ ]_{o_1} \dots [ \ ]_{o_k} [ \ ]_{t_1} \dots [ \ ]_{t_n} ]_1, \\ w_i = &\lambda, \quad i \in \{1, 2\} \cup Op \cup T, \\ R_1 = &\{\langle \rangle \rightarrow \langle w \rangle \mid w \in M^r\} \cup \{\langle o \rightarrow (\langle, in_o \rangle) \mid o \in Op\}, \\ &\cup \{\langle \rangle \boxed{t} \rightarrow (\lambda, in_t) \mid t \in T\}, \\ &\cup \{\$, \$1q - t\$2 \rightarrow (q - t, out) \mid qt \in \text{RR}, t \in T\}, \\ R_2 = &\{t\$2 \rightarrow (\$, \$2\langle \rangle \boxed{t}, out) \mid t \in T\}, \\ R_{\bar{p}} = &\{\$, \$1p \rightarrow (\bar{p} - \$1, out)\}, \quad p \in \text{Pref}, \\ R_{\hat{s}} = &\{\$, \$2 \rightarrow (\$, \$2 - \hat{s}, out)\}, \quad s \in \text{Suf}, \end{aligned}$$

$$\begin{aligned}
R_q &= \{\boxed{t_2} \rightarrow (\boxed{t_1}, out)\}, \quad q = \boxed{t_1} \rightarrow \boxed{t_2}, \quad t_1, t_2 \in T. \\
R_a &= \{(y \rightarrow x, out)\}, \quad a = (x \rightarrow y) \in A, \\
R_t &= \{\$2 \rightarrow (-t\$2, out)\}, \quad t \in T.
\end{aligned}$$

Indeed, the work of  $\Pi$  consists of the reverse application of operations of adding affixes and alternations in terminations and the rest of the word, according to the control words from  $M$ . The role of endmarkers  $\$1$  and  $\$2$  consists of ensuring that prefixes from Pref and suffixes from Suf are only removed from the appropriate ends of the word.

The first step consists in marking of a termination in the word, sending the string out to region 1. The subsequent evolution is reduced to selecting and performing reverse derivation steps in regions corresponding to the operations; region 1 serves to control the substeps of the process. At any time, the system sends a copy of the word into a region corresponding to its termination, and back to region 1, effectively unmarking the termination and moving it to the left of all suffixes, separated by a hyphen from the root, in case the control word was emptied to  $\langle \rangle$ . If the word between the markers (the root and the termination) matches some word in RR, the resulting word is sent out.

Besides accepting words, the system also produces the decomposition of the word. In order to do so, instead of removing prefixes and suffixes, they are moved outside of the interval between  $\$1$  and  $\$2$ .

### 3.1 A finer algorithm

We propose a variation of the algorithm above, fulfilling the following goal: the alternations are only allowed in the root of the word, not in the prefixes or suffixes to be removed. We proceed as follows: all reverse alternations are replaced with the prime version of the letters. Once the choice is made to stop performing the operations (the string is in a region corresponding to its termination and the control symbol is removed), every letter can be unprimed, and then the result is sent out if some word from RR is obtained between the markers. We present the resulting P system.

$$\begin{aligned}
\Pi &= (O, \mu, \Sigma, w_1, w_2, w_{o_1}, \dots, w_{o_k}, w_{t_1}, \dots, w_{t_n}, \\
&\quad R_1, R_2, R_{o_1}, \dots, R_{o_k}, R_{t_1}, \dots, R_{t_n}, i_0 = 2), \\
O &= V \cup V' \cup \bar{V} \cup \hat{V} \cup \boxed{T} \cup Op \cup \{\$1, \$2, -\}, \quad V = \{a, \dots, z\}, \\
V' &= \{a', \dots, z'\}, \quad \bar{V} = \{\bar{a}, \dots, \bar{z}\}, \quad \hat{V} = \{\hat{a}, \dots, \hat{z}\}, \quad \Sigma = V \cup \{\$1, \$2\}, \\
\mu &= [\begin{smallmatrix} [ & ]_2 & [ & ]_{o_1} & \dots & [ & ]_{o_k} & [ & ]_{t_1} & \dots & [ & ]_{t_n} & ]_1 \end{smallmatrix}], \\
w_i &= \lambda, \quad i \in \{1, 2\} \cup Op \cup T, \\
R_1 &= \{\langle \rangle \rightarrow \langle w \rangle \mid w \in M^r\} \cup \{\langle o \rightarrow (\langle, in_o) \mid o \in Op\}, \\
&\quad \cup \{\langle \rangle \boxed{t} \rightarrow (\lambda, in_t) \mid t \in T\}, \\
&\quad \cup \{\$1q - t\$2 \rightarrow (q - t, out) \mid qt \in RR, \quad t \in T\},
\end{aligned}$$

$$\begin{aligned}
R_2 &= \{t\$2 \rightarrow (\$2\langle \rangle \boxed{t}, out) \mid t \in T\}, \\
R_{\bar{p}} &= \{\$1p \rightarrow (\bar{p} - \$1, out)\}, \quad p \in \text{Pref}, \\
R_{\hat{s}} &= \{s\$2 \rightarrow (\$2 - \hat{s}, out)\}, \quad s \in \text{Suf}, \\
R_q &= \{\boxed{t_2} \rightarrow (\boxed{t_1}, out)\}, \quad q = \boxed{t_1} \rightarrow \boxed{t_2}, \quad t_1, t_2 \in T. \\
R_a &= \{(z \rightarrow x', out) \mid {}'^{-1}(z) = y\}, \quad a = (x \rightarrow y) \in A, \\
R_t &= \{a' \rightarrow a \mid a \in V\} \cup \{\$2 \rightarrow (-t\$2, out)\}, \quad t \in T.
\end{aligned}$$

The notation  $'^{-1}$  means removing all primes from the letters of the argument. Although it assumes an exponential number of rules with respect to the size of a root alternation, this size is never too long.

## 4 Parsing in the Romanian language

We start by illustrating the work of the last P system by an example of a fragment of a computation where  $\text{Pref} = \{\text{des}\}$ ,  $\text{RR} = \{\text{praf}\}$ ,  $\text{Suf} = \{\text{ui, re}\}$ ,  $T = \{\lambda\}$  and  $M = \{(\bar{d} \bar{e} \bar{s}), (a \rightarrow \check{a})(\widehat{ui}), (\widehat{re})\}$ , and the system processes input  $\$1\text{desprăfui}\$2$ . For conciseness, we only list the first evolution of the copies of the string leading to the output, using the notation (string, region). (The other two are obtained if the prefix *des* is marked before both suffixes or after one of them, yielding the same results, while for technical reasons some strings remain blocked in the system, not contributing to the result).

$$\begin{aligned}
(\$1\text{desprăfui}\$2, 2) &\Rightarrow (\$1\text{desprăfui}\$2\langle \rangle \boxed{\lambda}, 1) \Rightarrow \\
&(\$1\text{desprăfui}\$2\langle \rangle (\widehat{re}) \boxed{\lambda}, 1) \Rightarrow (\$1\text{desprăfui}\$2\langle \rangle \boxed{\lambda}, (\widehat{re})) \Rightarrow \\
&(\$1\text{desprăfui}\$2 - \widehat{re} \langle \rangle \boxed{\lambda}, 1) \Rightarrow (\$1\text{desprăfui}\$2 - \widehat{re} \langle \rangle (\widehat{ui})(a \rightarrow \check{a}) \boxed{\lambda}, 1) \Rightarrow \\
&(\$1\text{desprăfui}\$2 - \widehat{re} \langle \rangle (a \rightarrow \check{a}) \boxed{\lambda}, (\widehat{ui})) \Rightarrow (\$1\text{desprăf}\$2 - \widehat{ui} - \widehat{re} \langle \rangle (a \rightarrow \check{a}) \boxed{\lambda}, 1) \\
&\Rightarrow (\$1\text{desprăf}\$2 - \widehat{ui} - \widehat{re} \langle \rangle \boxed{\lambda}, (a \rightarrow \check{a})) \Rightarrow (\$1\text{despra'f}\$2 - \widehat{ui} - \widehat{re} \langle \rangle \boxed{\lambda}, 1) \\
&\Rightarrow (\$1\text{despra'f}\$2 - \widehat{ui} - \widehat{re} \langle \rangle (\bar{d} \bar{e} \bar{s}) \boxed{\lambda}, 1) \Rightarrow \\
&(\$1\text{despra'f}\$2 - \widehat{ui} - \widehat{re} \langle \rangle \boxed{\lambda}, (\bar{d} \bar{e} \bar{s})) \Rightarrow (\bar{d} \bar{e} \bar{s} - \$1\text{pra'f}\$2 - \widehat{ui} - \widehat{re} \langle \rangle \boxed{\lambda}, 1) \\
&\Rightarrow (\bar{d} \bar{e} \bar{s} - \$1\text{pra'f}\$2 - \widehat{ui} - \widehat{re}, \lambda) \Rightarrow (\bar{d} \bar{e} \bar{s} - \$1\text{praf}\$2 - \widehat{ui} - \widehat{re}, \lambda) \Rightarrow \\
&(\bar{d} \bar{e} \bar{s} - \$1\text{praf}\$2 - \widehat{ui} - \widehat{re}, 1) \Rightarrow (\bar{d} \bar{e} \bar{s} - \text{praf} - \widehat{ui} - \widehat{re}, 0).
\end{aligned}$$

By inspecting the examples, we have come to the conclusion that a derivation step can include, in the worst case, a prefix, a suffix, two root alternations and replacing a termination with another one. Some of the above mentioned operations may be absent.

We should note that the division of a word into morphemes may sometimes differ from the one commonly accepted in linguistics. However, this should not restrict the generality of the approach, and we did so in order to simplify the explanation.

In the parsing process described above, we accounted for the prefixes, suffixes, root alternations and the terminations. As we have already stated, Romanian

language has 86 prefixes and about 600 suffixes, see, e.g., [8]. Processing the dictionary [7] (not claiming its comprehensiveness) let us distinguish the following types of root alternations during the word derivation:

- of vowels:  $a \rightarrow \check{a}$ ,  $a \rightarrow e$ ,  $e \rightarrow \check{a}$ ,  $o \rightarrow u$ ,  $\hat{i} \rightarrow i$ ,  $\check{a} \rightarrow e$ ,  
 $ea \rightarrow e$ ,  $e \rightarrow ea$ ,  $oa \rightarrow o$ ,  $oa \rightarrow u$ ,  $ia \rightarrow ie$
- of consonants:  $t \rightarrow \check{t}$ ,  $d \rightarrow z$ ,  $h \rightarrow \check{s}$ ,  $z \rightarrow j$ ,  $d \rightarrow j$ ,  $t \rightarrow c$ ,  $t \rightarrow ci$

Note that, if desired, we can use the context information to refine the scope of the the root alternation rules, e.g., if we only wanted to perform the alternation  $a \rightarrow \check{a}$  between letters t and r, we could write this as a rule  $tar \rightarrow \check{t}ar$ , which does not affect the model at all.

The set of terminations that we use in our algorithm (set  $T$ ) consists of terminations for nouns and adjectives ( $\check{a}$ , e, ea, a, i, ică, the empty termination  $\lambda$ , u, o, a, l, iu, ui, ie, uie) and those for verbs (a, ea, e, i, î).

We now proceed with some more examples of input and output, so let us agree that

$$\begin{aligned}
 \text{Pref} &\supseteq \{\text{im}, \hat{\text{in}}, \text{de}, \text{re}, \text{des}\}, \\
 \text{RR} &\supseteq \{\text{pune}, \text{flori}, \text{flex}, \text{scrie}, \text{cicl}, \text{fac}, \text{tânăr}, \text{fată}, \text{mult}, \text{deştept}, \text{brad}, \text{praf}\}, \\
 \text{Suf} &\supseteq \{\text{ere}, \text{are}, \text{ibil}, \text{re}, \text{ire}, \text{i}, \text{i}\check{t}, \text{im}, \text{u}\check{t}, \text{ui}\}, \\
 T &\supseteq \{\lambda, \check{a}, e\}, \\
 A &\supseteq \{\hat{a} \rightarrow i, \check{a} \rightarrow e, a \rightarrow e, t \rightarrow \check{t}, e \rightarrow ea, a \rightarrow \check{a}\}, \\
 M &\supseteq \{(\hat{i} \ \overline{m}), (\hat{i} \ \overline{n}), (\overline{d} \ \overline{e}), (\overline{r} \ \overline{e}), (\overline{d} \ \overline{e} \ \overline{s}), (\widehat{ere}), (\widehat{are}), (\widehat{ibil}), (\widehat{re}), (\widehat{ire}), \\
 &\quad (\hat{a} \rightarrow i)(\check{a} \rightarrow e)(\widehat{\text{in}})(\widehat{i}), (a \rightarrow e)(\widehat{i}\check{t}), (t \rightarrow \check{t})(\widehat{im}), (\boxed{\lambda} \rightarrow \boxed{e}) \\
 &\quad (e \rightarrow ea)(\boxed{\lambda} \rightarrow \boxed{\check{a}}), (a \rightarrow \check{a})(\widehat{ut}), (a \rightarrow \check{a})(\widehat{ui})\}.
 \end{aligned}$$

Examples without root alternations: words

$\$_1\text{înflorire}\$_2$ ,  $\$_1\text{flexibil}\$_2$ ,  $\$_1\text{descriere}\$_2$ ,  $\$_1\text{reciclare}\$_2$ ,  $\$_1\text{desfacere}\$_2$  (burst into blossom, flexible, description, recycling, disassembling) will yield output  $\hat{i} \ \overline{n}\text{-flori--}\widehat{re}$ ,  $\text{flex--}\widehat{ibil}$ ,  $\overline{d} \ \overline{e}\text{-scrie--}\widehat{re}$ ,  $\overline{r} \ \overline{e}\text{-cicl--}\widehat{are}$ , and  $\overline{d} \ \overline{e} \ \overline{s}\text{-fac--}\widehat{ere}$ , respectively.

Examples with root alternations: words  $\$_1\text{întineri}\$_2$ ,  $\$_1\text{fetiţă}\$_2$ ,  $\$_1\text{multime}\$_2$ ,  $\$_1\text{deşteaptă}\$_2$ ,  $\$_1\text{brăduţ}\$_2$  and  $\$_1\text{desprăfuire}\$_2$  (youthen, little girl, multitude, dignified (fem.), small spruce, undusting) will yield output  $\hat{i}\overline{m}\text{-tânăr--}\widehat{i}$ ,  $\text{fat--}\check{a}\text{-}\widehat{i}\check{t}$ ,  $\text{mult--}\widehat{im}$ ,  $\text{deştept--}\widehat{ui}$ , and  $\overline{d} \ \overline{e} \ \overline{s}\text{-praf--}\widehat{ui} - \widehat{re}$ , respectively.

## 5 Conclusions

The paper discussed P systems used to word derivation in Romanian, namely, affixation of nouns, adjectives, and verbs as most productive parts of speech at lemmas affixation. We proposed variants of membrane parsing model taking into account alternations in the root dependent of fixed prefixes and suffixes. We may deduce that these models can be used not only for Romanian but for

other languages with analogous type of word derivation. These models can also be integrated into another NLP applications to solve more complicated problems in computer linguistics.

## Acknowledgements

This work was supported by project STCU-5384 “Models of high performance computations based on biological and quantum approaches”, and project ref. nr. 12.819.18.09A “Development of IT support for interoperability of electronic linguistic resources” from Supreme Council for Science and Technological Development of the Republic of Moldova.

## References

1. A. Alhazov, E. Boian, S. Cojocaru, A. Colesnicov, L. Malahov, M. Petic, C. Ciubotaru: Membrane Models of Romanian Word Affixation. In: *Applied linguistics and linguistic technologies: MegaLing-2012*. Kyiv: 2013 (in print) - In Russian.
2. A. Alhazov, E. Boian, S. Cojocaru, Yu. Rogozhin: Modelling Inflections in Romanian Language by P Systems with String Replication. *Computer Science Journal of Moldova* **17**, 2(50), 2009, 160–178.
3. A. Alhazov, S. Cojocaru, L. Malahova, Yu. Rogozhin: Dictionary Search and Update by P Systems with String-Objects and Active Membranes. *International Journal of Computers, Communications & Control* IV, 3, 2009, 206–213.
4. E. Boian, C. Cojocaru, A. Colesnicov, L. Malahov, C. Ciubotaru: Modeling of Romanian Word Derivation by Membrane Computations. In: *Applied linguistics and linguistic technologies: MegaLing-2011*, Kyiv, 2012, 57–72 - In Russian.
5. E. Boian, C. Cojocaru, A. Colesnicov, L. Malahov, C. Ciubotaru: P Systems in Computer Linguistics. In: *Applied linguistics and linguistic technologies: MegaLing-2009*, Kyiv, 2009, 62–70 - In Russian.
6. S. Cojocaru, E. Boian, M. Petic: Stages in Automatic Derivational Morphology Processing. In: *Knowledge Engineering, Principles and Techniques*, KEPT2009, Cluj-Napoca University Press, 2009, 97–104.
7. S. Constantinescu: *Dictionary of Derived Words*, HERRA, Bucharest, 2008. 288 pp. - In Romanian.
8. Al. Graur, M. Avram: *Word formation in Romanian*, vol. **II**, Romanian Academy Press, Bucharest, 1978, 310 pp. - In Romanian.
9. M. Petic: Automatic Derivational Morphology Contribution to Romanian Lexical Acquisition. In: *Natural Language Processing and its Application. Research in Computing Science*. Mexico, vol. **46**, 2010, 67–78.
10. Gh. Păun: *Membrane Computing. An Introduction*. Springer-Verlag. 2002, 420 pp.
11. Gh. Păun, G. Rozenberg, A. Salomaa: *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010, 118–143.
12. P systems webpage. <http://ppage.psystems.eu>



# Behavioural Equivalences in Real-Time P Systems

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science  
Blvd. Carol I no.8, 700506 Iași, Romania  
baman@iit.tuiasi.ro, gabriel@info.uaic.ro

**Abstract.** We present a real-time extension of P systems in which each membrane and each object has a lifetime attached to it, and we use these lifetimes to define and to study various behavioural equivalences. We also establish sufficient conditions for guaranteeing progression over time.

## 1 Introduction

Biologists are becoming increasingly aware that formal methods can help to avoid resources consumption in lab experiments [3]. The field of “computational methods in system biology” provides formal frameworks which are able to faithfully describe the behaviour of complex systems, to provide qualitative and quantitative reasoning, as well as to compare the similar behaviour of two related systems. During the last years, membrane computing [9, 14] has been applied to biology and could have an important impact in understanding how biological systems work, giving at the same time a way to describe, manipulate, analyse and verify them.

In this paper we define and study a real-time extension of P systems, inspired by the P systems with lifetimes defined in [1]. Inspired from biology where cells and intracellular proteins have a well-defined lifetime, we assign real-time lifetimes to each membrane and to each object. In order to simulate the passage of time, we use rules of the form  $(a, t) \xrightarrow{d} (a, t-d)$  for objects, and  $[ ]_{(i,t)} \xrightarrow{d} [ ]_{(i,t-d)}$  for membranes, for  $d \in \mathbb{R}$  and  $d \leq t$ . If the lifetime of an object  $a$  reaches 0 then the object is used to create a new multiset of objects  $u$  by applying a rule of the form  $(a, 0) \rightarrow u$ , while if the lifetime of a membrane  $i$  reaches 0 then the membrane is marked for dissolution by applying a rule of the form  $[ ]_{(i,0)} \rightarrow [\delta]_{(i,0)}$ . After dissolving a membrane, all objects and membranes previously contained in it become elements of the immediately upper membrane. A similar idea has been considered for spiking P systems, where a life duration was added for spikes, but not for cells [11]. If a spike has a lifetime equal to zero, it is removed.

A time-independent P system is a P system that always produces the same result, independently of the execution times of the rules [6]. If one assumes the existence of two time scales (an external time for the user, and an internal time for the device), then it is possible to construct accelerated P systems [5].

Behavioural equivalence is an important concept in biology, necessary for comparing the behaviour of various (sub)systems. For example, an artificial organ should be the functional equivalent of a natural organ, meaning that both behave in a similar manner up to a given time; e.g. the artificial kidney has the same functional characteristics as an “in vivo” kidney. Recently, it was shown in [12] that the vas deferens’ of the human, canine and bull are equivalent in many ways, including histological similarities. In [10], different methods are presented for comparing protein structures in order to discover common patterns.

When choosing which equivalence relation to adopt for a given model, we need to decide what properties are to be preserved by the equivalence relation. In membrane computing, two P systems (also called membrane systems) are considered to be equivalent whenever they have the same input/output behaviour [14]. Such an equivalence does not consider the temporal evolution of the two systems. Behavioural equivalences (bisimulations) for membrane systems were defined in [2, 4, 8]. As a novelty, we are looking for systems with equivalent timed behaviour. By defining several equivalences, we offer flexibility in selecting the right one when verifying biological systems and comparing them.

In computer science, theoretical methods are used to implement software tools able to verify the properties of complex concurrent systems. It is reasonable to expect that, for real-time P systems, we can create or adapt some tools based on verification techniques using temporal logics. What we do in this paper represents a first step in this direction, namely establishing the formal framework used in software verifications for biological systems sensitive to timeouts.

## 2 Real-Time P Systems

Membrane systems are essentially parallel and nondeterministic computing models inspired by the compartments of eukaryotic cells and their biochemical reactions. The structure of the cell is represented by a set of hierarchically embedded membranes which are all contained inside a skin membrane. The molecular species (ions, proteins, etc.) floating inside cellular compartments are represented by multisets of objects described by means of symbols over a given alphabet. The objects can be modified or communicated between adjacent compartments. Chemical reactions are represented by evolution rules which operate on the objects, as well as on the compartmentalised structure (by dissolving, dividing, creating, or moving membranes).

A membrane system can perform computations in the following way: starting from an initial configuration (the initial membrane structure and the initial multisets of objects placed inside the membranes), the system evolves by applying the evolution rules of each membrane in a nondeterministic manner. A rule is applicable when all the objects which appear in its left hand side are available in the membrane where the rule is placed.

Several variants of membrane systems are inspired by different aspects of living cells (communication through membranes, membrane mobility, etc.). Their computing power and efficiency have been investigated using the approaches of

formal languages, grammars, register machines and complexity theory. Membrane systems are presented together with many variants and examples in [13]. Several applications of these systems are presented in [9]. An updated bibliography can be found at the P systems web page <http://ppage.psysteams.eu>.

For an alphabet  $V = \{a_1, \dots, a_n\}$ , we denote by  $V^*$  the set of all strings over  $V$ ;  $\lambda$  denotes the empty string and  $V^+ = V^* \setminus \{\lambda\}$ . We use the string representation of multisets that is widely used in the field of membrane systems. An example of such a representation is the multiset  $u = aba$ , where  $u(a) = 2$ ,  $u(b) = 1$ . Given two multisets  $u, v$  over  $V$ , for any  $a \in V$ , we have  $(u \uplus v)(a) = u(a) + v(a)$  as the multiset union, and  $(u \setminus v)(a) = \max\{0, u(a) - v(a)\}$  as the multiset difference. We use  $\mathbb{R}_+$  to denote the non-negative reals.

Next we define real-time P systems, a variant of P systems with lifetimes [1].

**Definition 1.** A real-time P system of degree  $n \geq 1$  is a construct

$$\Pi = (V_t, H_t, \mu_t, w_1, \dots, w_n, \mathcal{L}, (R_1, \rho_1), \dots, (R_n, \rho_n)), \text{ where:}$$

1.  $V_t \subseteq V \times (\mathbb{R} \cup \infty)$  is a set of pairs of the form  $(a, t_a)$ , where  $a \in V$  is an object and  $t_a \in (\mathbb{R} \cup \infty)$  is the lifetime of the object  $a$ ;
2.  $H_t \subseteq H \times (\mathbb{R} \cup \infty)$  is a set of pairs of the form  $(h, t_h)$ , where  $h \in H$  is a membrane label and  $t_h \in (\mathbb{R} \cup \infty)$  is the lifetime of the membrane  $h$ ;
3.  $\mu_t \subseteq H_t \times H_t$  is a tree that describes the membrane structure, such that  $((i, t_i), (j, t_j)) \in \mu_t$  denotes that the membrane labelled by  $j$  with lifetime  $t_j$  is contained in the membrane labelled by  $i$  with lifetime  $t_i$ ; this structure does not depend on the lifetimes of the involved membranes;
4.  $w_i \subseteq (V_t)^*$  is a multiset of pairs from  $V_t$  assigned initially to membrane  $i$ ;
5.  $\mathcal{L}$  is a set of labels that attaches a unique label to each rule from  $R_1, \dots, R_n$ ;
6.  $R_i$ ,  $1 \leq i \leq n$ , is a finite set of evolution rules from membrane  $i$  of the following forms:
  - (a)  $r : u \rightarrow v$ , with  $u \in V_t^+$ , either  $v = v'$  or  $v = v'\delta$ ,  $v' \in ((V_t \times \{\text{here}, \text{out}\}) \cup (V_t \times \{\text{in}_j \mid 1 \leq j \leq n\}))^*$ ,  $r \in \mathcal{L}$ ;  $\delta$  is a special symbol not appearing in  $V$ ;  
 Considering that the multiset of objects  $u$  was placed inside membrane  $i$ , the targets indicate where, in the membrane structure, the multiset of objects  $v$  obtained from  $u$  should be placed: here - it remains in  $i$ ; out - is placed in the parent membrane of  $i$ ;  $\text{in}_j$  - is moved in a child of  $i$  labelled by  $j$ .
  - (b)  $r : (a, 0) \rightarrow u$ , for all  $a \in V$ ,  $u \in V_t^+$ ,  $r \in \mathcal{L}$   
 If an object  $a$  has the lifetime 0 then the object is replaced with the multiset  $u$ , thus simulating the degradation of proteins and the fact that new compounds are obtained.
  - (c)  $r : [\ ]_{(i,0)} \rightarrow [\delta]_{(i,0)}$ , for all  $1 \leq i \leq n$ ,  $r \in \mathcal{L}$   
 If the lifetime of a membrane  $i$  reaches 0 the membrane is dissolved.
7.  $\rho_i$ , for all  $1 \leq i \leq n$ , is a partial order relationship defined over the rules in  $R_i$  specifying a priority relation between these rules.

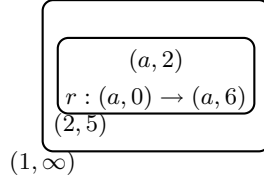
Therefore, a real-time P systems consists of a membrane structure  $\mu$  containing  $n \geq 1$  membranes, where each membrane  $i$  gets assigned a finite multiset of objects  $w_i$  and a finite set of evolution rules  $R_i$ . The sets  $V_t$  and  $H_t$  are potentially

uncountable, but at any moment a real-time P system contains just a finite number of objects and membranes. An evolution rule can produce the special object  $\delta$  to specify that, after the application of the rule, the membrane containing  $\delta$  has to be dissolved. After dissolving a membrane, all objects and membranes previously present in it become elements of the immediately upper membrane, while the rules of the dissolved membrane are removed. When no rule from  $R_i$ ,  $1 \leq i \leq n$ , is applicable, all clocks of a real-time P system are decreased with the same value (the minimum value of the present delays), such that a new rule from  $R_i$ ,  $1 \leq i \leq n$ , is applicable.

**Definition 2.** For a real-time P system  $\Pi$  the initial configuration is defined as  $C_0 = (\mu_t, w_1, \dots, w_n)$ . The set of all configurations over a real-time P system  $\Pi$  is denoted by  $\mathcal{C}_\Pi$ .

*Example 1.* Consider the following real-time P system of degree 2:

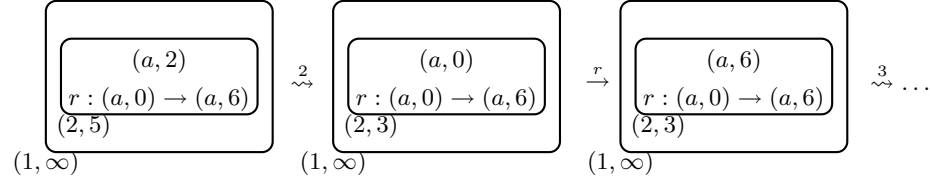
$\Pi_1 = (a \times (\mathbb{R} \cup \infty), \{1 \times \infty, 2 \times (\mathbb{R} \cup \infty)\}, \{(1, \infty), (2, 5)\}, \emptyset, (a, 2), \{r\}, \emptyset, (\{r : (a, 0) \rightarrow (a, 6)\}))$ . Graphically this looks like this:



The initial configuration for  $\Pi_1$  is  $C_1^0 = (\{(1, \infty), (2, 5)\}, \emptyset, (a, 2))$ . Starting from this initial configuration,  $\Pi$  has the following evolution:

$$C_1^0 \xrightarrow{2} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 0)) \xrightarrow{r} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 6)) \xrightarrow{3} \dots$$

Graphically these steps of evolution are represented as



The label of  $\xrightarrow{2}$  is determined by comparing the lifetime of object  $a$  (namely 2) with the lifetime of membrane 2 (namely 5), and then taking the minimum value (namely 2). Similar for the label of  $\xrightarrow{3}$ , that is the minimum between 6 and 3.

### 3 Timed Labelled Transition Systems

The operational semantics of a formalism is typically defined by using labelled transition systems. For formalisms involving time, modelling the passage of time may be encoded in timed labelled transition systems that distinguish between transitions due to rules and those due to passage of time.

**Definition 3.** A **timed labelled transition system (shortly TLTS)** is a tuple  $(\mathcal{C}_\Pi, C_0, \mathcal{L}, \rightarrow, \rightsquigarrow)$  where  $\mathcal{C}_\Pi$  is a set of configurations,  $C_0$  is the initial

configuration,  $\mathcal{L}$  is a set of labels,  $\rightarrow \subseteq \mathcal{C}_\Pi \times \mathcal{L} \times \mathcal{C}_\Pi$  is a **rule transition relation** and  $\rightsquigarrow \subseteq \mathcal{C}_\Pi \times \mathbb{R}_+ \times \mathcal{C}_\Pi$  is a **timed transition relation**. We write  $C \xrightarrow{r} C'$  for  $(C, r, C') \in \rightarrow$  and  $C \xrightarrow{d} C'$  for  $(C, d, C') \in \rightsquigarrow$ . If there is no such  $C' \in \mathcal{C}_\Pi$  such that  $C \xrightarrow{r} C'$  ( $C \xrightarrow{d} C'$ ), then we write  $C \not\xrightarrow{r} (C \not\xrightarrow{d})$ .

The timed labelled transition system of Definition 3 is general and can be applied to any formalism involving time. A particular system for real-time P systems is given by Table 1 and Definition 4.

**Table 1.** Rule Transitions (left column) and Timed Transitions (right column)

|   |  |
|---|--|
| <p>(OBJ) <math display="block">\frac{r : (a, 0) \rightarrow u \in R_i, (a, 0) \in w_i, \quad C = (\mu_t, w_1, \dots, w_n)}{C \xrightarrow{r} C' = (\mu'_t, w'_1, \dots, w'_n), \text{ with } \mu'_t = \mu_t, \quad w'_i = (w_i \setminus (a, 0)) \uplus \{u_t \mid (u_t, \text{here}) \in u\} \quad w'_j = w_j \uplus \{u_t \mid (u_t, \text{out}) \in u, ((j, t_j), (i, t_i)) \in \mu_t\} \quad w'_k = w_k \uplus \{u_t \mid (u_t, \text{in}_k) \in u, ((i, t_i), (k, t_k)) \in \mu_t\} \quad w'_m = w_m, m \neq i, j, k}</math></p> | <p>(TOBJ) <math display="block">\frac{(a, t) \in V_t, \quad 0 \leq d \leq t}{(a, t) \xrightarrow{d} (a, t-d)}</math></p>   |
| <p>(TMULT) <math display="block">\frac{0 \leq d \leq t, \quad \forall (a, t) \in w_i, (a, t) \xrightarrow{d} (a, t-d)}{w_i \xrightarrow{d} w'_i, \text{ with } w'_i = \{(a, t-d) \mid (a, t) \in w_i\}}</math></p>  |  |
| <p>(MEM) <math display="block">\frac{r : [\ ]_{(i,0)} \rightarrow [\delta]_{(i,0)}, \quad ((j, t_j), (i, 0)) \in \mu_t, \quad C = (\mu_t, w_1, \dots, w_n)}{C \xrightarrow{r} C' = (\mu'_t, w'_1, \dots, w'_n) \quad \text{with } w'_i = \emptyset, w'_j = w_i \uplus w_j \quad w'_k = w_k, \text{ for } k \neq i, k \neq j \quad \text{and } \mu'_t = ((\mu_t \setminus ((i, t_i), (j, t_j)))) \setminus \{(i, 0), (k, t_k)\} \uplus \{(j, t_j), (k, t_k)\}}</math></p>  | <p>(TMEM) <math display="block">\frac{(i, t) \in H_t, \quad 0 \leq d \leq t}{(i, t) \xrightarrow{d} (i, t-d)}</math></p>   |
| <p>(TSTRUCT) <math display="block">\frac{0 \leq d \leq t_i, 0 \leq d \leq t_j, \quad \forall ((i, t_i), (j, t_j)) \in \mu_t, \quad (i, t_i) \xrightarrow{d} (i, t_i-d), \quad (j, t_j) \xrightarrow{d} (j, t_j-d)}{\mu_t \xrightarrow{d} \mu'_t, \quad \text{with } \mu'_t = \{((i, t_i-d), (j, t_j-d)) \mid ((i, t_i), (j, t_j)) \in \mu_t\}}</math></p>   |  |
| <p>(EVOL) <math display="block">\frac{r : u \rightarrow v \in R_i, u \in w_i, \quad C = (\mu_t, w_1, \dots, w_n)}{C \xrightarrow{r} C' = (\mu'_t, w'_1, \dots, w'_n) \quad \text{with } w'_j = w_j, \text{ for } j \neq i \quad \text{and } w'_i = (w_i \setminus u) \uplus v}</math></p>   | <p>(TEVOL) <math display="block">\frac{d \in \mathbb{R}_+, \mu_t \xrightarrow{d} \mu'_t, \quad \forall 1 \leq i \leq n, w_i \xrightarrow{d} w'_i, \quad C = (\mu_t, w_1, \dots, w_n)}{C \xrightarrow{d} C' = (\mu'_t, w'_1, \dots, w'_n)}</math></p> |

The transition relation  $\rightarrow$  describes rule application and a sequence of such transitions describes an execution within the same instant of real-time, whereas the timed transition relation  $\rightsquigarrow$  describes the passage of real-time. The operational semantics of a real-time P system  $\Pi$  is:

**Definition 4.** For a membrane system  $\Pi$  the TLTS is  $(\mathcal{C}_\Pi, C_0, \mathcal{L}, \rightarrow, \rightsquigarrow)$  where the relations  $\rightarrow$  and  $\rightsquigarrow$  are the smallest relations satisfying the inference rules from Table 1, the priority relations  $\rho_i$  of  $\Pi$ , and the following constraint expressing that  $\rightarrow$  has a higher priority than  $\rightsquigarrow$  and guaranteeing maximal progress:

if there exists  $C'$  such that  $C \xrightarrow{r} C'$ , then  $C \not\rightsquigarrow^d$  for all  $d > 0$ .

In Table 1, in rule **(OBJ)** an object  $(a, 0) \in w_i$  is replaced by a multiset  $u$  and the membrane structure remains unchanged, thus the configuration  $C = (\mu_t, w_1, \dots, w_n)$  is transformed into the configuration  $C' = (\mu'_t, w'_1, \dots, w'_n)$  with  $\mu'_t = \mu_t$ , and the objects distributed among membranes according to the structure  $\mu_t$  and the targets from  $u$ . Rules **(EVOL)** and **(TEVOL)** are similar to **(OBJ)**. In rule **(MEM)** a membrane  $i$  is dissolved and thus  $w_i, w_j$  ( $i$  is placed inside  $j$ ) and  $\mu_t$  are modified to  $w'_i = \emptyset$ ,  $w'_j = w_i \uplus w_j$  and  $\mu'_t$  (identical to  $\mu_t$  excepting that the pair  $((i, t_i), (j, t_j))$  is removed and all  $(i, i_t)$  are replaced by  $(j, j_t)$ ).

**Proposition 1.** For any  $C, C', C'' \in \mathcal{C}_\Pi$ , and any  $d, d' \in \mathbb{R}_+$ ,

1. **(Zero time advance)**  $C \rightsquigarrow^0 C$ .
2. **(Time determinacy)** If  $C \rightsquigarrow^d C'$  and  $C \rightsquigarrow^d C''$ , then  $C' = C''$ .
3. **(Time continuity)**  $C \rightsquigarrow^{d+d'} C$  if and only if there is a  $C''$  such that  $C \rightsquigarrow^d C''$  and  $C'' \rightsquigarrow^{d'} C$ .

## 4 Timed Equivalences

Behavioural equivalence should be used to compare systems behaviour; whenever two systems are shown to be identical, no observer or context can distinguish between them. A good behavioural equivalence guarantees that, in any context, a system can be safely replaced by an equivalent system, thus allowing compositional reasoning. A suitable notion of equivalence between timed systems is obtained by extending the standard notion of bisimilarity to take into account timed transitions [7].

The notions defined in this section are generally applicable to all formalisms involving time than can be encoded in TLTS. Since in the previous section we defined a specific TLTS for real-time P systems, we use their corresponding behavioural equivalences to compare membrane systems (e.g., as in Example 1).

**Definition 5.** Let  $(\mathcal{C}_{\Pi_1}, C_{01}, \mathcal{L}_1, \rightarrow, \rightsquigarrow)$  and  $(\mathcal{C}_{\Pi_2}, C_{02}, \mathcal{L}_2, \rightarrow, \rightsquigarrow)$  be two TLTS. A binary relation  $\mathcal{R} \subseteq \mathcal{C}_{\Pi_1} \times \mathcal{C}_{\Pi_2}$  is called a **strong timed simulation (ST simulation)** if whenever  $(C, D) \in \mathcal{R}$ , then:

1. for any  $r \in \mathcal{L}$ ,  $C' \in \mathcal{C}_{\Pi_1}$ , if  $C \xrightarrow{r} C'$ , then there exists some  $D' \in \mathcal{C}_{\Pi_2}$  such that  $D \xrightarrow{r} D'$  and  $(C', D') \in \mathcal{R}$ ;
2. for any  $d \in \mathbb{R}_+$ ,  $C' \in \mathcal{C}_{\Pi_1}$ , if  $C \xrightarrow{d} C'$ , then there exists some  $D' \in \mathcal{C}_{\Pi_2}$  such that  $D \xrightarrow{d} D'$  and  $(C', D') \in \mathcal{R}$ .

If  $\mathcal{R}$  and  $\mathcal{R}^{-1}$  are strong timed simulations, then  $\mathcal{R}$  is called a **strong timed bisimulation** (ST bisimulation). We define **strong timed bisimilarity** by

$$\sim \stackrel{\text{def}}{=} \{(C, D) \in \mathcal{C}_{\Pi_1} \times \mathcal{C}_{\Pi_2} \mid \text{there exists a ST bisimulation } \mathcal{R} \text{ and } (C, D) \in \mathcal{R}\}$$

The definition treats timed transitions as rule transitions and thus the strong timed bisimilarity coincides with the original notion of bisimilarity over two labelled transition systems  $(\mathcal{C}_{\Pi_1}, C_{01}, \mathcal{L}_1 \cup \mathbb{R}_+, \rightarrow \cup \rightsquigarrow)$  and  $(\mathcal{C}_{\Pi_2}, C_{02}, \mathcal{L}_2 \cup \mathbb{R}_+, \rightarrow \cup \rightsquigarrow)$ .

*Remark 1.*  $\sim$  is an equivalence relation and the largest ST bisimulation.

#### 4.1 Bounded Timed Equivalence

Strong timed bisimilarity is too strong since all behaviours that violate time constraints are considered failures. An alternative is to weaken comparison criteria to behaviours up to a given deadline, ignoring the behaviours after the deadline.

*Example 2.* Consider the following systems:

$\Pi_1 = (a \times (\mathbb{R} \cup \infty), \{1 \times \infty, 2 \times (\mathbb{R} \cup \infty)\}, \{(1, \infty), (2, 5)\}, \emptyset, (a, 2), \{r\}, \emptyset, (\{r : (a, 0) \rightarrow (a, 6)\}))$  and

$\Pi_2 = (a \times (\mathbb{R} \cup \infty), \{1 \times \infty, 2 \times (\mathbb{R} \cup \infty)\}, \{(1, \infty), (2, 7)\}, \emptyset, (a, 4), \{r\}, \emptyset, (\{r : (a, 0) \rightarrow (a, 6)\}))$ .

The initial configuration for  $\Pi_1$  is  $C_1^0 = (\{(1, \infty), (2, 5)\}, \emptyset, (a, 2))$ , while for  $\Pi_2$  is  $C_2^0 = (\{(1, \infty), (2, 7)\}, \emptyset, (a, 4))$ . Before time 2, both  $\Pi_1$  and  $\Pi_2$  have exactly the same evolutions. If we don't care about the behaviour of the systems beyond time 2 it makes sense to identify the two systems up to time 2. Nevertheless, these systems cannot be identified by strong timed bisimulation. We can see that  $\Pi_1$  has the following evolution:

$$C_1^0 \xrightarrow{2} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 0)) \xrightarrow{r} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 6))$$

but this cannot be matched by the evolution of  $\Pi_2$ :

$$C_2^0 \xrightarrow{2} (\{(1, \infty), (2, 5)\}, \emptyset, (a, 2)) \xrightarrow{2} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 0)) \xrightarrow{r} (\{(1, \infty), (2, 3)\}, \emptyset, (a, 6)).$$

Hence,  $\Pi_1$  and  $\Pi_2$  cannot be identified by strong timed bisimulation. We need a notion of equivalence that allows us to identify systems whose behaviours match up to a given deadline.

A notion of *timed bisimilarity up to time  $t$*  is introduced in [18] to compare the behaviour of timed CSP processes. This notion can be applied to any pair of TLTS and thus to our formalism. In order to define an *equivalence up to time  $t$*  we need the following terminology:

A **binary relation** over  $\mathcal{C}_{\Pi 1}$  and  $\mathcal{C}_{\Pi 2}$  is a relation  $\mathcal{R} \subseteq \mathcal{C}_{\Pi 1} \times \mathcal{C}_{\Pi 2}$ , where  $\mathcal{C}_{\Pi 1}$  and  $\mathcal{C}_{\Pi 2}$  can be equal. The **identity relation** is  $\text{id} \stackrel{\text{def}}{=} \{(C, C) \mid C \in \mathcal{C}_{\Pi 1} \cap \mathcal{C}_{\Pi 2}\}$ . The **inverse of a relation**  $\mathcal{R}$  is  $\mathcal{R}^{-1} \stackrel{\text{def}}{=} \{(D, C) \mid (C, D) \in \mathcal{R}\}$ . The **composition of relations**  $\mathcal{R}_1$  and  $\mathcal{R}_2$  is  $\mathcal{R}_1 \mathcal{R}_2 \stackrel{\text{def}}{=} \{(C, C'') \mid \exists C' \in \mathcal{C}_{\Pi 1} \cap \mathcal{C}_{\Pi 2} \text{ such that } (C, C') \in \mathcal{R}_1 \wedge (C', C'') \in \mathcal{R}_2\}$ .

**Definition 6.** The binary relation  $\mathcal{R}_t$ ,  $t \in \mathbb{R}_+$ , over  $\mathcal{C}_{\Pi 1}$  and  $\mathcal{C}_{\Pi 2}$  is called a **strong time-bounded simulation** (STB simulation) if whenever  $(C, D) \in \mathcal{R}_t$ , then:

1. for any  $r \in \mathcal{L}$ ,  $C' \in \mathcal{C}_{\Pi 1}$ , if  $C \xrightarrow{r} C'$ , then there exists  $D' \in \mathcal{C}_{\Pi 2}$  such that  $D \xrightarrow{r} D'$  and  $(C', D') \in \mathcal{R}_t$ ;
2. for any  $d \in \mathbb{R}_+$ ,  $d < t$ ,  $C' \in \mathcal{C}_{\Pi 1}$ , if  $C \xrightarrow{d} C'$ , then there exists  $D' \in \mathcal{C}_{\Pi 2}$  such that  $D \xrightarrow{d} D'$  and  $(C', D') \in \mathcal{R}_{t-d}$ .

If  $\mathcal{R}_t$  and  $\mathcal{R}_t^{-1}$ ,  $t \in \mathbb{R}_+$  are STB simulations, then  $\mathcal{R}_t$  is called a **strong time-bounded bisimulation** (STB bisimulation). We define **STB bisimilarity** by

$$\simeq_t \stackrel{\text{def}}{=} \{(C, D) \in \mathcal{C}_{\Pi 1} \times \mathcal{C}_{\Pi 2} \mid \text{for } t \in \mathbb{R}_+ \text{ there exists a STB bisimulation } \mathcal{R}_t \text{ and } (C, D) \in \mathcal{R}_t\}$$

We also define the union of all STB bisimilarities  $\simeq_t$ , as

$$\simeq = \bigcup_{t \in \mathbb{R}_+} \simeq_t$$

The first clause states that the derived configurations are matched up to the same time  $t$ . The second clause states that the derived configurations are matched up to time  $t - d$ , namely when they advance in time (by  $d$  time units), the bound is reduced accordingly.

Now let us come back to Example 1.

*Example 3.* We have that  $C_1^0 \simeq_2 C_2^0$ , as the two configurations  $C_1^0$  and  $C_2^0$  have a timed transition at any time  $d < 2$ , namely  $C_1^0 \xrightarrow{d} C_1^d$  and  $C_2^0 \xrightarrow{d} C_2^d$ . Note that bisimilarity up to time 2 does not include bisimilarity at time 2 since  $C_1^0 \xrightarrow{2} C_1^2$  and  $C_2^0 \xrightarrow{2} C_2^2$ , but  $C_2^0 \xrightarrow{2} C_2^2$  and  $C_2^2 \not\xrightarrow{2} C_2^2$ .

This bisimilarity up to time  $t$  satisfies the following property that states how equivalence up to a deadline  $t$  includes equivalence up to any bound  $u \leq t$ .

**Proposition 2.** For any TLTS  $(\mathcal{C}_{\Pi 1}, C_{01}, \mathcal{L}_1, \rightarrow, \rightsquigarrow)$  and  $(\mathcal{C}_{\Pi 2}, C_{02}, \mathcal{L}_2, \rightarrow, \rightsquigarrow)$ ,  $t, u \in \mathbb{R}_+$ ,  $C \in \mathcal{C}_{\Pi 1}$ ,  $C' \in \mathcal{C}_{\Pi 1} \cap \mathcal{C}_{\Pi 2}$  and  $C'' \in \mathcal{C}_{\Pi 2}$ :

1. If  $C \simeq_t C''$  then for any  $u \leq t$ ,  $C \simeq_u C''$ .
2. If  $C \simeq_t C'$  and  $C' \simeq_u C''$  then  $C \simeq_{\min\{t, u\}} C''$ .

Furthermore, we also have the following properties.

**Proposition 3.** For any TLTS  $(\mathcal{C}_{\Pi 1}, C_{01}, \mathcal{L}_1, \rightarrow, \rightsquigarrow)$  and  $(\mathcal{C}_{\Pi 2}, C_{02}, \mathcal{L}_2, \rightarrow, \rightsquigarrow)$ ,



1.  $\simeq$  is a STB bisimulation.
2.  $\simeq$  is closed to identity, inverse, composition and union.
3.  $\simeq$  is the largest STB bisimulation.
4.  $\simeq$  is an equivalence.

*Proof (Sketch).*

1. Assume that  $C \simeq D$ . By definition of  $\simeq$ , there must be a STB bisimulation  $\simeq_t$  such that  $(C, D) \in \simeq_t$ . We need to check that  $\simeq$  and  $\simeq^{-1}$  satisfy the conditions of STB simulations.
2. (a) The identity timed relation  $\text{id}$  is a STB bisimulation because any  $C$  can match its own transitions up to any time  $t$ .  
 (b) Since  $\simeq$  is a STB bisimulation, the inverse  $\simeq^{-1}$  also is one by definition.  
 (c) If  $\mathcal{R}_1$  and  $\mathcal{R}_2$  are STB bisimulations  $\simeq$ , then we need to show that their composition  $\mathcal{R}_1 \mathcal{R}_2$  is a STB bisimulation as well.  
 (d) Finally, we obtain that the union  $\cup_{i \in I} \mathcal{R}_i$  of STB bisimulations  $\mathcal{R}_i$  is a STB bisimulation, as follows. Let  $(C, D) \in \cup_{i \in I} \mathcal{R}_i \cap \simeq_t$ . Therefore, for some  $i \in I$ ,  $(C, D) \in \mathcal{R}_i \cap \simeq_t$ . If  $C \xrightarrow{r} C'$  then  $D \xrightarrow{r} D'$  and  $(C', D') \in \mathcal{R}_i \cap \simeq_t$  and therefore,  $(C', D') \in \cup_{i \in I} \mathcal{R}_i \cap \simeq_t \subseteq \cup_{i \in I} \mathcal{R}_i$ . Similarly, if  $C \xrightarrow{d} C'$  for some  $d < t$  then  $D \xrightarrow{d} D'$  with  $(C', D') \in \mathcal{R}_i \cap \simeq_t$  and so  $(C', D') \in \cup_{i \in I} \mathcal{R}_i \cap \simeq_t \subseteq \cup_{i \in I} \mathcal{R}_i$  as required.
3. Suppose that there is a STB bisimulation  $\mathcal{R} = \bigcup_{t \in \mathbb{R}_+} \mathcal{R}_t$  larger than  $\simeq$ , i.e.  $\simeq \subsetneq \mathcal{R}$ . So there are  $C, D$  and  $t$  for which  $(C, D) \in \mathcal{R}_t$  while  $C \not\simeq_t D$ . But  $C \not\simeq_t D$  is only possible if there is no STB bisimulation  $\mathcal{R}_t$  that contains  $(C, D)$ , contradicting the assumption.
4. To show that  $\simeq$  is an equivalence, we should prove that  $\simeq$  is reflexive, symmetric and transitive.

□

## 4.2 Bounded Timed Bisimulation “up to” Techniques

In what follows we provide some techniques that extend the “up to” techniques from [17] to the context of bounded timed bisimulations. The standard proof technique to establish that  $C_1$  and  $C_2$  are bisimilar is to find a bisimulation  $\mathcal{R}$  such that  $(C_1, C_2) \in \mathcal{R}$  and  $\mathcal{R}$  is closed under transitions: the derivatives  $(C'_1, C'_2)$  of  $(C_1, C_2)$  are also in  $\mathcal{R}$ . Since such derivatives are added to  $\mathcal{R}$  without the possibility of manipulating them, a bisimulation relation often contains many strongly related pairs. As an example, a bisimulation relation might contain pairs of configurations obtainable from other pairs through application of algebraic laws (e.g., composition). These redundancies can make both the definition and the verification of a bisimulation relation annoyingly heavy and tedious. This means that sometimes is difficult to find directly such a relation  $\mathcal{R}$ .

A property that we naturally expect to hold is that symbols which do not appear in the left-hand side of the evolution rules, do not influence the evolution of real-time P systems when added to the initial configurations. Consider the configuration  $C_0 = (\mu_t, w_0, w_1)$ , and two objects  $(b, 3)$  and  $(c, 3)$  that appear

only in time passing rules. To prove that  $C_{01} = (\mu_t, w_0(b, 3), w_1)$  and  $C_{012} = (\mu_t, w_0(c, 3), w_1)$  are bisimilar, we would like to use the binary relation

$$\mathcal{R} \stackrel{def}{=} \{(C_{01}, C_{02})\}$$

But, according to Definition 6,  $\mathcal{R}$  is not a bisimulation relation. If we add pairs of configurations to  $\mathcal{R}$  in order to turn it into a bisimulation relation, then we might find that the simplest solution is to take the relation

$$\mathcal{R}' \stackrel{def}{=} \{(C, D) \mid C \simeq C_{01}, D \simeq C_{02}\}$$

The size of  $\mathcal{R}'$  is rather discouraging. However, this extension is unnecessary since the bisimilarity between the two configurations in  $\mathcal{R}$  already implies the bisimilarity between the configurations of all pairs of  $\mathcal{R}'$ . The notions defined in the current section aim to simplify the bisimulation proof method. The new technique would allow for the above example to prove that  $C_{01}$  and  $C_{02}$  are bisimilar simply using the relation  $\mathcal{R}$ . In this sense, we generalise the bisimulation proof method by relaxing Definition 6. There is an useful alternative technique, the so-called bisimulation “up to” some relation  $\mathcal{R}'$ : for a relation  $\mathcal{R}$ , which is not a bisimulation, if  $(C_1, C_2) \in \mathcal{R}$ , then the derivatives  $(C'_1, C'_2)$  are in  $\mathcal{R}'$ . Under certain conditions we can establish that  $C_1$  and  $C_2$  are bisimilar. For this technique, a general framework that works on untimed labelled transition systems is presented in [16]. We cannot use directly that framework, but the framework can be extended to timed labelled transition systems. We begin by introducing a notion of a timed relation “progressing” towards another timed relation:

**Definition 7.** Let  $(\mathcal{C}_{\Pi 1}, C_{01}, \mathcal{L}_1, \rightarrow, \rightsquigarrow)$  and  $(\mathcal{C}_{\Pi 2}, C_{02}, \mathcal{L}_2, \rightarrow, \rightsquigarrow)$  be two TLTS and let  $\mathcal{R}_t, \mathcal{R}'_t, t \in \mathbb{R}_+$  be any timed relations. We say that  $\mathcal{R}_t$  **strongly progresses** to  $\mathcal{R}'_t$ , written  $\mathcal{R}_t \mapsto \mathcal{R}'_t$ , if for any  $C, D \in \mathcal{C}_{\Pi}$ , whenever  $(C, D) \in \mathcal{R}_t$ , then:

1. for any  $r \in \mathcal{L}$ ,  $C' \in \mathcal{C}_{\Pi 1}$ ,  $C \xrightarrow{r} C'$ , then there exists  $D' \in \mathcal{C}_{\Pi 2}$  such that  $D \xrightarrow{r} D'$  and  $(C', D') \in \mathcal{R}'_t$ ;
2. for any  $d \in \mathbb{R}_+$ ,  $d < t$ ,  $C' \in \mathcal{C}_{\Pi 1}$ ,  $C \rightsquigarrow^d C'$ , then there exists  $D' \in \mathcal{C}_{\Pi 2}$  such that  $D \rightsquigarrow^d D'$  and  $(C', D') \in \mathcal{R}'_{t-d}$ .

The definition is similar to that of STB bisimulation, except that the derivatives  $(C', D')$  must be in  $\mathcal{R}'_{t'}$  rather than in  $\mathcal{R}_{t'}$ . In fact, STB bisimulation can be seen as a specific case:

*Remark 2.*  $\mathcal{R}$  is a STB bisimulation if and only if  $\mathcal{R} \mapsto \mathcal{R}$ .

**Proposition 4.** If  $\mathcal{R}_t \mapsto \mathcal{R}'_t$  and  $\mathcal{R}'_t$  is a STB bisimulation, then  $\mathcal{R}_t \subseteq \simeq$ .

Hence, to establish that  $C \simeq_t D$  it is enough to find a relation  $\mathcal{R}_t$  with  $(C, D) \in \mathcal{R}_t$  which strongly progresses to a known STB bisimulation  $\mathcal{R}'_t$ . The choice of  $\mathcal{R}'_t$  depends on the particular equivalence we are trying to establish. One of the most common cases is when  $\mathcal{R}'_t = \simeq_t$ . However, in general we may not have at hand a relation  $\mathcal{R}'_t$  known to be a bisimulation. Nevertheless we may find that  $\mathcal{R}_t$  progresses to a relation  $\mathcal{R}'_t = \mathcal{F}(\mathcal{R}_t)$  for some function  $\mathcal{F}$  on relations.

The idea is that if  $\mathcal{R}_t$  progresses to  $\mathcal{F}(\mathcal{R}_t)$  and  $\mathcal{F}$  satisfies certain conditions, then  $\mathcal{R}_t$  is included in  $\simeq_t$ . Thus, to establish  $C \simeq_t D$  we need to find such  $\mathcal{F}$  and  $\mathcal{R}_t$  containing  $(C, D)$ .

Before we can characterise  $\mathcal{F}$  functions, we need the following:

**Proposition 5.** *Let  $\mathcal{R}_t, \mathcal{R}'_t, \mathcal{R}''_t, \mathcal{R}'''_t \subseteq C_\Pi \times C_\Pi$ ,  $t \in \mathbb{R}_+$  be some timed relations.*

1. *If  $\mathcal{R}_t \subseteq \mathcal{R}'_t$  and  $\mathcal{R}'_t \mapsto \mathcal{R}''_t$  then  $\mathcal{R}_t \mapsto \mathcal{R}''_t$ .*
2. *If  $\mathcal{R}_t \mapsto \mathcal{R}'_t$  and  $\mathcal{R}'_t \subseteq \mathcal{R}''_t$  then  $\mathcal{R}_t \mapsto \mathcal{R}''_t$ .*
3. *If  $\mathcal{R}_t \mapsto \mathcal{R}''_t$  and  $\mathcal{R}'_t \mapsto \mathcal{R}'''_t$  then  $\mathcal{R}_t \mathcal{R}'_t \mapsto \mathcal{R}''_t \mathcal{R}'''_t$ .*

*Proof (Sketch).*

1. Assume  $\mathcal{R}_t \subseteq \mathcal{R}'_t$  and  $\mathcal{R}'_t \mapsto \mathcal{R}''_t$ . Let  $(C, D) \in \mathcal{R}_t$ . Then  $(C, D) \in \mathcal{R}'_t$  and since  $\mathcal{R}'_t \mapsto \mathcal{R}''_t$ , we have that
  - (a)  $C \xrightarrow{r} C'$  implies  $D \xrightarrow{r} D'$  with  $(C', D') \in \mathcal{R}''_t$ , (and vice-versa)
  - (b)  $C \xrightarrow{d} C'$  for  $d < t$  implies  $D \xrightarrow{d} D'$  with  $(C', D') \in \mathcal{R}''_{t-d}$  (and vice-versa)
 This shows that  $\mathcal{R}_t \mapsto \mathcal{R}''_t$ .

The other two cases are treated in a similar manner.  $\square$

**Proposition 6.** *Let  $\{\mathcal{R}_i\}_{i \in I}$  and  $\{\mathcal{R}'_j\}_{j \in J}$  be two sets of timed relations, and define the relations  $\mathcal{R} \stackrel{\text{def}}{=} \bigcup_{i \in I} \mathcal{R}_i$  and  $\mathcal{R}' \stackrel{\text{def}}{=} \bigcup_{j \in J} \mathcal{R}'_j$ . Then*

1. *If for each  $i \in I$  there is a  $j \in J$  such that  $\mathcal{R}_i \mapsto \mathcal{R}'_j$ , then  $\mathcal{R} \mapsto \mathcal{R}'$ .*
2. *If for each  $i \in I$  there is a  $i' \in I$  such that  $\mathcal{R}_i \mapsto \mathcal{R}_{i'}$ , then  $\mathcal{R}$  is a STB bisimulation.*

These functions on relations are characterised as follows:

**Definition 8 (Safe functions).** *A function  $\mathcal{F}$  on timed relations is sound if for any timed relations  $\mathcal{R}$ , if  $\mathcal{R} \mapsto \mathcal{F}(\mathcal{R})$  then  $\mathcal{R} \subseteq \simeq$ .*

However, using this definition it is hard to check whether a function is safe. An example of a function that is not safe is the function that maps every relation to the universal relation  $C_\Pi \times C_\Pi$ . In what follows we define a class of sound functions for which membership is easy to check. We define strongly safe functions:

**Definition 9 (Strongly safe functions).** *A function  $\mathcal{F}$  on timed relations is strongly safe if for any timed relations  $\mathcal{R}, \mathcal{R}'$ , if  $\mathcal{R} \subseteq \mathcal{R}'$  and  $\mathcal{R} \mapsto \mathcal{R}'$  then  $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}(\mathcal{R}')$  and  $\mathcal{F}(\mathcal{R}) \mapsto \mathcal{F}(\mathcal{R}')$ .*

**Proposition 7.** *The following functions are safe:*

1.  $\mathcal{F}_{\text{id}}(\mathcal{R}) \stackrel{\text{def}}{=} \mathcal{R}$ ;
2.  $\mathcal{F}_{\text{id}}(\mathcal{R}) \stackrel{\text{def}}{=} \text{id}$ ;
3.  $\mathcal{F}_{\simeq}(\mathcal{R}) \stackrel{\text{def}}{=} \simeq$ .

The main property, the core of the technique, is the following:

**Lemma 1.** *If  $\mathcal{R} \mapsto \mathcal{F}(\mathcal{R})$  for some strongly safe function  $\mathcal{F}$  then  $\mathcal{R} \subseteq \simeq$  and  $\mathcal{F}(\mathcal{R}) \subseteq \simeq$ .*

This technique relies on having a safe function  $\mathcal{F}$  on timed relations. We have given a few basic safe functions  $\mathcal{F}_{\text{id}}$ ,  $\mathcal{F}_{\text{id}}$  and  $\mathcal{F}_{\simeq}$ , but often these are not enough. As shown in [16, 17], it is possible to start from basic safe functions and build more complex ones. The following proposition gives us some basic operators on these functions which preserve the safety property.

**Proposition 8.** *If  $\mathcal{F}_i$ ,  $\mathcal{F}$  and  $\mathcal{G}$  are strongly safe, so are  $\cup_{i \in I} \mathcal{F}_i$ ,  $\mathcal{F} \circ \mathcal{G}$  and  $\mathcal{F}\mathcal{G}$ , where  $(\cup_{i \in I} \mathcal{F}_i)(\mathcal{R}) \stackrel{\text{def}}{=} \cup_{i \in I} \mathcal{F}_i(\mathcal{R})$ ,  $(\mathcal{F} \circ \mathcal{G})(\mathcal{R}) \stackrel{\text{def}}{=} \mathcal{F}(\mathcal{G}(\mathcal{R}))$  and  $(\mathcal{F}\mathcal{G})(\mathcal{R}) \stackrel{\text{def}}{=} \mathcal{F}(\mathcal{R})\mathcal{G}(\mathcal{R})$ .*

This gives us some commonly used safe functions:

**Lemma 2.** *The following functions are safe:*

1.  $\mathcal{F}_{\text{ius}} \stackrel{\text{def}}{=} \mathcal{F}_{\text{id}} \cup \mathcal{F}_{\simeq};$
2.  $\mathcal{F}_{\text{sis}} \stackrel{\text{def}}{=} \mathcal{F}_{\simeq} \mathcal{F}_{\text{id}} \mathcal{F}_{\simeq}.$

For example using  $\mathcal{F}_{\text{ius}}$  we can prove  $C \simeq_t D$  by finding a relation  $\mathcal{R}_t$  containing  $(C, D)$  which progresses to  $\mathcal{F}_{\text{ius}}(\mathcal{R}_t) = \mathcal{R}_t \cup \simeq$ , namely its derivatives  $(C', D')$  are in either  $\mathcal{R}_t$  or in  $\simeq$ . Another common example is using  $\mathcal{F}_{\text{sis}}(\mathcal{R}_t) = \simeq \mathcal{R}_t \simeq$ . In this case, proving  $C \simeq_t D$  requires finding an  $\mathcal{R}_t$  containing  $(C, D)$  which progresses to  $\simeq \mathcal{R}_t \simeq$ , namely for its derivatives  $(C', D')$  there are  $C_1$  and  $C_2$  such that  $C' \simeq_{t'} C_1$ ,  $(C_1, C_2) \in \mathcal{R}_{t'}$  and  $C_2 \simeq_{t'} D'$ .

## 5 Conclusion

In this paper we proposed a real-time extension of P systems, in which the semantics is given by two types of rules: the transition relation  $\rightarrow$  describes rule application and a sequence of such transitions describes an execution within the same instant of real-time, whereas the timed transition relation  $\rightsquigarrow$  describes the passage of real-time.

Over this class of membrane systems we have defined timed bounded equivalences using timed labelled transition systems, and an extended Sangiorgi “up to” technique. An important goal of defining these bisimulations is to offer flexibility in selecting the right one when verifying biological systems and comparing them. A good behavioural equivalence guarantees that, in any context, a membrane system can be safely replaced by an equivalent membrane system.

**Acknowledgements.** Many thanks to the reviewers for their useful comments. The work was supported by a grant of the Romanian National Authority for Scientific Research, project number PN-II-ID-PCE-2011-3-0919.

## References

1. B. Aman, G. Ciobanu. Adding Lifetime to Objects and Membranes in P Systems. *International Journal of Computers, Communication & Control*, vol.V (2010) 268–279.
2. O. Andrei, G. Ciobanu, D. Lucanu. A Rewriting Logic Framework for Operational Semantics of Membrane Systems. *Theoretical Computer Science*, vol.373 (2007) 163–181.
3. J. Bachman, P. Sorger. New Approaches to Modelling Complex Biochemistry. *Nature Methods*, vol.8 (2011) 130–131.
4. R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, S. Tini. Compositional Semantics and Behavioural Equivalences for P Systems. *Theoretical Computer Science*, vol.395 (2008) 77–100.
5. C.S. Calude, Gh. Păun. Bio-Steps Beyond Turing. *Biosystems*, vol.77 (2004) 175–194.
6. M. Cavaliere, D. Sburlan. Time and Synchronization in Membrane Systems. *Fundamenta Informaticae*, vol.64 (2005) 65–77.
7. G. Ciobanu. Behaviour Equivalences in Timed Distributed  $\Pi$ -Calculus. *Lecture Notes in Computer Science* vol.5380 (2008) 190–208.
8. G. Ciobanu. Semantics of P Systems. *The Oxford Handbook of Membrane Computing* (2010) 413–436.
9. G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez (Eds.). *Applications of Membrane Computing*. Springer, Natural Computing Series, 2006.
10. I. Eidhammer, I. Jonassen, W. Taylor. Structure Comparison and Structure Patterns. *Journal of Computational Biology*, vol.7 (2000) 685–716.
11. R. Freund, M. Ionescu, M. Oswald. Extended Spiking Neural P Systems with Decaying Spikes and/or Total Spiking. *International Journal of Foundations of Computer Science*, vol.19 (2008) 1223–1234.
12. D.E. Leocadio, A.R. Kunselman, T. Cooper, J.H. Barrantes, J.C. Trussell. Anatomical and Histological Equivalence of the Human, Canine, and Bull Vas Deferens. *The Canadian Journal of Urology*, vol.18 (2011) 5699–5704.
13. Gh. Păun. *Membrane Computing. An Introduction*, Springer, 2002.
14. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
15. D. Sangiorgi. A Theory of Bisimulation for the  $\Pi$ -Calculus. *Lecture Notes in Computer Science*, vol.715 (1993) 127–142.
16. D. Sangiorgi. On the Bisimulation Proof Method. *Journal of Mathematical Structures in Computer Science*, vol.8 (1998) 447–479.
17. D. Sangiorgi and D. Walker. *The  $\pi$ -calculus: A Theory of Mobile Processes*, Cambridge University Press, 2001.
18. S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, vol.116 (1995) 193–213.



# The Geometric Membrane Structure of Finite Interactive Systems Scenarios

I.T. Banu-Demergian and G. Stefanescu

Department of Computer Science, University of Bucharest, Romania  
th\_iulia84@yahoo.com; gheorghe.stefanescu@fmi.unibuc.ro

**Abstract.** Finite interactive systems (FIS) are one of many equivalent presentations of regular 2-dimensional languages (2Reg). Known regular expressions for 2Reg (denoted 2RE) are based on intersection and renaming, giving little insight on the structure of regular languages. Recently, the authors have introduced a new type of regular expressions for 2-dimensional languages using arbitrary shapes and tiling operations parametrized by restrictions on the connection interfaces. Their result on the representation of FIS languages with this type of regular expressions (a Kleene theorem) is based on an unexpected connection between the shapes of scenarios in finite interactive systems and membrane systems. Compared with the usual P systems, these FIS-based membrane systems are more rigid (geometric) and they lack the dynamics; probably, the latter issue can be solved by adding a new dimension in the model, going to 3-dimensional shapes. The paper provides a detailed analysis of the FIS-based membrane systems.

**Keywords:** parallel programming, interactive programming, finite interactive systems, regular expressions, regular algebra, two-dimensional languages, scenarios, Kleene theorem, membrane computing, P systems

## 1 Introduction

Natural computing gathers a diversity of powerful and precise models of computation, sharing the same source of inspiration, the laws of nature. In particular, membrane computing [19–21], a recent branch of natural computing, takes its roots from the study of cellular membranes.

Various types of membrane systems, called P systems, have been proposed as distributed and parallel computing models and also related to other classic models of interactive computation. For instance, relationship with Petri nets has been studied in [23]; the possibility of having P systems generating regular languages has been investigated in [8]. It is proved that regular languages are inverse-morphic images of languages of finite spiking neural P systems, and recursively enumerable languages are projections of inverse-morphic images of languages generated by spiking neural P systems.

Kleene originally introduced regular expressions [13] in connection with neural networks and finite automata. Kleene theorem states that finite automata

and regular expressions are equivalent (i.e., they specify the same language). In the meantime, regular expressions became a core formalism for many other models of sequential computation. In particular, they provide the backbone of a rich algebraic theory of automata, see, e.g. [24, 9, 16, 14, 4, 15, 5]. For parallel computation, enrichment of the sequential models with mechanisms for modeling process interaction are needed. We only mention a Kleene theorem for Petri nets [22, 10]: Petri nets and a class of concurrent regular expressions are equivalent.

A robust class of “regular two dimensional languages” (2Reg) had been proposed in [11, 18]. It may be specified by many equivalent formalisms: a class of monadic second-order formulas; a type of cellular automata; a class of 2-dimensional regular expressions, using intersection and renaming (2RE); a tile system; etc.

In [7, 6] a type of tissue-like P systems with active membranes is used as a generative device for rectangular picture languages, arising from tiling in the integer plane  $\mathbb{Z}^2$ . The result of a computation is read from all membranes of the system, each membrane corresponding to a pixel of the picture. The communication graph of the system is a rectangular grid in which an inner node is connected with four neighbours, a marginal node with three neighbours and a corner node with two neighbours. The membrane labels, elements of  $(\mathbf{d} + \mathbf{r})^*$ , hold informations for generating the grid, using two evolution directions:  $\mathbf{d}$  (“down”) and  $\mathbf{r}$  (“right”).

A new representation for two-dimensional languages was introduced in [1]. It is based on contours and their composition (see also [2]). Roughly speaking, a *contour* is a closed, non-overlapping line over a  $\mathbb{Z}^2$  grid, formed by a list of oriented segments (“right”, “down”, “left”, “up”), starting at a particular point. The role of a contour is to describe a finite set of cells as its interior area. The region delimited by a contour holds a *general 2-dimensional word*, the cells being filled with letters from a given alphabet. The operation of merging two adjacent contours is called *contour composition*. Based on contour composition a new type of regular expressions for two dimensional languages n2RE was introduced in [1]. In this new formalism n2RE, the intersection and renaming of 2RE are replaced with a powerful tiling system that builds the words step by step, being closer in spirit with classical 1-dimensional regular expressions.

The question of finding an n2RE-like representation of FIS languages has been addressed in [3]. It was shown that a slight extension x2RE of n2RE, considering compositions involving the extreme cells of the words, is powerful enough to represent FIS languages. The main technical result is a representation theorem for scenarios in finite interactive systems which shows that FIS scenarios have a rich and unexpected recursive membrane structure.

In this paper we explore the FIS-based membrane structures in connection with P systems. Compared with the usual P systems, these FIS-based membrane systems are more rigid (geometric). A more important difference is that they lack the dynamics. However, this latter issue probably can be solved by adding a new dimension in the model, going to 3-dimensional shapes. The paper provides a



detailed analysis of the scenarios and the membrane system corresponding to a small FIS representing Pascal triangle words.

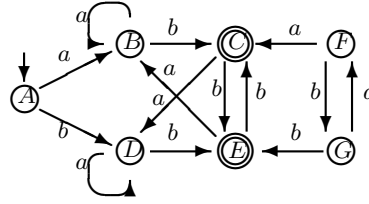
The paper is organized as follows. Section 2 briefly reviews basic informations on finite automata, regular expressions, finite interactive systems and classical 2-dimensional regular expressions. Section 3 gives a detailed presentation of the new type of regular expressions and Section 4 describes the recursive membrane tiling mechanism for recognizing complete 2x2 FISs languages. Section 5 addresses the question on the relationship with P systems. Related and future works and references conclude the paper.

## 2 Preliminaries

In this section we introduce the basic definitions and results regarding finite automata and finite interactive systems (the latter is an extension of finite automata in two dimensions).

### 2.1 Finite automata and regular expressions

*Finite automata.* To describe the informal model of finite state automata let us consider the example given in Fig. 1.



**Fig. 1.** A (complete, deterministic) finite state automaton

Such an automaton is used to recognise words. A word over an alphabet  $V$  is accepted if starting from an initial state (a state with a small incoming arrow) and following transitions associated to the letters in the word eventually a final state is reached (a state bearing a double-circle notation). For instance, the word  $bbab$  is accepted, while  $bbaaa$  is not accepted. An automaton is complete if in each state there is at least one outgoing arrow for each element in  $V$ . An automaton is deterministic if in each state there is at most one outgoing arrow for each element in  $V$ .

*Regular expressions.*

**Definition 1.** (regular expressions [13]) Regular expressions over an alphabet  $V$  are obtained applying the rules (1) and (2) below a finite number of times:

- (1)  $0, 1$  and  $a(\in V)$  are regular expressions;
- (2) if  $E, F$  are regular expressions, then  $E + F, E \cdot F$  and  $E^*$  are regular expressions.

The *language (or event)*  $|E|$ , specified by a regular expression  $E$ , is inductively defined as follows:  $|0| = \emptyset$ ;  $|1| = \{\lambda\}$ ;  $|a| = \{a\}$ ;  $|E + F| = |E| \cup |F|$ ;  $|E \cdot F| = |E| \cdot |F|$ ;  $|E^*| = |E|^*$ .  $\square$

The relationship between regular languages and regular expressions is given by the following fundamental theorem of Kleene.

**Theorem 1.** (Kleene, 1956) *A language is regular (e.g., it is recognized by a nondeterministic finite automaton) iff it may be specified by a regular expression.*

An algebraic proof for this result can be found in [26]. It is based on an algebraic proof of the following identity:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^* = \begin{bmatrix} a^* + a^*bwca^* & a^*bw \\ wca^* & w \end{bmatrix}$$

where  $w = (ca^*b + d)^*$ .

## 2.2 Finite interactive systems (FISs)

Finite interactive systems are a two-dimensional extension of finite automata, well suited for recognising two-dimensional words.

A *finite interactive system* [27, 28] is defined by:

- a set  $S$  of *states* (denoted by numbers  $1, 2, \dots$ ) and a set  $C$  of *classes* (denoted by capital letters  $A, B, \dots$ );
- a set  $T$  of *transitions* of the following form:  $(A, 1) \rightarrow a \rightarrow (B, 2)$ , where  $a$  is a letter of a given alphabet  $\Sigma$  and  $A, B, 1, 2$  are as above;
- specification of the *initial/final* states and classes.

A FIS is complete if it specifies a transition  $(c1, s1) \rightarrow t \rightarrow (c2, s2)$  for any pair  $((c1, s1), (c2, s2))$  in  $((C \times S) \times (C \times S))$ .

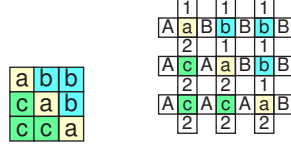
A useful *cross/tile representation* may be used; it is based on showing the transitions and stating which states and classes are initial/final. An example is

$$S1: \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline A & a & B \\ \hline 2 & & 1 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 2 & & 1 \\ \hline A & c & A \\ \hline 2 & & 1 \\ \hline \end{array} \quad \begin{array}{|c|c|c|} \hline 1 & & 2 \\ \hline B & b & B \\ \hline 1 & & 2 \\ \hline \end{array} \quad \text{with } 1, A \text{ initial and } 2, B \text{ final.}$$

The *FISs recognizing procedure* is via accepted scenarios. A *scenario* alternates class/state information and letters according to the transitions. It is an *accepting scenario* if the northern border has initial states, the western border has initial classes, the eastern border has final classes, and the southern border has final states.

Graphically, a scenario may be easily obtained using the crosses representing the transitions and identifying the matching classes or states of the neighbouring cells. Below, we show a few examples of scenarios for the FIS  $S1$  above.

word accepting  $S1$  scenario



word not-accepting  $S1$  scenario  
(on south, 1 is not final)



### 2.3 A known class of 2-dimensional regular expressions - 2RE

We start with “classical” 2-dimensional regular expressions (2RE). They are introduced below into two stages.

First, *simple 2-dimensional regular expressions* (simple 2RE) are defined by two sets of regular operators (one for the vertical direction, the other for the horizontal direction) which share the additive part. Formally, they use:

1. the *additive operators*:  $\emptyset$  (for *empty set*) and  $+$  (for *union*)
2. the *vertical composition operators*:
  - $I_v$  (*vertical identity*),  $;$  (*vertical composition*) and  $*_v$  (*iterated vertical composition*)
  - our preferred textual notations are:  $|$ ,  $;$  and  $*$
3. the *horizontal composition operators*:
  - $I_h$  (*horizontal identity*),  $;$  (*horizontal composition*) and  $*_h$  (*iterated horizontal composition*)
  - our preferred textual notations are:  $-$ ,  $>$  and  $\wedge$

Next, *2-dimensional regular expressions* are obtained adding intersection and renaming to simple 2RE. Formally, they use the following additional operators

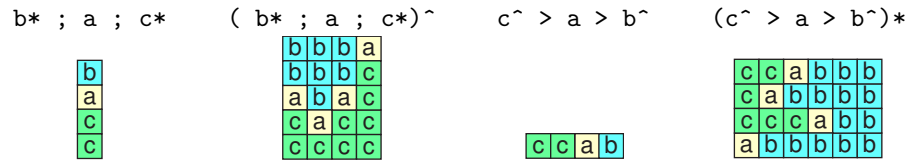
1. *intersection*: our preferred textual notation is  $\wedge$
2. *renaming* via a letter-to-letter homomorphism  $\rho: V \rightarrow V'$  ( $V$  and  $V'$  are the old and the new alphabets, respectively)

*Notation*: As usual, the set of words specified by an expression  $E$  is denoted by  $|E|$ .

A few examples of regular expressions and words satisfying these expressions are presented below. They are related to the following expression:

$$E = (b^* ; a ; c^*)^\wedge \wedge (c^\wedge > a > b^\wedge)^*.$$

Subexpressions and represented words:



It can be proved that the intersection has only square words with  $a$  on the

diagonal, b on the top right area and c on the bottom left area. To conclude,

$$(b^* ; a ; c^*)^{\wedge} /\ (c^{\wedge} > a > b^{\wedge})^* \text{ represents words as } \begin{array}{|c|c|c|c|} \hline a & b & b & b \\ \hline c & a & b & b \\ \hline c & c & a & b \\ \hline c & c & c & a \\ \hline \end{array}.$$

The first part of the expression constrains the column patterns, while the second the rows. Intersection does the magic action of selecting only the wished 2-dimensional words.

The connection between 2RE and FIS languages is formulated in the following theorem:

**Theorem 2.** ([11, 28, 25]) *The languages represented by finite interactive systems and those specified by 2-dimensional regular expressions 2RE are the same.*

The inconveniences of this approach mostly come from the use of renaming and intersection. For instance the 2RE formalism is not robust under renaming, i.e one can find two expression  $E1, E2$  over  $V1, V2$  respectively and a renaming  $\rho: V \rightarrow V'$  such as  $|E1| = |E2|$  but  $\rho(|E1|) \neq \rho(|E2|)$ . Notice that the expression  $E3 = (a^* ; a ; a^*)^{\wedge} /\ (a^{\wedge} > a > a^{\wedge})^*$ , obtained by syntactically renaming  $a, b, c$  as  $a$  into the expression  $E$  above, represents all rectangular words of  $a$ 's, not only the square ones as one expects. Moreover, intersection and renaming are difficult to handle and non-intuitive operations.

Hence we propose an alternative formalism based on a system of tiling shapes and a powerful set of composition operators for these shapes (extending vertical/horizontal compositions, and their iterated versions).

### 3 A new type of regular expressions - n2RE

This section is devoted to a new type of regular expressions for representing two-dimensional words.

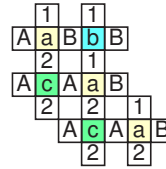
#### 3.1 General 2-dimensional words

A *general 2-dimensional word* is a set of unit cells in the 2-dimensional space filled with letters from the given alphabet. An example is below, also showing how the recognizing procedure may be extended to non-rectangular words.

a non-rectangular word



accepting  $S1$  scenario



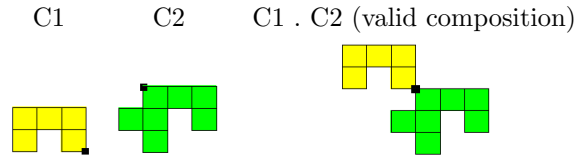
It is also possible to have words with several disconnected components.

### 3.2 General composition

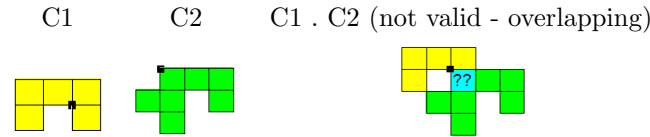
We define a *general composition* operator on 2-dimensional words as follows: given two words, get new words by putting them together such that no interior cell of the first word may overlap an interior cell of the other.

Actually this is a particular form of tiling. What is perhaps different here is that no restriction to have a physical contact between the words (via side borders or corners) is required.

For a graphical example, notice that



shows a valid composition, while



shows an example of composition leading to an invalid result (the result has overlapping areas). In this example, the black points (little black filled squares) on the contours show how the words are linked: identify these black points to get the composed words. This composition via identification of black points is extended to two-dimensional words as follows. For two words  $W1$ ,  $W2$ , consider arbitrary contours  $C1$ ,  $C2$  representing them (having as internal areas the shapes of the words) and arbitrary positions as black points (starting points for representing these contours). Then,  $W1 \cdot W2$  consists of all words resulting from valid compositions of such contours and placing the letters of the words  $W1$ ,  $W2$  in the corresponding positions of the resulting composites. E.g., the composite  $a \cdot a$

contains, among other, the following words ... More on contour representation of 2-dimensional words may be found in [1, 2].

### 3.3 Particular composition operators

The new type of 2-dimensional regular expressions, to be defined below, puts constraints on the connection points on the borders of the composed words. These constraints act on the following three types of elements: *side borders*, *land corners* (turning points on the border contour having 3 neighbouring cells outside the word and one neighbouring cell inside), and *golf corners* (turning points on the border contour with 3 neighbouring cells inside and one neighbouring cell outside). The resulting restricted composition operators extend the usual vertical and horizontal composition operators on rectangular words.

*Points of interest on the words borders.* A point is represented as a pair  $p = (x, y) \in \mathbb{Z}^2$ . A vertical line segment  $((x, y), (x, y + 1))$  is specified by its middle point  $l = (x, y + 0.5)$ ; similarly, a horizontal line segment  $((x, y), (x + 1, y))$  is denoted by  $l = (x + 0.5, y)$ . Finally, a unit cell with the corners  $\{(x, y), (x + 1, y), (x + 1, y + 1), (x, y + 1)\}$  is specified by its center point  $c = (x + 0.5, y + 0.5)$ .

Let us use the following notation (their meaning is explained right after the listing):

- *side borders*: elements in  $C1 = \{w, e, n, s\}$ , where **w** stands for “west border”, **e** for “east border”, **n** for “north border”, and **s** for “south border”;
- *land corners*: elements in  $C2 = \{nw, ne, sw, se\}$ , where **nw** stands for “north-west land corner”, **ne** for “north-east land corner”, **sw** for “south-west land corner”, and **se** for “south-east land corner”;
- *golf corners*: elements in  $C3 = \{nw', ne', sw', se'\}$ , where **nw'** stands for “north-west golf corner”, **ne'** for “north-east golf corner”, **sw'** for “south-west golf corner”, and **se'** for “south-east golf corner”;
- *extreme side borders*: elements in  $C4 = \{xw, xe, xn, xs\}$ ;
- *extreme land corners*: elements in  $C5 = \{xnw, xne, xsw, xse\}$

A line specified by the point  $l = (x, y + 0.5)$  is on the *east border* of a word  $f$  if the cell represented by the point  $c = (x - 0.5, y + 0.5)$  is in the internal area of  $f$ , while the cell represented by  $c = (x + 0.5, y + 0.5)$  is in the external area of  $f$ . For the other west, north, and south directions, the definition is similar. A point  $p = (x, y) \in \mathbb{Z}^2$  is on the *south-east land corner* of a word  $f$  if the cell represented by  $c = (x - 0.5, y + 0.5)$  is in the area of  $f$ , while the other 3 cells around are not in the area of  $f$  (they are in the external area of  $f$ ). For the other 3 types of land corners the definition is similar. A point  $p = (x, y) \in \mathbb{Z}^2$  is on the *south-east golf corner* of a word  $f$  if the cell specified by  $c = (x - 0.5, y + 0.5)$  is not in the area of  $f$  (it is in the external area of  $f$ ), while the other 3 cells around are in the area of  $f$ . For the other 3 types of golf corners the definition is similar.

A cell is an *extreme cell* if it touches at most one other cell in the interior area. A side (i.e., north, west, east, south) or a corner on the border of a word is *extreme* if it belongs to an extreme cell and does not touch another cell (side elements with an end point touching another cell are also excluded - they are not extreme).

*Glueing combinations.* The constraints on glueing the borders are independently put on one or more of the following combinations  $(x, y)$ :

$x$  and  $y$  are different and either they are both in  $\{e, w\}$ , or both in  $\{s, n\}$ , or both are land corners in  $\{nw, ne, sw, se\}$ , or both are combinations golf-land corners for the same directions.

Spelling out the resulting combinations we get the following lists:

- linking side borders:  $L1 = \{(w, e), (e, w), (n, s), (s, n)\}$ ;

- linking land corners:  $L2 = \{(nw, ne), (nw, se), (nw, sw), (ne, nw), (ne, se), (ne, sw), (se, nw), (se, ne), (se, sw), (sw, nw), (sw, ne), (sw, se)\}$ ;
- linking golf-land corners:  $L3 = \{(nw', nw), (nw, nw'), (ne', ne), (ne, ne'), (se', se), (se, se'), (sw', sw), (sw, sw')\}$ .

Let  $EL1$ ,  $EL2$ ,  $EL3$  denote linking-combinations where all side borders and land corners are restricted to be extreme. (e.g.  $EL1 = \{(xw, xe), (xe, xw), (xn, xs), (xs, xn)\}$ )

The set of all combinations in  $L1 \cup L2 \cup L3 \cup EL1 \cup EL2 \cup EL3$  is denoted by **Connect**.

*Constricting formulas.* On each of the above eligible glueing combinations  $(x, y)$  we put a constraint consisting of a propositional logic formula <sup>1</sup>  $F \in PL(\phi_1, \phi_2, \phi_3, \phi_4)$ , i.e., a boolean formula built up starting with the following atomic formulas:

$$\phi_1(x, y) = "x < y", \phi_2(x, y) = "x = y", \phi_3(x, y) = "x > y", \text{ and } \phi_4(x, y) = "x \# y".$$

The meaning of the connectors is the following: “<” - left is included into the right; “=” - left is equal to the right; “>” - left includes the right; “#” - left and right overlaps, but no one is included in the other.

For instance:  $f(e = w)g$  means “restrict the general composition of  $f$  and  $g$  such that the east border of  $f$  is identified to the west border of  $g$ ”;  $f(e > w)g$  - the east border of  $f$  includes all the west border of  $g$ , but some east borders of  $f$  may still be not covered by west borders of  $g$ ; etc.

We also use the notation

$$\phi_0(x, y) = "x ! y", \text{ where “!” means empty intersection.}$$

Actually, this is a derived formula  $\neg(\phi_1(x, y) \vee \phi_2(x, y) \vee \phi_3(x, y) \vee \phi_4(x, y))$ .

*Particular composition operators.* We are now in a position to introduce the particular composition operators induced by the above constricting formulas.

**Definition 2.** (restricted compositions)

A *restriction formula*  $\phi$  is a boolean combination in  $PL(F_1, \dots, F_n)$ , where  $F_i$  are constricting formulas involving certain eligible glueing combinations  $(x_i, y_i) \in \text{Connect}$ . A *restricted composition operation*  $_{\phi}$  is the restriction of the general composition to composite words satisfying  $F$ . A word  $h \in f . g$  belongs to  $f (F) g$  if for all glueing combinations  $(x_i, y_i)$  occurring in  $F$  the contact of the  $x_i$  border of  $f$  and  $y_i$  border of  $g$  satisfies  $F_i$ .  $\square$

<sup>1</sup>  $PL(Atom)$  denotes the set of propositional logic formulas built up with atomic formulas in  $Atom$ . For typing reasons, the boolean operations “not”, “and”, and “or” are denoted by “ $\neg$ ”, “ $\&$ ”, and “ $\vee$ ”, respectively.

The non-restricted general composition  $f.g$  is the same as  $f(.)g$ .

Notice that the restricted composition operations are not always associative; e.g.,

$$((a (s=n) a) (e>w) b) (e>w) c \neq (a (s=n) a) (e>w) (b (e>w) c).$$

When some parentheses are missing, we suppose a left-parentheses order applies, as in  $((C1 \text{ op } C2) \text{ op } C3)$ .

*On expressiveness.* One can easily show that the side borders (east/west) or land corners alone are less expressive than acting together. The example below shows golf-corner points add to the expressive power of the composition as well: the left composite can not be expressed without golf-corner constraints.

blue (nw'<nw) yellow:

|   |   |   |   |
|---|---|---|---|
| b | b | b | b |
| b | y | y | y |
| b | b | b | y |
|   | y | y | y |

blue (s#n & e#w) yellow

|   |   |   |   |
|---|---|---|---|
| b | b | b | b |
| b | y | y | y |
| b | b | b | y |
| y | y | y |   |

|   |   |   |   |
|---|---|---|---|
|   | y | y | y |
| b | b | b | b |
| b |   | y | y |
| b | b | b |   |

*Not specified glueing combinations.* The interpretation we just introduced shows that the constricting formulas act on the involved glueing combinations, while for the glueing combinations  $(x_j, y_j)$  not occurring in the formula no constraints are imposed, at all. One can also introduce a *stricter interpretation*  $-(F)_s-$  where the not-occurring combinations are considered to have no contact at all (empty intersection). In this latter interpretation, the contacts are only those directly specified in the formula. For instance,  $a (e=w)_s b$  is empty; to have the same result as in  $a (e=w) b$ , one has to specify all contacts elements as in  $a ((e=w) \& (ne=nw) \& (se=sw))_s b$ .

*Iterated composition operators.* The *iterated composition operators* are denoted by

$$(\_) * (F), \text{ for a restriction formula } F.$$

**Definition 3.** The set of expressions obtained using all the operators defined so far are denoted by x2RE; they represent *two-dimensional regular expressions extended with composition operators on extreme cells*. Dropping the operators involving the extreme cells we get n2RE, the basic *new type of regular expressions for two-dimensional words*.  $\square$

*Examples* The examples below are related to  $S1$ , the original FIS we have considered in the beginning of the section. We first show the expressions, then include samples of words associated to these expressions.

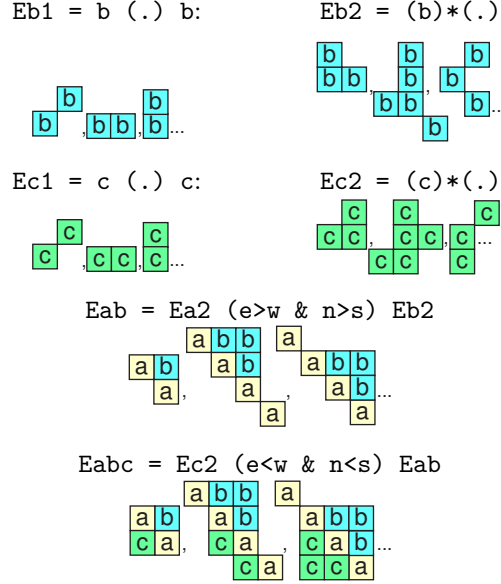
$$Ea1 = a (se=nw) a:$$

|   |
|---|
| a |
| a |

$$Ea2 = (a) * (se=nw)$$

|   |   |   |
|---|---|---|
|   |   | a |
| a | a | a |
| a | a | a |





Combined with the constraint to have rectangular words, the regular expression **Eabc** specifies the language of **S1**. For instance a word in **Eabc** is accepted if the result of the expression

$$Eabc \ (n=s) \_s \ Lx \ (s=n) \_s \ Lx \ (e=w) \_s \ Cx \ (w=e) \_s \ Cx$$

is not empty, where  $Lx=(x)*(e=w)$  and  $Cx=(x)*(n=s)$ .

*Chains, and pretzel-like shapes.* A *pretzel word* is a set of cells such that each cell has precisely 2 touching cells around and no proper subset has this property.

The construction of pretzels is in two stages: (1) first, we give a procedure for generating chains; (2) then, a pretzel is produced by an appropriate connection of two chains.

For constructing a chain, we use the strict interpretation of the composition, i.e., the one where the only connecting elements are those specified in the constricting formula. This, combined with the restriction to make composition for extreme elements only, leads to a formula for iterative generation of chains: start with a cell and iteratively add one cell at a time, connected via an extreme element.

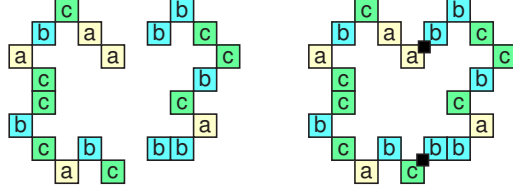
Finally, a pretzel is obtained composing two chains via their extreme elements and asking for equality, to avoid having only one end of a chain connected.

*Example* Expression recognizing chain words over the alphabet  $V = \{a, b, c\}$ :

$$C = (a + b + c) * [(xse > xnw) \vee (xsw > xne) \vee (xne > xsw) \vee (xnw > xse) \vee (xe > xw) \vee (xn > xs) \vee (xw > xe) \vee (xs > xn)] \_s.$$

An expression recognizing pretzel words over the alphabet  $V = \{a, b, c\}$ :

$$P = C[(xse = xnw) \vee (xsw = xne) \vee (xne = xsw) \vee (xnw = xse)]_s C.$$

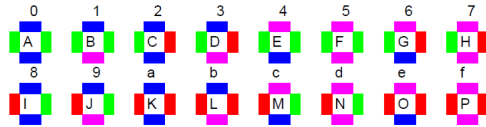


#### 4 Tiling cells, chains, and membranes in complete $2 \times 2$ finite interactive systems

This section describes a two-dimensional version of the procedure used in Theorem 1. A key ingredient is the expression  $(a + bd^*c)^*$ , for the complete automation with transitions  $1 \xrightarrow{a} 1, 1 \xrightarrow{b} 2, 2 \xrightarrow{c} 1, 2 \xrightarrow{d} 2$ , describing the passing from state 1 to itself. Actually, it shows there are two indecomposable paths from 1 to 1 which are freely combined. These paths are the following: (1)  $a$ , giving a direct transition from 1 to 1; and (2)  $bd^*c$ , corresponding to going from 1 to 2, looping there, and finally, going back to 1.

The question we are discussing in this section is on the extension of the above mechanism to two dimensions. Notice that finite interactive systems are much more complex than finite automata<sup>2</sup>, so one can not expect to get simple results.

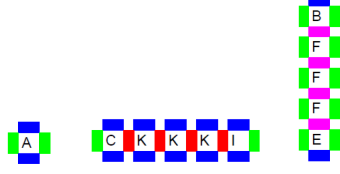
A complete  $2 \times 2$  FIS is specified by the following transitions:



The states/classes of this FIS will be denoted using the initials of the colors: the classes are g (green) and r (red), while the states are b (blue) and m (magenta). A scenario is called indecomposable if all its west, north, east, south borders are labeled with c1, s1, c2, s2 respectively and it doesn't contain any sub-scenarios with this property.

Suppose we want to construct the recognised 2-dimensional words corresponding to scenarios going from g/b, used for west/north borders, to g/b, used for east/south borders. The indecomposable scenarios include the following combinations:

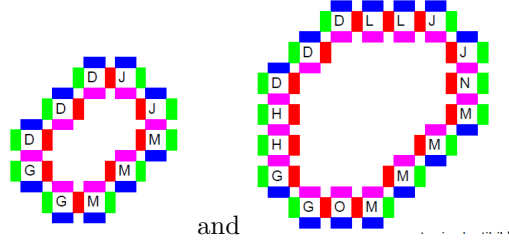
<sup>2</sup> For instance, by the projection of the languages recognised by finite interactive systems (FISs) to the 1st row (to get string languages), one gets context-sensitive languages[17]. As a side-effect, this shows emptiness problem for FISs is not decidable.



The first scenario A specifies the direct passing from g/b to g/b. The 2nd scenario, corresponding to the expression  $(C (e=w) (K *(e=w))) (e=w) I$ , is a generic indecomposable scenario describing the case when one keeps fixed the state b and goes from g to the other class r, loops there, and finally comes back to g. The case of the 3rd scenario is similar. Let us denote by Ea, Eb, and Ec regular expressions for the words corresponding to these 3 types of scenarios.

Much more complicated is the situation with the indecomposable scenarios corresponding to the case when both g/b are changed going to the combination r/m, loop (tile!) there, and finally come back to g/b.

The first step (going from g/b to r/m) leads to a kind of *pretzel-shape*. A first approximation of this form is described in the following pictures.



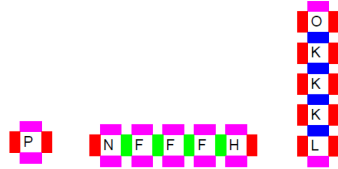
Formally, a pretzel-shape is a word where each cell has precisely 2 neighbouring cells (on north, south, east, west, or diagonal directions) and no proper sub-word has this property.

Let us denote by Ed an expression for these pretzel-like scenarios associated to our complete  $2 \times 2$  FIS. Given an expression for a pretzel-like scenario  $f$  we denote by  $f[X]$  the result of filling the pretzel hole with  $X$ . An expression for defining  $f[X]$  may be

$$f \ [ (s>n) \& (w>e) \& (n>s) \& (e>w) ] \ X.$$

It is worthwhile to emphasize that this composition may be not empty only in the case  $X$  has no holes.

The second step (tiling in r/m) is relatively easy: it consists in tiling the interior contour of the pretzel with the following scenarios



and pretzel-like scenarios from r/m to g/b. The extra constraint put here is to have no holes after this procedure, except for the interior holes of the included new pretzel-like words.

The last step (going from r/m to g/b) is similar to the first step.

Suppose we have regular expressions for describing these indecomposable scenarios. Then, the two-dimensional word language corresponding to the passing from g/b to g/b satisfies the relation

$$X = (Ea + Eb + Ec + Ed[X]) * (.) \quad (1)$$

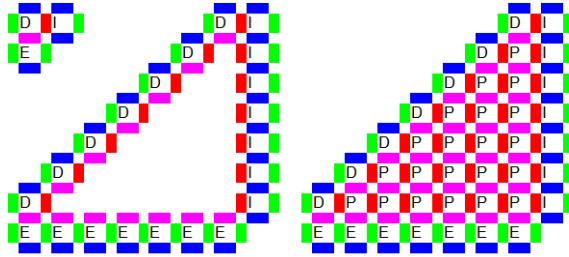
We conclude this section formally stating the result obtained so far.

**Theorem 3.** ([3]) *The x2RE formalism and a mechanism for solving the recursive equation 1 are expressive enough to represent the words recognised by the complete 2×2 FIS.*

Notice that the restriction to 2×2 FISs is not important. With an iterative application of the method, the result carries over finite interactive systems with an arbitrary number of states/classes. Also, we may use the same tiling mechanism for generating the language of an incomplete 2×2 FIS.

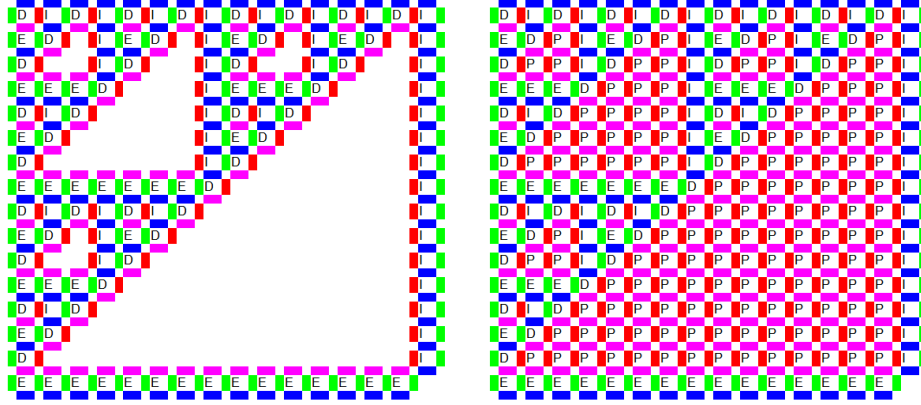
*Example.* We calculate the set of words passing from g/b to g/b recognized by the FIS “Pascal triangle” with 4 transitions: D, E, P, I. There are no direct transitions or line/column words from g/b to g/b. Hence the only indecomposable scenarios from g/b to g/b are pretzel words from g/b to r/m filled with r/m to r/m scenarios:

$$Ed = D*(ne=sw)(xe \# xw \& xs \# xn)[(I*(s=n))(sw=ne)(E*(e=w))]$$



The expression X generates all scenarios passing from g/b to g/b. If additionally we accept only rectangular words, we obtain the language recognized by the FIS “Pascal triangle”.

$$X = Ed[P*(.)]*(.)$$



## 5 Relating FIS scenarios and P systems

In this section we briefly discuss the relationship between the FIS-based membrane systems and P systems. We suppose the reader is familiar with the original P systems model and the multitude of variations occurring in the literature till now.

A first observation is that FIS languages are not Turing complete (we already mentioned this). To get the full computation power one has to add data for states/classes as in the register-voice interactive model rv-IS [29].

There are two aspects to be discussed: the *static structure* of the membranes and the *dynamics* of the systems.

*Static structure.* With respect to the static structure, notice that the FIS-based membrane systems are more rigid (or geometric). An additional difference is that the contents of the membrane are also rigid/geometric. In particular, there are cells, chains, walls, or other membranes tiling together and organizing the interior areas of the membranes. This particular emphasis on the geometric shapes of the membranes has positive and negative consequences. On the positive side, we mention a significantly decrease of nondeterminism in the system leading to more efficient simulations. On the opposite side, the cells are forced to interact with their neighbouring cells only, making more difficult to model general reaction rules involving far apart objects of a membrane region.

*Dynamics.* One way to introduce the dynamics corresponding to P systems runs is to add a new dimension into the model. One can consider 2FIS, a model where the 1-dimensional space part of the finite interactive systems is replaced by a 2-dimensional space. Now, each action (cell) has 3 orthogonal directions, say: up-down and left-right for space and back-front for time. Ignoring the time dimension, one gets usual finite interactive systems, for which the static membrane structure may be studied as in the present paper. Adding the time dimension, one can model the passing from a static membrane structure to another. Again, due

to the rigid/geometric structure of the model some costly high-level operations in P systems like arbitrary multiplication of the objects in a step, membrane division, dissolution, or migration can not be done in a step, but probably can be modelled if more 2FIS steps are allowed.

## 6 Related and future works

Regular expressions are introduced in the seminal paper by Kleene [13] on the representation of events in neural nets and automata; it was published in the early 1950s. Kleene theorem (i.e., the equivalence between finite automata and appropriate regular expressions) was extended to cover other computing models of interest and is a basis for the development of algebraic theories for those models.

The study of two dimensional languages [11, 18] has started in 1960's. In 1990's, a robust class of "regular two dimensional languages" has been identified; it may be specified either by tile systems, or by a type of cellular automata, or by a class of monadic second-order formulas, etc. Unfortunately, the class is quite complex - for instance, emptiness property is not decidable, see [17].

Interactive computation [12] is becoming more and more important in the recent years, in particular due to the advance of multicore computation. We use a model rv-IS [29] based on space-time duality. In particular, finite interactive systems [27] are the space-time invariant extension of finite automata in this context. A Kleene theorem for finite interactive systems follows directly from their equivalence with tile systems [25].

With respect to P systems, the FIS-based membrane systems model may be a good proposal for an intermediate level between P-system and physical cells. If this is the case, then a full translation of high-level P systems concepts to the low-level FIS-based membrane systems deserves to be done. Finite interactive systems are strongly related to massively parallel structural programs, hence one can get efficient implementations this way.

## References

1. Banu-Demergian, I., Paduraru, C., Stefanescu, G.: A new representation of two-dimensional patterns and applications to interactive programming. In: FSEN 2013. LNCS (2013, to appear)
2. Banu-Demergian, I., Stefanescu, G.: On the contour representation of two-dimensional patterns (2013), draft
3. Banu-Demergian, I., Stefanescu, G.: Representation of scenarios in finite interactive systems (2013), draft, submitted
4. Bloom, S., Esik, Z.: Equational axioms for regular sets. *Mathematical Structures in Computer Science* 3, 1–24 (1993)
5. Bonsangue, M., Rutten, J., Silva, A.: A Kleene theorem for polynomial coalgebras. In: *Proc. FSSCS'09*. LNCS, vol. 5504, pp. 122–136. Springer-Verlag (2009)
6. Ceterchi, R., Gramatovici, R., Jonoska, N.: Tiling rectangular pictures with P systems. In: *Membrane Computing*, pp. 88–103. Springer (2004)

7. Ceterchi, R., Gramatovici, R., Jonoska, N., Subramanian, K.: Tissue-like P systems with active membranes for picture generation. *Fundamenta Informaticae* 56(4), 311–328 (2003)
8. Chen, H., I. Rudolf, Ionescu, M., Paun, G., Pérez-Jiménez, M.: On string languages generated by spiking neural P systems. *Fundamenta Informaticae* 75(1), 141–162 (2007)
9. Conway, J.: *Regular Algebra and Finite Machines*. Chapman and Hall (1971)
10. Garg, V., Ragunath, M.: Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Science* 96, 285–304 (1992)
11. Giammarresi, D., Restivo, A.: Two-dimensional languages. In: *Handbook of formal languages*, pp. 215–267. Springer (1997)
12. Goldin, D., Smolka, S., Wegner, P.: *Interactive computation: The new paradigm*. Springer (2006)
13. Kleene, S.: Representation of events in nerve nets and finite automata. *Automata Studies* (34), 3–41 (1956)
14. Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events. In: *LICS'91*. pp. 214–225. IEEE (1991)
15. Krob, D.: Complete systems of  $\beta$ -rational identities. *Theoretical Computer Science* 89, 207–343 (1991)
16. Kuich, W., Salomaa, A.: *Semirings, automata and languages*. Springer-Verlag, Berlin (1985)
17. Latteux, M., Simplot, D.: Context-sensitive string languages and recognizable picture languages. *Information and Computation* 138(2), 160–169 (1997)
18. Lindgren, K., Moore, C., Nordahl, M.: Complexity of two-dimensional patterns. *Journal of statistical physics* 91(5-6), 909–951 (1998)
19. Paun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
20. Paun, G.: Introduction to membrane computing. In: *Applications of Membrane Computing*, pp. 1–42. Springer (2006)
21. Paun, G., Rozenberg, G., Salomaa, A.: *The Oxford handbook of membrane computing*. Oxford University Press, Inc. (2010)
22. Petri, C.: *Kommunikation mit automaten*. Ph.D. thesis, Instituts für Instrumentelle Mathematik, Bonn, Germany (1962)
23. Qi, Z., You, J., Mao, H.: P systems and Petri nets. In: *Membrane Computing*, LNCS, vol. 2933, pp. 286–303. Springer (2004)
24. Salomaa, A.: Two complete axiom systems for the algebra of regular events. *Journal of the ACM (JACM)* 13(1), 158–169 (1966)
25. Sofronia, A., Popa, A., Stefanescu, G.: Undecidability results for finite interactive systems. *Romanian Journal of Information Science and Technology* 12(2), 265–279 (2009), also: Arxiv, CoRR abs/1001.0143, 2010
26. Stefanescu, G.: *Network algebra*. Springer Verlag (2000)
27. Stefanescu, G.: Algebra of networks: Modeling simple networks as well as complex interactive systems. In: *Proof and System-Reliability*, pp. 49–78. Springer (2002)
28. Stefanescu, G.: Interactive systems: From folklore to mathematics. In: *ReImics'01*. LNCS, vol. 2561, pp. 197–211. Springer (2002)
29. Stefanescu, G.: Interactive systems with registers and voices. *Fundamenta Informaticae* 73(1), 285–305 (2006)





# Modelling of Surface Runoff using 2D P colonies

Luděk Cienciala, Lucie Ciencialová, and Miroslav Langer

Institute of Computer Science  
and

Research Institute of the IT4Innovations Centre of Excellence,  
Silesian University in Opava, Czech Republic  
{ludek.cienciala, lucie.ciencialova, miroslav.langer}@fpf.slu.cz

**Abstract.** We continue the investigation of 2D P colonies introduced as a class of abstract computing devices composed of independent agents, acting and evolving in a shared 2D environment where the agents are located. Agents have limited information about the contents of the environment where they can move in four directions. In this paper we continue in the research of modelling of surface runoff by 2D P colonies. We add to the simulation environment information about flow direction and amount of water in pits (places without runoff, lakes,...). We compare the data from the simulation with the data generated by simulation model of water erosion SIMWE.

## 1 Introduction

P colonies were introduced in the paper [7] as formal models of computing devices belonging to membrane systems and similar to formal grammars called colonies. This model is inspired by the structure and the behaviour of communities of living organisms in a shared environment. The independent organisms living in a P colony are called agents. Each agent is represented by several objects embedded in a membrane. The number of objects inside each agent is the same and constant during computation. The environment is agents' communication channel and storage place for objects. At any moment all agents "know" about all the objects in the environment and they can access any object immediately. The reader can find more information about P colonies in [6, 2]. P colonies are one of the types of P systems. They were introduced in 2000 in [9] by Gheorghe Păun as a formal model inspired by the structure and the behaviour of cells.

With each agent a set of programs is associated. The program, which determines the activity of an agent, is very simple and depends on the contents of agents and on types and number of objects placed in the environment. An agent can change the contents of the environment through programs and it can affect the behaviour of other agents through the environment. This influence between agents is the key factor in the functioning of the P colony. At any moment each object inside every agent is affected by the execution of the program.

For more information about P systems see [11, 10] or [13].

In addition 2D P colony has the environment in a form of a 2D grid of square cells. The agents are located in this grid and their view is limited to the cells that

immediately surround them [1]. Based on the contents of these cells, the agents decide their future locations.

Behaviour of each agent is based on its set of programs. The programs are formed from two rules of type rewriting, communication and movement. By using the rewriting rule one object within the agent is changed (evolved) to another object. When the communication rule is applied one object from the environment is consumed by the agent and one object from content of the agent is placed to the environment. The last type of rules is the movement rule. The condition for the movement of an agent is to find specific objects in specific locations in the environment. This is specified by a matrix with elements - objects. The agent is looking for at most one object in every surrounding cell. If the condition is fulfilled then the agent moves one cell up, down, left or right.

The program can contain one movement rule at most. To achieve the greatest simplicity in agent behaviour, we set another condition. If the agent moves, it cannot communicate with the environment. So if the program contains a movement rule, then the second rule is the rewriting rule.

Although the colony is a theoretical computing model through 2D, it is a suitable tool for modelling the behaviour of natural multi-agent systems - colonies of bacteria or ants, spreading substances in homogeneous and inhomogeneous medium.

In this paper we present hydrological modelling flow of liquid over the Earth's surface using 2D P colonies. Based on the entered data - the slope surface, a source of fluid and quantity - we simulate the fluid distribution in the environment.

To obtain the similarity of our model with the real situation of water overflow we compare the results obtained by the simulation using 2D P colonies with results that provide a hydrological simulation model SIMWE. SIMWE is implementation of process based water erosion simulation developed by Mitas et al. in 1996 in [8].

The first part of the paper is devoted to 2D P colonies. The rest is organised as follows: The issue of the flow of liquid over the surface, problem solution - maps preparation, definition of the agent, process simulation, comparison with results of the model SIMWE and future expansion.

## 2 Definitions

Throughout the paper we assume that the reader is familiar with the basics of the formal language theory.

We use  $NRE$  to denote the family of the recursively enumerable sets of natural numbers,  $N$  is the set of natural numbers. Let  $\Sigma$  be the alphabet. Let  $\Sigma^*$  be the set of all words over  $\Sigma$  (including the empty word  $\varepsilon$ ). We denote the length of the word  $w \in \Sigma^*$  by  $|w|$  and the number of occurrences of the symbol  $a \in \Sigma$  in  $w$  by  $|w|_a$ .

A multiset of objects  $M$  is a pair  $M = (V, f)$ , where  $V$  is an arbitrary (not necessarily finite) set of objects and  $f$  is a mapping  $f : V \rightarrow N$ ;  $f$  assigns

to each object in  $V$  its multiplicity in  $M$ . The set of all multisets with the set of objects  $V$  is denoted by  $V^\circ$ . The set  $V'$  is called the support of  $M$  and is denoted by  $\text{supp}(M)$  if for all  $x \in V'$   $f(x) \neq 0$  holds. The cardinality of  $M$ , denoted by  $|M|$ , is defined by  $|M| = \sum_{a \in V} f(a)$ . Each multiset of objects  $M$  with the set of objects  $V' = \{a_1, \dots, a_n\}$  can be represented as a string  $w$  over alphabet  $V'$ , where  $|w|_{a_i} = f(a_i)$ ;  $1 \leq i \leq n$ . Obviously, all words obtained from  $w$  by permuting the letters represent the same multiset  $M$ . The  $\varepsilon$  represents the empty multiset.

### 3 2D P colonies

We briefly summarize the notion of 2D P colonies. A P colony consists of agents and an environment. Both the agents and the environment contain objects. With each agent a set of programs is associated. There are three types of rules in the programs.

The first rule type, called the evolution rule, is of the form  $a \rightarrow b$ . It means that the object  $a$  inside the agent is rewritten (evolved) to the object  $b$ . The second rule type, called the communication rule, is of the form  $c \leftrightarrow d$ . When the communication rule is performed, the object  $c$  inside the agent and the object  $d$  outside the agent swap their places. Thus, after the execution of the rule, the object  $d$  appears inside the agent and the object  $c$  is placed outside the agent. The third rule type, called the motion rule, is of the form matrix  $3 \times 3 \rightarrow$  “move direction”. Based on the contents of the neighbouring cells, an agent can move one step to the left, right, up or down.

A program can contain maximum one motion rule. When there is a motion rule inside a program, there cannot be a communication rule inside the same program.

**Definition 1.** *The 2D P colony is a construct*

$$\Pi = (A, e, Env, B_1, \dots, B_k, f), k \geq 1, \text{ where}$$

- $A$  is an alphabet of the colony, its elements are called objects,
- $e \in A$  is the basic environmental object of the colony,
- $Env$  is a pair  $(m \times n, w_E)$ , where  $m \times n, m, n \in \mathbb{N}$  is the size of the environment and  $w_E$  is the initial contents of environment, it is a matrix of size  $m \times n$  of multisets of objects over  $A - \{e\}$ .
- $B_i, 1 \leq i \leq k$ , are agents, each agent is a construct  $B_i = (O_i, P_i, [o, p])$ ,  $0 \leq o \leq m, 0 \leq p \leq n$ , where
  - $O_i$  is a multiset over  $A$ , it determines the initial state (contents) of the agent,  $|O_i| = 2$ ,
  - $P_i = \{p_{i,1}, \dots, p_{i,l_i}\}, l \geq 1, 1 \leq i \leq k$  is a finite set of programs, where each program contains exactly 2 rules, which are in one of the following forms each:
    - \*  $a \rightarrow b$ , called the evolution rule,  $a, b \in A$ ;
    - \*  $c \leftrightarrow d$ , called the communication rule,  $c, d \in A$

- \*  $[a_{q,r}] \rightarrow s, 0 \leq q, r \leq 2, a_{q,r} \in A, s \in \{\Leftarrow, \Rightarrow, \Uparrow, \Downarrow\}$ , called the motion rule;
- \* The third part of program is natural number  $h \in N$ , which determine priority level of the program.
- $[o, p]$  are the coordinates of the initial placement agent in the environment.
- $f \in A$  is the final object of the colony.

The configuration of the 2D P colony is given by the state of the environment - matrix of type  $m \times n$  with multisets of objects over  $A - \{e\}$  as its elements, and by the state of all agents - pairs of objects from alphabet  $A$  and the coordinates of the agents. An initial configuration is given by the definition of the 2D P colony.

The computational step consists of three parts. The first part lies in determining the applicable set of programs according to the actual configuration of the P colony. In the second part we have to choose one program corresponding to each agent from the set of applicable programs with maximum priority level. The third part is the execution of the chosen programs.

A change of the configuration is triggered by the execution of programs and it involves changing the state of the environment, contents and placement of the agents.

The computation is nondeterministic and maximally parallel. The computation ends by halting when no agent has an applicable program.

The result of the computation is the number of copies of the final object placed in the environment at the end of the computation.

The reason for the introduction of 2D P colonies is not the study of their computational power but monitoring their behaviour during the computation. We can define measures to describe the dynamics of the computation:

- the number of moves of agents
- the number of agents inside the certain cell or the set of the cells
- the number of visited cells (or not visited cells)
- the number of copies of a certain object in the home cell or throughout the environment.

## 4 The issue of the flow of liquid over the surface

The issue of the flow of liquid over the Earth's surface is studied by experts from two areas - hydrology and geoinformatics. Both of these disciplines work closely together on the issue of the so-called "surface runoff". Surface runoff is the water flow that occurs when the soil is infiltrated to full capacity and excess water from rain, meltwater, or other sources flows over the land.

Surface runoff can be generated in four reasons: infiltration excess overland flow, saturation excess overland flow, antecedent soil moisture, subsurface return flow. Infiltration excess overland flow occurs when the rate of rainfall on a surface exceeds the rate at which water can infiltrate the ground, and any depression storage has already been filled. When the soil is saturated and the depression

storage filled, and rain continues to fall, the rainfall will immediately produce surface runoff - saturation excess overland flow. Soil retains a degree of moisture after a rainfall. This residual water moisture (antecedent soil moisture) affects the soil's infiltration capacity. During the next rainfall event, the infiltration capacity will cause the soil to be saturated at a different rate. The higher the level of antecedent soil moisture, the more quickly the soil becomes saturated. Once the soil is saturated, runoff occurs. After water infiltrates the soil on an up-slope portion of a hill, the water may flow laterally through the soil, and exfiltrate (flow out of the soil) closer to a channel. This is called subsurface return flow or throughflow.

We can say that generation surface runoff depends on type of soil, temperature, humidity and rainfall. The task of our model is to determine which way the flow would run and which areas could be affected by flash floods.

#### 4.1 SIMWE - Simulation of Water Erosion

SIMWE is a bivariate model of erosion, sediment transport and deposition by overland flow, designed for complex terrain, soil and cover conditions. It uses a Green's function Monte Carlo method to solve the underlying continuity equations. More can reader find in [8]. The model is implemented as two modules in software GRASS GIS. It is a Geographic Information System (GIS) used for data management, image processing, graphics production, spatial modelling, and visualization of many types of data (see [4]).

The first module is called `r.sim.water` and it is a landscape scale simulation model of overland flow designed for spatially variable terrain, soil, cover and rainfall excess conditions. A 2D shallow water flow is described by the bivariate form of Saint Venant equations (e.g. [5]). The numerical solution is based on the concept of duality between the field and particle representation of the modeled quantity. The key inputs of the model include elevation, flow gradient vector, rainfall excess rate and a surface roughness coefficient. Output includes a water depth raster map and a water discharge raster map.

The second module `r.sim.sediment` is simulation model of soil erosion, sediment transport and deposition caused by flowing water designed for spatially variable terrain, soil, cover and rainfall excess conditions. The function of this module is out of scope of this paper.

## 5 Application of 2D P colonies in solving the problem of surface runoff

2D P colonies seem to be suitable tool for modeling surface runoff. The environment can contain objects representing slope of terrain, type of cover and soil. Agents represent the units of water and their programs determine behaviour of water running over the surface. We can assume that the soil is already saturated thus the main factor of overland flow is the slope of the field. The type of terrain and soil is not implemented yet.

We divide solution of the problem into two parts - (1) preparation of maps (2D P colony's environment) and (2) definition of agents. We assume that the soil is already saturated thus the main factor of overland flow is the slope of the field.

### 5.1 Preparation of maps

Map data is obtained from the geographic information system (GIS) and processing system GRASS. We use the map data for the Czech Republic obtained from dataset FreeGoedataCZ.

Raster graphics images are probably the most appropriate format for modelling real-world phenomena in the field of GIS. To process this format, many tools were created and can be used for performing various analyses. A raster image is composed of a regular network of cells, usually in a square shape, to which values of displayed properties can be assigned independently. More information about GIS and image processing the reader can find in [3] and about geosimulation in [12].

The first step to simulate the flow of liquid over relief was the determination of its runoff from individual pixels (cells). Gradient with respect to an adjacent cell is defined as the ratio of the height difference to the horizontal distance. Gradient is positive due to the lower neighbours, or negative due to higher and zero in relation to the neighbours of the same height. Lowest neighbour is neighbour with the largest positive gradient.

There are two basic algorithms to calculate the runoff:


- Single flow direction (SFD) - each pixel of the liquid flows in one direction only (toward neighbour in the direction of the largest gradient). Each pixel belongs to only one basin.
- Multiple flow direction (MFD) - fluid can flow out of each pixel in multiple directions, maximum of eight. In the case of MFD a unit volume flow is fairly distributed among all lower neighbours. The MFD may include the pixel to multiple basins.

There is implemented a tool for calculating the flow direction in GRASS software, called simply TerraFlow. TerraFlow tool works as a multiple flow (MFD) or simple flow direction (SFD). After its execution integer raster file is created that specifies the flow direction for each cell.

Eight basic directions of the flow are represented by the numbers 45, 90, 135, 180, 225, 270, 315 and 360 (see Table 1). If there is more than one direction (MFD), the number contained in the cell is generated as sum of values of the directions.

In 2D P colony model we can use the result of both algorithms. If the MFD is used to generate the directions the agent can only move in one of the directions specified in the cell.

What we obtain from GRASS is a raster file with natural number in each cell corresponding to the runoff from this cell. Because 2D P colony works with discrete symbols and not with numbers, it needed to transcode numbers to symbols. A coding table is shown on Table 2.

|     |   |     |
|-----|---|-----|
| 135 | 90  | 45  |
| 180 |  | 360 |
| 225 | 270   | 315 |

**Table 1.** The eight basic directions

|                  |               |              |            |              |            |            |            |            |
|------------------|---------------|--------------|------------|--------------|------------|------------|------------|------------|
| <b>direction</b> | $\rightarrow$ | $\leftarrow$ | $\uparrow$ | $\downarrow$ | $\searrow$ | $\swarrow$ | $\nearrow$ | $\nwarrow$ |
| <b>symbol</b>    | $a$           | $E$          | $i$        | $m$          | $q$        | $u$        | $y$        | $2$        |

**Table 2.** The coding table

## 5.2 Definition of the agent

Agents in 2D P colonies have capacity 2. It follows that the agent contains two objects, and each program is composed by two rules.

Each of the objects inside the agent carries the information about the state of the agent. The first object has information about the activity of the agent. At this stage of the simulation it is the information that the agent “flows” down the terrain ( object  $X$ ) or it is still inactive (belonging to the rainfall that have not fall - objects  $A, B, C, D, F, G, R, S, T, U, V, W$ , it stops in sinks - configuration of agent is  $VS$ ) . The second object stores information about the previous direction of flow. This information can further modify the way of the agent as inertia.

Objects and their association to the flow directions are given in the following table.

|                  |               |              |            |              |            |            |            |            |
|------------------|---------------|--------------|------------|--------------|------------|------------|------------|------------|
| <b>direction</b> | $\rightarrow$ | $\leftarrow$ | $\uparrow$ | $\downarrow$ | $\searrow$ | $\swarrow$ | $\nearrow$ | $\nwarrow$ |
| <b>symbol</b>    | 9             | 8            | 6          | 7            | $D$        | $D$        | $U$        | $U$        |
| <b>symbol</b>    | $L$           | $K$          | $H$        | $I$          | $I$        | $I$        | $H$        | $H$        |

The first subset of programs with priority 0 is defined for the first step of computation. The initial configuration of each “working” agent is  $Xe$ .

$$\begin{aligned}
 (1) & \left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; e \rightarrow 9; 0 \right\rangle; (2) \left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; e \rightarrow 8; 0 \right\rangle; \\
 (3) & \left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; e \rightarrow 6; 0 \right\rangle; (4) \left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; e \rightarrow 7; 0 \right\rangle; \\
 (5) & \left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; e \rightarrow D; 0 \right\rangle; (6) \left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; e \rightarrow D; 0 \right\rangle; \\
 (7) & \left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; e \rightarrow U; 0 \right\rangle; (8) \left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; e \rightarrow U; 0 \right\rangle;
 \end{aligned}$$

In the case of cross direction (after applying the programs (5) and (6) resp. (7) and (8)) the agent moves one step left or right and it is necessary to take one step down (resp. up).

$$(9) \left\langle \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; D \rightarrow I; 1 \right\rangle; (10) \left\langle \begin{bmatrix} * & * & * \\ * & * & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; U \rightarrow H; 1 \right\rangle;$$

While agents apply programs with priority 1 (9) and (10), agents, that do not move in a cross direction, must stand. Therefore, they use a program composed of two rewriting rules. The programs have priority 2.

$$(11) \langle X \rightarrow X; 6 \rightarrow H; 2 \rangle; (12) \langle X \rightarrow X; 7 \rightarrow I; 2 \rangle; (13) \langle X \rightarrow X; 8 \rightarrow K; 2 \rangle; (14) \langle X \rightarrow X; 9 \rightarrow L; 2 \rangle;$$

The following programs with priority 0 are used to guide the agent in the next steps, the agent may hold information about the movement in the previous step.

$$\begin{aligned} (15) & \left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; H \rightarrow 9; 0 \right\rangle; (16) & \left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; H \rightarrow 8; 0 \right\rangle; \\ (17) & \left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; H \rightarrow U; 0 \right\rangle; (18) & \left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; H \rightarrow 7; 0 \right\rangle; \\ (19) & \left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; H \rightarrow D; 0 \right\rangle; (20) & \left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; H \rightarrow D; 0 \right\rangle; \\ (21) & \left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; H \rightarrow U; 0 \right\rangle; (22) & \left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; H \rightarrow U; 0 \right\rangle; \\ (23) & \left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; I \rightarrow 9; 0 \right\rangle; (24) & \left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; I \rightarrow 8; 0 \right\rangle; \\ (25) & \left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; I \rightarrow 6; 0 \right\rangle; (26) & \left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; I \rightarrow 7; 0 \right\rangle; \\ (27) & \left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; I \rightarrow D; 0 \right\rangle; (28) & \left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; I \rightarrow D; 0 \right\rangle; \\ (29) & \left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; I \rightarrow U; 0 \right\rangle; (30) & \left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; I \rightarrow U; 0 \right\rangle; \\ (31) & \left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; J \rightarrow 9; 0 \right\rangle; (32) & \left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; J \rightarrow L; 0 \right\rangle; \\ (33) & \left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; J \rightarrow 6; 0 \right\rangle; (34) & \left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; J \rightarrow 7; 0 \right\rangle; \end{aligned}$$



$$\begin{aligned}
(35) & \left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; J \rightarrow 7; 0 \right\rangle; (36) \left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; J \rightarrow D; 0 \right\rangle; \\
(37) & \left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; J \rightarrow 6; 0 \right\rangle; (38) \left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; J \rightarrow U; 0 \right\rangle; \\
(39) & \left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; K \rightarrow N; 0 \right\rangle; (40) \left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; K \rightarrow 8; 0 \right\rangle; \\
(41) & \left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; K \rightarrow 6; 0 \right\rangle; (42) \left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow; K \rightarrow 7; 0 \right\rangle; \\
(43) & \left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; K \rightarrow D; 0 \right\rangle; (44) \left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; K \rightarrow 7; 0 \right\rangle; \\
(45) & \left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; K \rightarrow U; 0 \right\rangle; (46) \left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; K \rightarrow 6; 0 \right\rangle;
\end{aligned}$$

We need one more program for “resetting” inertia. This is for the case when the slope of the terrain changes extremely. (47)  $\langle X \rightarrow X; N \rightarrow e; 0 \rangle$ ;

The next set of programs with priority 7 applies to the case that there are sinks in studied area. We add some number of copies of object  $V$  to every cell in the sink area. The number of the objects corresponds to the quantity of water which can be contained in the sink and the same number of agents have to be stopped here.

$$\begin{aligned}
(48) & \langle X \leftrightarrow V; I \rightarrow S; 7 \rangle; (49) \langle X \leftrightarrow V; J \rightarrow S; 7 \rangle; (50) \langle X \leftrightarrow V; K \rightarrow S; 7 \rangle; \\
(51) & \langle X \leftrightarrow V; L \rightarrow S; 7 \rangle;
\end{aligned}$$

If we run the obtained 2D P colony in the simulator, agents, which represent a unit volume of water, will begin to move around the environment. The number of agents located in one cell at one moment corresponds to the quantity of water that at once flowed through the territory in one unit of time.

The agents that “overflow” from the filled sinks move on the direction to the neighboring cell containing direction out of the sink. It is done by following programs with the priority set to 6.

$$\begin{aligned}
(52) & \left\langle \begin{bmatrix} \alpha & * & * \\ * & X & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; \beta \rightarrow U; 6 \right\rangle; \alpha \in \{a, E, i, 2\} \\
(53) & \left\langle \begin{bmatrix} * & \alpha & * \\ * & X & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow; \beta \rightarrow 6; 6 \right\rangle; \alpha \in \{i, E, a, y, 2\} \\
(54) & \left\langle \begin{bmatrix} * & * & \alpha \\ * & X & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; \beta \rightarrow U; 6 \right\rangle; \alpha \in \{y, 2, i, a, E\}
\end{aligned}$$

$$\begin{aligned}
(55) & \left\langle \begin{bmatrix} * & * & * \\ \alpha & X & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow; \beta \rightarrow 8; 6 \right\rangle; \alpha \in \{E, i, m, 2, u\} \\
(56) & \left\langle \begin{bmatrix} * & * & * \\ * & X & \alpha \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow; \beta \rightarrow 9; 6 \right\rangle; \alpha \in \{a, i, m, y, q\}; \\
(57) & \left\langle \begin{bmatrix} * & * & * \\ * & X & * \\ \alpha & * & * \end{bmatrix} \rightarrow \Leftarrow; \beta \rightarrow D; 6 \right\rangle; \alpha \in \{E, i, m, 2, u\} \\
(58) & \left\langle \begin{bmatrix} * & * & * \\ * & X & * \\ * & \alpha & * \end{bmatrix} \rightarrow \Downarrow; \beta \rightarrow 7; 6 \right\rangle; \alpha \in \{a, E, m, q, u\} \\
(59) & \left\langle \begin{bmatrix} * & * & * \\ * & X & * \\ * & * & \alpha \end{bmatrix} \rightarrow \Rightarrow; \beta \rightarrow D; 6 \right\rangle; \alpha \in \{a, i, m, y, q\}; \beta \in \{I, J, K, L\}
\end{aligned}$$

## 6 The example simulation

The example visualization is based on data from FreeGoedataCZ. A final statistics is done over four different data sets - four different locations.

The processed map is map of area with sink and its size is  $10 \times 10$  and its directions are shown on the Table 3. Transcoded symbols are shown on the Table 4.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ | ↑ |
| 1 | ← | ← | ↘ | ↓ | ↙ | ↗ | ← | ↘ | ↘ | → |
| 2 | ← | ← | ↘ | ↓ | ↙ | ← | ← | → | → | → |
| 3 | ← | ← | ↓ | ↓ | ← | ← | ↗ | ↓ | → | → |
| 4 | ← | ↓ | ↙ | ↙ | ← | ← | ↗ | ↓ | ↙ | → |
| 5 | ← | ↓ | ↓ | ↙ | ↙ | ↙ | ↙ | ↓ | ↙ | → |
| 6 | ← | → | ↘ | ↓ | ↓ | ↙ | ↙ | ↙ | ← | → |
| 7 | ← | ↑ | → | ↘ | ↓ | ↙ | ↙ | ↙ | ↙ | → |
| 8 | ← | ↗ | ↑ | → | ↓ | ← | ↙ | ← | ↙ | → |
| 9 | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ | ↓ |

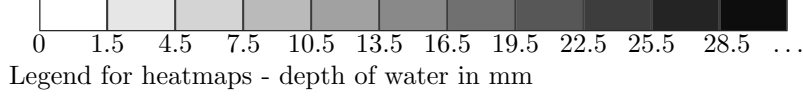
**Table 3.** Processed map

|   | 0        | 1        | 2        | 3        | 4        | 5        | 6        | 7        | 8        | 9        |
|---|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| 0 | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> | <i>i</i> |
| 1 | <i>E</i> | <i>E</i> | <i>q</i> | <i>m</i> | <i>u</i> | <i>u</i> | <i>E</i> | <i>q</i> | <i>q</i> | <i>a</i> |
| 2 | <i>E</i> | <i>E</i> | <i>q</i> | <i>m</i> | <i>u</i> | <i>E</i> | <i>E</i> | <i>a</i> | <i>a</i> | <i>a</i> |
| 3 | <i>E</i> | <i>E</i> | <i>m</i> | <i>m</i> | <i>E</i> | <i>E</i> | <i>u</i> | <i>m</i> | <i>a</i> | <i>a</i> |
| 4 | <i>E</i> | <i>m</i> | <i>u</i> | <i>u</i> | <i>E</i> | <i>E</i> | <i>u</i> | <i>m</i> | <i>u</i> | <i>a</i> |
| 5 | <i>E</i> | <i>m</i> | <i>m</i> | <i>u</i> | <i>u</i> | <i>u</i> | <i>u</i> | <i>m</i> | <i>u</i> | <i>a</i> |
| 6 | <i>E</i> | <i>a</i> | <i>q</i> | <i>m</i> | <i>m</i> | <i>u</i> | <i>u</i> | <i>u</i> | <i>E</i> | <i>a</i> |
| 7 | <i>E</i> | <i>i</i> | <i>a</i> | <i>q</i> | <i>m</i> | <i>u</i> | <i>u</i> | <i>u</i> | <i>u</i> | <i>a</i> |
| 8 | <i>E</i> | <i>y</i> | <i>i</i> | <i>a</i> | <i>m</i> | <i>E</i> | <i>u</i> | <i>E</i> | <i>u</i> | <i>a</i> |
| 9 | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> | <i>m</i> |

**Table 4.** Transcoded symbols

A source of water is placed into cells  $[2, 3]$ ,  $[3, 3]$ ,  $[4, 3]$ ,  $[2, 4]$ ,  $[3, 4]$ ,  $[4, 4]$ ,  $[2, 5]$ ,  $[3, 5]$ ,  $[4, 5]$ . In every source cell there are 8 agents. To simulate rain all agents are not active in the initial configuration. Only one agent has the configuration of *Xe* in each cell. The next nineteen become active always in two computational steps. The numbers of active agents in the environment are shown in the Tables 5(A) - 10(A) - the first column of tables (heatmaps). The tables (heatmaps) in

the second column (Tables 5(B)-10(B) show corresponding results achieved from SIMWE algorithm.



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 2 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2   | 3   | 4   | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|-----|-----|-----|---|---|---|---|---|
| 0 | * | *   | *   | *   | *   | * | * | * | * | * |
| 1 | * | 0   | 0   | 0   | 0   | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0   | 0   | 0   | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3 | 4.1 | 3.7 | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 8.0 | 9.2 | 4.8 | 0 | 0 | 0 | 0 | * |
| 5 | * | 3.4 | 7.1 | 5.1 | 4.8 | 0 | 0 | 0 | 0 | * |
| 6 | * | 0.7 | 4.7 | 4.1 | 0.8 | 0 | 0 | 0 | 0 | * |
| 7 | * | 0   | 0.5 | 0.9 | 0.5 | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 0   | 0   | 0   | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *   | *   | *   | * | * | * | * | * |

**Table 5.** (A) Active agents after  $2^{nd}$  step of computation, (B) raster data - depth of water after 4 minutes of rainfall (in mm)

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 5 | 5 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 2 | 4 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2    | 3    | 4   | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|------|------|-----|---|---|---|---|---|
| 0 | * | *   | *    | *    | *   | * | * | * | * | * |
| 1 | * | 0   | 0    | 0    | 0   | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0    | 0    | 0   | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3  | 4.1  | 3.7 | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 14.8 | 15.3 | 4.3 | 0 | 0 | 0 | 0 | * |
| 5 | * | 1.7 | 12.1 | 5.6  | 4.8 | 0 | 0 | 0 | 0 | * |
| 6 | * | 6.2 | 13.7 | 4.2  | 1.8 | 0 | 0 | 0 | 0 | * |
| 7 | * | 0   | 0.7  | 4.7  | 0.5 | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 0    | 0    | 0   | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *    | *    | *   | * | * | * | * | * |

**Table 6.** (A) Active agents after  $4^{th}$  step of computation, (B) raster data - depth of water after 8 minutes of rainfall (in mm)

We compare the simulation process using 2D P colonies and using SIMWE algorithm.

One agent corresponds to 3 mm of water and two steps of computation take 4 minutes. From previous tables we can derive following results: At the beginning of simulation the agents move more slowly than water over the surface but at the second half of simulation the agents move more quickly than water. Area touched by water is larger in SIMWE simulation but depth of water in these cells is only about 1 mm. The graphical representation of the frequency of depth

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 4 | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 4 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 3 | 5 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2    | 3    | 4   | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|------|------|-----|---|---|---|---|---|
| 0 | * | *   | *    | *    | *   | * | * | * | * | * |
| 1 | * | 0   | 0    | 0    | 0   | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0    | 0    | 0   | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3  | 4.1  | 3.7 | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 11.9 | 10.0 | 4.3 | 0 | 0 | 0 | 0 | * |
| 5 | * | 2.7 | 13.8 | 5.7  | 4.8 | 0 | 0 | 0 | 0 | * |
| 6 | * | 9.5 | 16.6 | 8.2  | 1.8 | 0 | 0 | 0 | 0 | * |
| 7 | * | 0.5 | 1.6  | 6.8  | 1.5 | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 1.6  | 8.3  | 0   | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *    | *    | *   | * | * | * | * | * |

**Table 7.** (A) Active agents after 6<sup>th</sup> step of computation, (B) raster data - depth of water after 12 minutes of rainfall (in mm)

|   | 0 | 1 | 2  | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|----|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1  | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 3  | 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 1 | 3  | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 1 | 10 | 2 | 0 | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0  | 4 | 3 | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0  | 0 | 4 | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0  | 0 | 3 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2    | 3    | 4    | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|------|------|------|---|---|---|---|---|
| 0 | * | *   | *    | *    | *    | * | * | * | * | * |
| 1 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3  | 4.1  | 3.7  | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 8.9  | 10.0 | 4.3  | 0 | 0 | 0 | 0 | * |
| 5 | * | 2.7 | 12.8 | 5.7  | 4.8  | 0 | 0 | 0 | 0 | * |
| 6 | * | 5.5 | 32.3 | 7.4  | 1.8  | 0 | 0 | 0 | 0 | * |
| 7 | * | 0.8 | 1.3  | 15.6 | 10.6 | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 1.6  | 2.1  | 13.2 | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *    | *    | *    | * | * | * | * | * |

**Table 8.** (A) Active agents after 8<sup>th</sup> step of computation, (B) raster data - depth of water after 16 minutes of rainfall (in mm)

|   | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|----|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 2 | 1  | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 2 | 4 | 1  | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 2 | 4 | 1 | 1  | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 2 | 6 | 4 | 0  | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 8 | 1  | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 4  | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 10 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2    | 3    | 4    | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|------|------|------|---|---|---|---|---|
| 0 | * | *   | *    | *    | *    | * | * | * | * | * |
| 1 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3  | 4.1  | 3.7  | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 6.9  | 12.0 | 4.3  | 0 | 0 | 0 | 0 | * |
| 5 | * | 3.7 | 13.8 | 5.7  | 4.8  | 0 | 0 | 0 | 0 | * |
| 6 | * | 6.2 | 17.4 | 14.4 | 1.8  | 0 | 0 | 0 | 0 | * |
| 7 | * | 0.5 | 1.8  | 25.0 | 4.9  | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 0.6  | 1.5  | 13.5 | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *    | *    | *    | * | * | * | * | * |

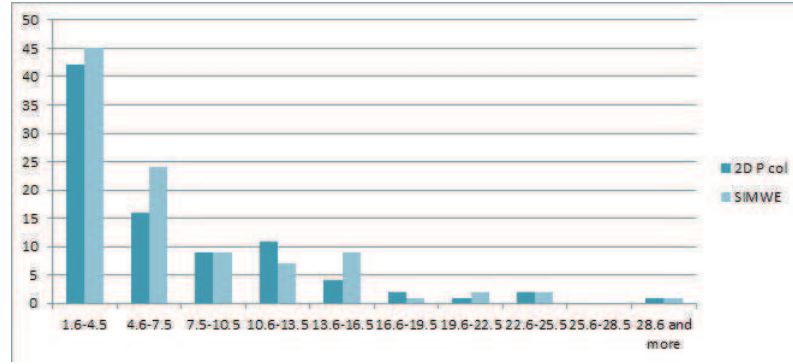
**Table 9.** (A) Active agents after 10<sup>th</sup> step of computation, (B) raster data - depth of water after 20 minutes of rainfall (in mm)

of water is shown on the Figure 1. The models give different results in 8.854 percent of cells in whole simulation.

|   | 0 | 1 | 2 | 3 | 4  | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|----|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 0  | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 2 | 1  | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 2 | 4 | 1  | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 2 | 5 | 1 | 1  | 0 | 0 | 0 | 0 | 0 |
| 6 | 0 | 2 | 7 | 2 | 0  | 0 | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 6 | 3  | 0 | 0 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 | 8  | 0 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 | 15 | 0 | 0 | 0 | 0 | 0 |

|   | 0 | 1   | 2    | 3    | 4    | 5 | 6 | 7 | 8 | 9 |
|---|---|-----|------|------|------|---|---|---|---|---|
| 0 | * | *   | *    | *    | *    | * | * | * | * | * |
| 1 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 2 | * | 0   | 0    | 0    | 0    | 0 | 0 | 0 | 0 | * |
| 3 | * | 0   | 3.3  | 4.1  | 3.7  | 0 | 0 | 0 | 0 | * |
| 4 | * | 0   | 6.9  | 12.0 | 4.3  | 0 | 0 | 0 | 0 | * |
| 5 | * | 3.7 | 15.8 | 5.7  | 4.8  | 0 | 0 | 0 | 0 | * |
| 6 | * | 7.2 | 22.0 | 6.6  | 1.8  | 0 | 0 | 0 | 0 | * |
| 7 | * | 1.8 | 2.0  | 20.3 | 9.2  | 0 | 0 | 0 | 0 | * |
| 8 | * | 0   | 1.6  | 2.3  | 25.1 | 0 | 0 | 0 | 0 | * |
| 9 | * | *   | *    | *    | *    | * | * | * | * | * |

**Table 10.** (A) Active agents after 12<sup>th</sup> step of computation, (B) raster data - depth of water after 24 minutes of rainfall (in mm)



**Fig. 1.** Frequency of depth of water in 2D P colony and SIMWE

## 7 Conclusion

The aim of this paper was to analyse the situation and to create a 2D model P colonies that would simulate the flow of liquid over the Earth's surface, a phenomenon called Surface runoff. This process is very common in nature and accumulation of water leads to flash flooding or floods in general. Flow down of water on the surface is influenced by many factors: the surface slope, soil saturation, temperature, humidity, size of source and lots of others. We applied the slope of the terrain in the environment of 2D P colonies. Finally, we compared the process of the simulations with the results provided by the algorithm SIMWE, module of geographic information system software GRASS GIS. The simulation results differs in 8.854 percent of cells.

### *Remark 1.*

This work was partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by SGS/24/2013, SGS/7/2011 and by project OPVK no. CZ.1.07/2.2.00/28.0014.

## References

1. Cienciala, L., Ciencialová, L., Perdek, M.: 2D P colonies. In Csuhaĵ-Varjú et al. (eds.). CMC 2012, Springer, LNCS 7762, 2013, pp. 161–172.
2. Csuhaĵ-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, G.: *Cells in environment: P colonies*, Multiple-valued Logic and Soft Computing, 12, 3-4, 2006, pp. 201–215.
3. Eastman, R. J.: *IDRISI Andes Guide to GIS and Image Processing*, Clerk Lab. Clerk University, Worcester, MA, USA, 2006.
4. GRASS DEVELOPMENT TEAM. GRASS GIS: The world’s leading Free GIS software [online]. 1998, 25-May-2013 [cit. 2013-07-01]. <http://grass.osgeo.org/>
5. Julien, P. Y., Saghaĳan, B., Ogden, F. L.: Raster-based Hydrologic modeling of spatilly varied surface funoff. *Water Resources Bulletin*, 31(3), 1995, pp. 523 - 536.
6. Kelemen, J., Kelemenová, A.: *On P colonies, a biochemically inspired model of computation*. Proc. of the 6<sup>th</sup> International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH, Hungary, 2005, pp. 40–56.
7. Kelemen, J., Kelemenová, A., Păun, Gh.: *Preview of P colonies: A biochemically inspired computing model*. Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems, ALIFE IX (M. Bedau at al., eds.) Boston, Mass., 2004, pp. 82–86.
8. Mitas, L., Mitasova, H., Brown, W. M., Astley, M.: Interacting fields approach for evolving spatial phenomena: application to erosion simulationfor optimized land use. In *Proc. of the III. Int. Conf. On Integration of Environmental Modeling and GIS* (Goodchild, M. F. at al. eds.), Santa Barbara, USA, 1996
9. Păun, Gh.: *Computing with membranes*. Journal of Computer and System Sciences 61, 2000, pp. 108–143.
10. Păun, Gh.: *Membrane computing: An introduction*. Springer-Verlag, Berlin, 2002.
11. Păun, Gh., Rozenberg, G., Salomaa, A.: *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2009.
12. Torrents, B.: *Geosimulation*. John Wiley & Sons, 2004.
13. P systems web page: <http://psystems.disco.unimib.it>

# Implementation of P Systems by using Big Data Technologies

Alex Ciobanu<sup>1</sup>, Florentin Ipat<sup>2,1</sup>

<sup>1</sup>Department of Computer Science, Faculty of Mathematics and Computer Science,  
University of Pitesti, Str. Targu din Vale 1, 110040 Pitesti, Romania

<sup>2</sup>Department of Computer Science, Faculty of Mathematics and Computer Science,  
University of Bucharest, Str. Academiei nr.14, sector 1, Bucharest, Romania  
`alex.ciobanu@gmail.com`, `florentin.ipate@ifsoft.ro`

**Abstract.** Due to their inherent parallel and non-deterministic nature, P system implementations require vast computing and storage resources. This significantly limits their applications, even more so when the calculation of *all possible evolutions* of the P system is required. This article exposes the scalability possibilities available with the Big Data ecosystem for P systems implementations, using Map Reduce parallelism to build the P system computation tree. The Hadoop based implementation is then used for generating test suites for cell like P systems, in particular for context-dependent rule coverage testing. Our preliminary evaluations on a benchmark of automatically generated P systems confirms that the proposed approach scales well.

**Keywords:** P systems testing, Hadoop, P system computation tree, Map Reduce, Big Data, NoSQL

## 1 Introduction

Membrane computing, a field of research which studies distributed and parallel computing models called *P systems*, is a rapidly growing research area. Initially coined by Gheorghe Păun in [1], P systems have been studied from a computational and modelling perspective. Many variants have been introduced [2] and investigated, further a set of applications has been identified and modelled with such systems. P systems offer the possibility of modelling natural phenomena using a very natural and logical syntax. Unfortunately natural phenomena are inherently extremely complex and the simulation of P systems which model such phenomena inherit the complexity, therefore requiring significant computational power to process. At a certain point the computational power and storage capacity of a single machine is simply insufficient for the simulations and testing of such P system, at which point grid or clustered computing is considered. In an attempt to reuse established technologies for the computations of P systems we will show a method of using a Map Reduce framework and a NoSQL database in the simulation of P systems. These technologies (which at times fall under the blanket term of Big Data) are designed to leverage large scale commodity hardware clusters as massively scalable environment for parallel computing. We

will use Big Data technologies to compute a computation tree of a non deterministic P system, and show a potential application of such computations. Some theoretical work has been done in using Hadoop with P systems in [7], but no implementation has been attempted.

## 2 Preliminaries

### 2.1 Map Reduce

MapReduce [3] is a framework developed circa 2004 at Google in an attempt to deal with their very large scale data warehousing needs. Although the Google implementation of the Map Reduce ecosystem is proprietary, Doug Cutting from within the Apache foundation developed an open source implementation under the project name Hadoop. We use this implementation for our experiments. The Hadoop ecosystem has many subcomponents including a file system, coordination applications, meta programming languages and many other components. For the purposes of our discussion we will focus on the core map reduce functionality developed as a basis for distributed computation within applications. Map Reduce is conceptually based on functional programming paradigms or to be more specific two primitives (higher order functions) called map and reduce.

**Map** (in functional programming) is defined as a higher order function with type signature:  $map :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$ . In other words a function that acts upon a vector of data in one data domain  $\alpha$  and returns a vector of data in another data domain  $\beta$  having transformed from domain  $\alpha$  to domain  $\beta$  by the given transformational function. In more familiar syntax if we have an input vector  $A = [a_1, a_2 \dots a_n]$  and a function  $f$ , then  $map(A, f) = f(A) = A'$  where  $A' = [f(a_1), f(a_2) \dots f(a_n)]$ . The data type of the resulting vector does not have to match the data type of initial vector but the cardinality of the two vectors must be equal.

**Reduce** also referred to as fold, accumulate, compress, or inject (in functional programming) is defined as a higher order function with type signature  $reduce :: (\alpha \rightarrow \beta \rightarrow \beta) \rightarrow [\alpha] \rightarrow \beta$ . In other words a function that acts upon a vector and returns the aggregation of the elements within that vector, aggregating based on the provided function. If we had our vector  $V = [v_1, v_2 \dots v_n]$  and our reduce function  $g$ , then  $reduce(V, g) = v'$  where  $v' = g(v_1, g(v_2, (\dots g(v_{n-1}), v_n)))$  (assuming a right side evaluation). In this case the type of  $V$  and  $v'$  are the same but the cardinality of the input vector is  $n$  while the cardinality of the result is 1. At the same time the reduce function  $g$  must be associative, commutative, and distributive as to allow for random ordering in the reduce process. Although in this example the reduce is right evaluated, evaluation can happen in any direction and in any order.



**Map-Reduce** within the context of Hadoop deviates from this strict definition of functional programming in a couple of ways. Most notable is the format of the input and output of both the map and reduce function, which are defined as a tuple of order 2. These tuples also referred to as a Key Value pair  $\langle K, V \rangle$  are the basis of all interactions within Hadoop. The Map task takes input of a  $\langle K, V \rangle$  pair and produces  $i$   $\langle K, V \rangle$  pairs (where  $0 \leq i \leq n$ ). In the next step (reduce) all  $\langle K, V \rangle$  pairs where the key is identical are passed to the reduce function. Basically the input of a Reduce function is a key with a list of values  $\langle K, [V, V, V...] \rangle$ . In the reducer (the execution of a reduce task) all of the values are reduced together and a series of  $j$   $\langle K, V \rangle$  pairs (where  $0 \leq j \leq m$ ) are produced which are the output of the entire process. The output of one execution can now become the input of next run of the application in series of executions. The Map and Reduce processes are written in Java and the execution of a Map or Reduce task entails running the map or reduce function on a server in a cluster of machines. A Mapper is a server in the cluster which is running a map task at that particular instance in time. Upon initiation of the application all servers run map tasks until the entire input set has been exhausted. Afterwards all servers become reducers and run the reduce function until again all  $\langle K, V \rangle$  pairs produced by the mappers are exhausted. The resultant  $\langle K, V \rangle$  pairs produced by the reducers are written out to the file system. Since all of the processes are run on independent servers with very little shared between processes, clusters can scale up to thousands of servers.

To give an example of MapReduce we can look at the canonical example, word count. Word count calculates the number of occurrences of each word within a text. The Map task takes as input a chunk of the file (the key being the byte offset of the file and the value is the actual text in the file). The map task tokenizes the text and creates  $\langle K, V \rangle$  pairs where the key is the actual word and value is the number of occurrences of the word (1 initially). The reducer takes in each unique key (in our case word) and does a sum of the integer values associated with it, fundamentally doing a word count. Figure 2 exemplifies the case where we have two input files one with the text "hello world" and the other with the text "goodbye world"

## 2.2 NoSQL Database

NoSQL is actually a blanket term to describe a suite database technologies which are not compliant to standard relational paradigms. Most of the underlying concepts come from a Google article [4] which describes a technology for distributed databases. NoSQL comprises of many different database technologies including document orient databases, graph databases, columnar stores and the technology we used for our experiments Key-Value stores. Most NoSQL databases use a lightweight mechanism for storing and retrieving data in exchange for greater scalability and availability. Oracle NoSQL database (the implementation of NoSQL used for this article) has similar properties to a map or dictionary from computer science theory. The database is able to store a key

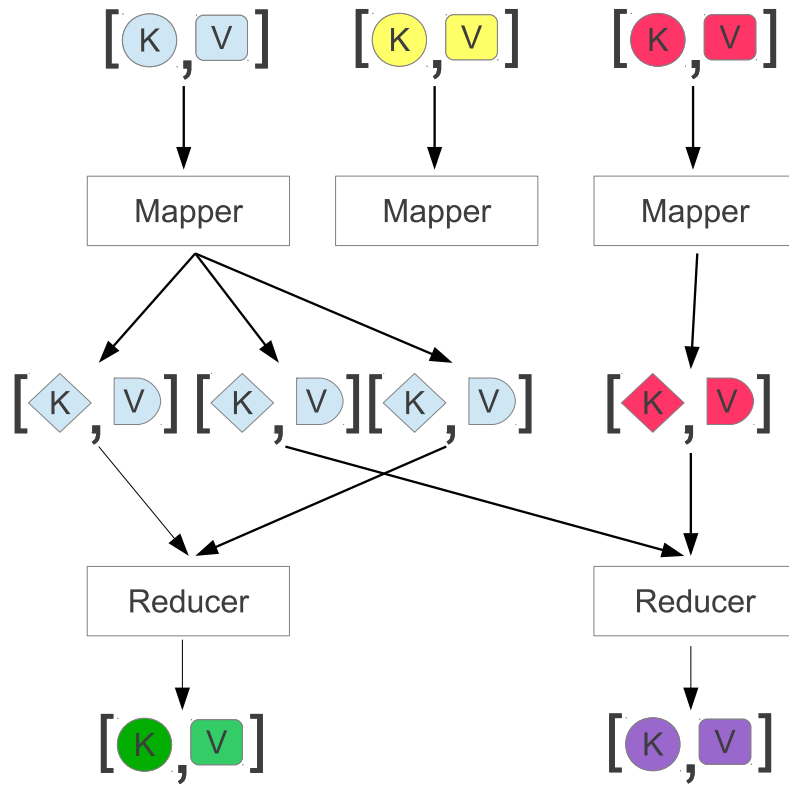


Fig. 1: Example of the flow of data with a Map-Reduce execution

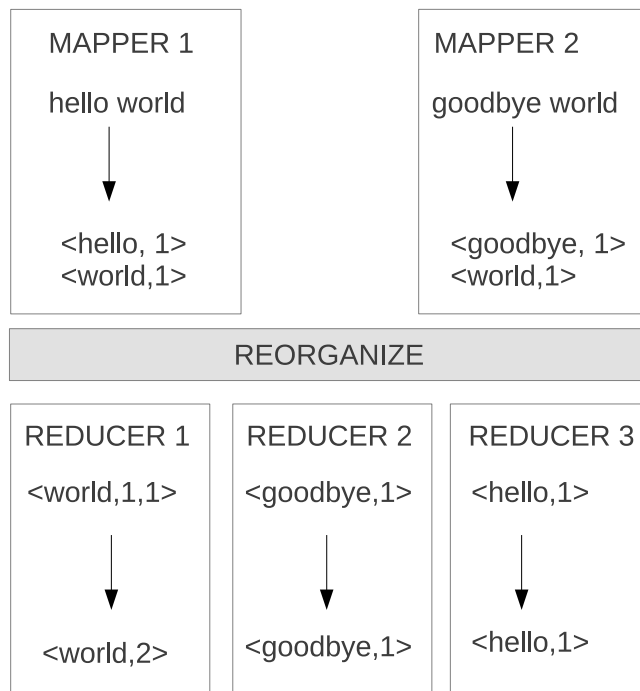


Fig. 2: Steps of Map-Reduce word count

value pair, and it's able to retrieve a value based on its corresponding key. In Oracle NoSQL database the keys are defined in a slightly more complex way. A key is composed of two components: a major component and a minor component, which are both a list of strings.

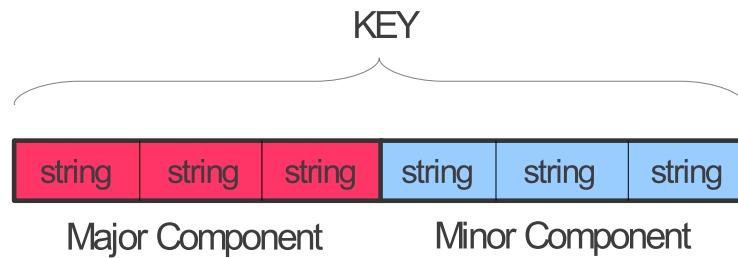


Fig. 3: A diagram of how a key is composed

When data is retrieved from the database, partial keys can be used allowing the retrieval of multiple values at the same time. A partial key (one that only contains the major component and  $i$  minor components where  $0 \leq i < n - 1$  and  $n$  is the number of minor components) is used to retrieve multiple keys which are logically linked and are processed together.

From a physical data storage perspective NoSQL uses a very similar architecture to a Hadoop cluster. NoSQL achieves its scalability and availability through a distributed storage mechanism. Multiple parallel servers are used to store the data. Usually the data is mirrored across multiple distinct servers. If a server is lost, the data can be retrieved from another server with the same data. At the same time if client requests data from an overloaded server a jump can be made to another server with lower utilization and the same data. Hashing algorithms are employed to eliminate the need for a linear search of all servers when retrieving data. In Oracle's NoSQL Database the major component of the keys is used as the indicator for which server to use, as an effort is made to keep all keys with identical major components together on one server. This enables faster multi-retrieve executions.

There is an important side effect of distributed databases relating to the consistency of data, or better said the lack thereof. NoSQL databases use a term called eventual consistency which states given a long enough period in which no updates occur, the system will reach a consistent state, but at a particular point in time there is no guarantee the system will be consistent. Given the distributed nature of the storage (usually triple mirroring) an update pushed to

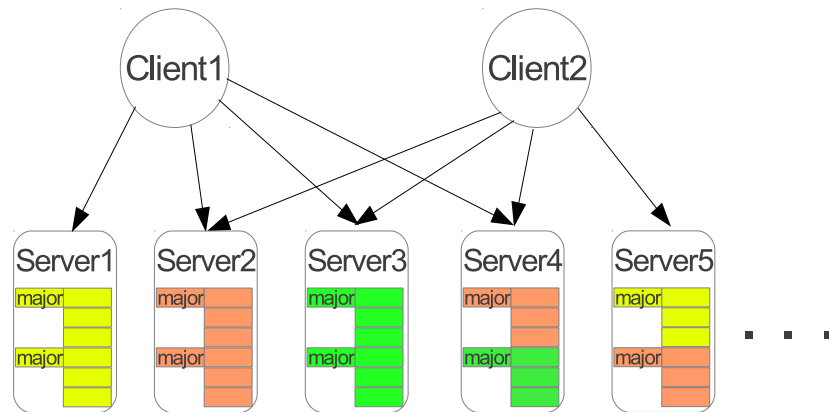


Fig. 4: A diagram of a NoSQL physical deployment

one sever is not guaranteed to propagate to all servers before a reading of the exact data point, hence a read might offer an old version of that data point. These limitations must be considered when designing an application against a NoSQL database. For more information see [9]

### 2.3 Parallelism

The subject of parallelism in computing can be split into many different categories including symmetric multiprocessing, grid computing massive parallel processing, grid computing and many more. The focus of this article will be exclusively grid computing, the distribution of work to multiple physical machines weakly linked through commodity networking, an architecture in which many of the worlds super computers share. This architecture allows for massive scaling (theoretically to unlimited number of processing units) while at the same time eliminating the need for exotic hardware which is both expensive and difficult to come by. An excellent overview and description of Parallelism in computing and the different can be found at [8]. Looking at the map and reduce function from a parallelism perspective it is quite natural that they distribute very nicely. Looking at the map function there is no link or sharing between the mapping of individual elements of a vector hence the map function can be executed on a different node of a cluster for each element of a vector with linear scaling and no performance impact as the number of nodes increases (barring data movement issues). The reduce function shares a similar parallelism capability (assuming associativity, commutativity, and distributivity) as little as two elements can be reduced (combined) on each node of the cluster and (given a set of  $n$  unique keys) we can theoretically scale to a  $n$  node cluster. It is to note there is some com-

munications overheard as the data produced by a map task (with identical keys) needs to moved on a single node of the cluster to be able to run the reduce function. For practical purposes implementation is usually limited to thousands of nodes due to network limitation although larger implementations are suspected to exist at web 2.0 corporations.

## 2.4 P systems

A (cell like) P system is a computational model inspired by the chemical reactions across cell membranes. The formal definition with which we will be working is:

DEFINITION 2.41 *A P system is a tuple*

$$\Pi = (\mathcal{V}, \mu, \mathcal{W}_1, \dots, \mathcal{W}_n, \mathcal{R}_1 \dots \mathcal{R}_n)$$

where

- $\mathcal{V}$  is the alphabet (a finite and nonempty) set of objects
- $\mu$  is the membrane structure, a hierarchical arrangement of compartments named membranes identified by integer 1 to  $n$
- $\mathcal{W}_i$  where  $0 \leq i \leq n$  are strings over  $\mathcal{V}$ , describing the multisets of objects initially placed in the  $i$  regions of  $\mu$ .
- $\mathcal{R}_i$   $0 \leq i \leq n$  is a finite set of evolution rules for each region  $i$  where evolution rule  $r$  is of the form

$$r : u \rightarrow (a_1, t_1) \dots (a_n, t_n) \quad (1)$$

where  $u$  and  $a_i$  is a multiset over  $\mathcal{V}$ , and  $t_i$  is an element from  $\mu$ .  $t_i$  is limited to the current membrane, the region immediately outside the current region or any of the region immediately inside the current region.

Although many variations on P system exist, for the purposes of this article we will concern ourselves with only this very basic definition (above) to look at how Big Data technologies can help in handling the state explosion problem. Complications such as polarization can be added to the computations upon request. It is also important to note that although this definition can have P systems which can only have one possible evolution path, our focus will be on non-deterministic P system with multiple possible evolutions for every membrane, in every configuration.

## 2.5 Computation Tree

A computation tree is a directed acyclic graph representation of the evolutions of a P system. The graph has a single root node (which represents the initial multiset of the P system), and every edge represents a possible evolution of the P system. All subsequent nodes in the graph are possible evolutions of the P system where the edges leading to the node represent the rules which must be applied to reach that configuration. Our computation tree assumes maximally

parallel execution. For example if we had the following P system:

$$\Pi = (\mathcal{V}, \mu, \mathcal{W}_1, \mathcal{R}_1),$$

where

- $\mathcal{V} = \{a, b\}$
- $\mu = []^1$
- $\mathcal{W}_1 = a^2$
- $\mathcal{R}_1 = \{r_1 : a \rightarrow a, b; r_2 : a \rightarrow b\}$

We would see the computation tree in Figure 4:

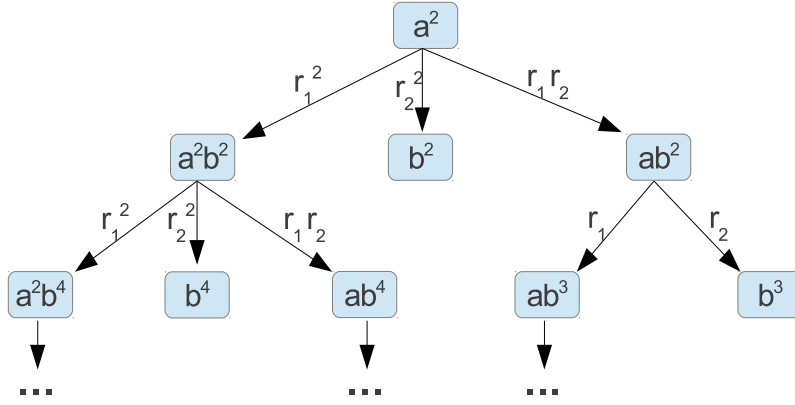


Fig. 5: A sample of a computation tree

### 3 Building a P system computation tree with Hadoop & NoSQL Database

In developing the computation tree of a P system we will be using the Oracle NoSQL database and Hadoop to facilitate a massively parallel calculation of the computation tree. The use of these technologies bring several complications as to ensure all relevant steps are parallelizable. In that we have developed the following distinct steps which are followed in sequence in order to calculate the computation tree. They are

1. Load the components  $(\mathcal{V}, \mu, \mathcal{W}_n, \mathcal{R}_n)$  of the P system into the NoSQL database

2. Calculate all possible rule combinations for each multiset at the current level of the computation tree
3. Calculate the multisets which are produced by applying the rule sequences discovered in step 2
4. Repeat Step 2 and 3 for subsequent levels of the computation tree

It is important to note that we are creating the computation tree in a breath first manner where all of the nodes for a level  $n$  are calculated before any of the nodes for level  $n + 1$  are discovered.

### 3.1 Representing a P system as a series of $\langle K, V \rangle$ pairs

A  $\langle K, V \rangle$  is a very simple model for storing data and as such there are theoretically many ways in which a P system can be represented as a series of pairs. For our implementation we focused on developing a model that is most conducive to the computation tree calculations we wanted to do. As such there was an explicit effort in using integers to represent elements of the P system rather than strings, as integer operations are much more efficient than string operations. Further integer representations allow for matrix and vector mathematics to be directly applied during our computations without the need to consider how strings would be handled. The second design decision was to group elements together by their use within our calculations and not by how they fit in logically within a P system. Given these design principles we used the following  $\langle K, V \rangle$  pairs to represent a P system  $\Pi = (\mathcal{V}, \mu, \mathcal{W}_1, \dots, \mathcal{W}_n, \mathcal{R}_1 \dots, \mathcal{R}_n)$ .

#### Alphabet $\mathcal{V}$

There is a single key which stores the alphabet. Its corresponding value is a Java serialized object which stores an array of strings representing the alphabet. This is the only place the actual alphabet is stored, and any further mention of an alphabet object is done through the integer index of this array. For example if our alphabet is  $\mathcal{V} = [\alpha, \beta, \gamma, \delta]$  then to refer to  $\alpha$  one would simply use the number 0, number 1 for  $\beta$ , number 2 for  $\gamma$  and so forth.

#### Membrane Structure $\mu$

The absolute membrane structure is not very interesting to our calculations, rather the children and parent membranes of each membrane is useful. As such for each membrane there are two  $\langle K, V \rangle$  pairs stored. One which stores the children of that membrane (as a Java serialized array of strings) and one which stores the parent membrane (as a simple string). Although storing both parents and children is redundant the overhead is minimal and it eliminates the need of searching relationship trees to discover ancestry. This is also a  $\langle K, V \rangle$  pair which holds a list of all membranes IDs (without any semantic information) to enable iteration through all membranes.

#### Rules $\mathcal{R}$

The rules are the most performance critical element of our application as they are used in many of the calculations that are done. The rules are grouped by membrane and split by sides of the equation, as that is how they will be



consumed. The rules are stored as integer matrices where each row represents a rule and each column represents an alphabet object. For example if we had

$$\begin{aligned}\mathcal{V} &= \{a, b, c\} \\ \mathcal{R}_1 &= a^1 c^2 \rightarrow a^2 b^1 \\ \mathcal{R}_2 &= b^2 \rightarrow a^2 c^1\end{aligned}$$

then the first matrix will be all an aggregation of all of the left sides of the rules. We call this the consumers

$$\text{consumers} = \begin{bmatrix} 1 & 0 & 2 \\ 0 & 2 & 0 \end{bmatrix}$$

The second matrix will be an aggregation of all of the right sides of the rules. We call this the producers

$$\text{producers} = \begin{bmatrix} 2 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$$

For each membrane there will be these two matrices stored as Java serialized objects of two dimensional arrays. The decision to split the rules into left side and right side was made out of the realization that these two elements will be used independently of each other. When dealing with rules which produces objects in multiple membranes we transform the matrix into a cube where the third dimension maintains a list of all relevant membranes.

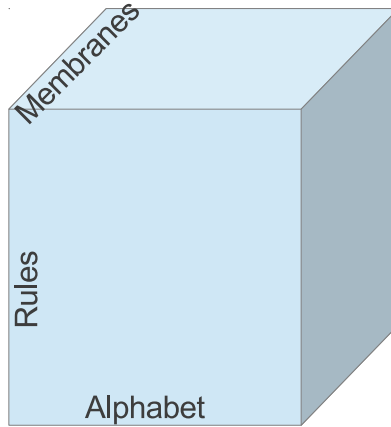


Fig. 6: Cube representation of the rules of a membrane

This storage schema assumes a dense coding of the objects and is very efficient if most of the alphabet objects are used in every rule. If there is a very sparse use of objects within rules then this coding mechanism may use excess storage.

### Multisets

The multisets are stored as an array of integers, similar to the way rules are

stored. The index of a multiset array corresponding to an object from the alphabet and the integers stored represents the multiplicity of that object.

### 3.2 Storing a computation tree as a series of $\langle K, V \rangle$ pairs

#### Nodes

To represent a tree as a series of  $\langle K, V \rangle$  we use the smallest piece of information to store in each  $\langle K, V \rangle$  pair, particularly we store the multiset of a membrane. For P systems which have multiple membranes each membrane has its own node, so a configuration of the P system is actually comprised of multiple  $\langle K, V \rangle$  pairs. When storing a node of the computation tree the key under which it is stored contains a significant amount of meta-data. In defining the key we exploit the make-up of a key described in section 2.2. There are three different pieces of information stored in the key of a node.

1. The level of the computation tree this node corresponds to
2. The membrane of the P system this node corresponds to
3. An unique id for this particular configuration.

The first two make up the major component of the Key, while the third makes up the minor component:

Major Component: List (Level of tree , membrane number)

Minor Component: (Unique id)

It is important to note the Unique id does not uniquely identify a node in the computation tree, and is only unique in combination with the membrane number. For example if there are 5 membranes in the P system then there should be 5 different nodes with the identical Unique id, one for each membrane, and combined they make up one configuration. This is done so each  $\langle K, V \rangle$  pair in the database is the minimum unit for calculation, as the computation of a membrane is completely independent of all other membranes. This will fit in very nicely into the MapReduce tasks described in the next section.

#### Edges

For each node of the tree there are two additional data points stored in the database. These represent the meta-data which would normally be stored in the edges of the graph:

1. A list of all child configuration for each Unique ID
2. A list of all rules applied on a particular evolution

This information is stored separate to the tree nodes as it applies to multiple nodes simultaneously. Each node represents only one membrane from a configuration and it is trivial which membrane is the child of which membrane. This

mapping can only be done at the configuration level as there is a directly link between the parent and child of a configuration. The same applies to storing which rules were applied to go from one configuration to another. It is very difficult to separate which rules produced all of the objects in a particular membrane given membrane communication as such, the rules applied are per configuration not per node. As described in the previous section the Unique ID identifies a configuration so it is quite easy to store a  $\langle K, V \rangle$  where the key is the unique ID and the value is a Java serialized array of all the children or a list of rules applied. These two supplementary  $\langle K, V \rangle$  enable the traversing of the tree in a logical way.

### 3.3 Determining all possible evolutions

One of the most critical and performance intensive aspects of developing the computation tree is discovering the possible evolutions of a configuration (particularly when dealing with a non deterministic P system). This calculation is non parallelizable and the performance of the entire system is gated on the algorithm used to discover all possible evolutions. The applicability of a rule is context dependent (dependent on the particular configuration) hence reuse of calculation is difficult, and brute force evaluation is a linear search to a potentially very large set of all possible rule combinations. In this section we have developed two algorithms, one for the general case, and one optimized for a specific case.

#### General Case Algorithm

To calculate all of the possible evolutions of a P system from a given configuration for the general case where there is not apriori information about the rules within a membrane we use a brute force algorithm. This algorithm goes through each of the rules and discovers the maximum number of time a particular rule can be applied in a context independent space (i.e. ignoring all other rules). Next we calculate all of the possible vectors of rules which could possibly be applied. For a P system where the maximum time  $rule_i$  can be applied is  $max(r_i)$ , there should be at most  $\prod max(r_i)$  combinations. Once every possible combination is calculated each one of these vectors is tested for correctness and maximality. If they pass both criteria then they are stored in list of possible evolutions of the P system in that particular configuration. The algorithm for checking a possible vector is described next.

To describe our function we have the following definitions:

- $\mathcal{R}$  is the vector of rules in the membrane
- $\mathcal{X}$  is the vector of rules under test
- $\mathcal{M}$  is the configuration (multiset) of the membrane
- *applyAllRules* is a function which takes a vector of rules and returns the multiset resultant from applying those rules
- *applicable* check if rule  $r$  is applicable given the multiset  $s$

The algorithm is:

```

 $\mathcal{C} \leftarrow \text{applyAllRules}(\mathcal{R})$ 
if  $\mathcal{C} = \mathcal{M}$  then
  return maximal
else
  for all  $c \in \mathcal{C}$  do
    if  $|c_i| > |m_i|$  then
      return incorrect
    end if
  end for
  for all  $r \in \mathcal{R}$  do
    if  $\text{applicable}(r, \mathcal{M} - \mathcal{C})$  then
      return not maximal
    end if
  end for
  return maximal
end if

```

Once every possible combination of rules has been tested with this algorithm, the rules vectors (which return maximal) are the vectors which produce all possible evolutions of the P system from the specified configuration for that membrane.

### Special Case Algorithm

If we impose certain restriction on the acceptable rule, new solving mechanisms for finding all possible maximal combinations of rules become available. Similar approached have been tried by: [6]. To exemplify this approach:

We know that for a given multiset  $\mathcal{M}$  and a set of rules  $\mathcal{R}$  where  $r_i$  is of the form  $u \rightarrow (a_1, t_1) \dots (a_n, t_n)$  and  $|r_i|$  represents the number of times a rule  $i$  is applied,

$$\forall m \in \mathcal{M}, \Sigma |r_i| \leq |m| \quad (2)$$

where  $u$  of  $r_i$  contains  $m$

But if

$$\forall v \in \mathcal{V} \exists r \in \mathcal{R} : r = v \rightarrow \alpha \quad (3)$$

where  $\mathcal{V}$  is the alphabet of the P system and  $\alpha$  is an arbitrary vector over  $\mathcal{V}$  (in other words if every object in the multiset is consumed) then

$$\forall m \in \mathcal{M}, \Sigma |r_i| = |m| \quad (4)$$

Combine (4) with the fact that  $|r_i| \in \mathbb{N}$  and you get a system of linear equation which can be solved. The solutions to the system of equations represents all possible combinations of rules which satisfy the maximality requirements.

**Numerical Example:** If we had the following configuration:

$$\mathcal{V} = \{a, b, c\}$$

$$\mathcal{M} = \{a^4, b^5, c^3\}$$

$$\mathcal{R} = \begin{cases} r_1 = a^1, b^1 \rightarrow \alpha \\ r_2 = a^1, c^1 \rightarrow \alpha \\ r_3 = a^1 \rightarrow \alpha \\ r_4 = b^1 \rightarrow \alpha \\ r_5 = c^1 \rightarrow \alpha \end{cases} \quad \text{where } \alpha \text{ is any arbitrary multiset over } \mathcal{V}$$

By expanding the equation (4) we get

$$\begin{cases} |r_1| + |r_2| + |r_3| = 4 \\ |r_1| + |r_4| = 5 \\ |r_2| + |r_5| = 3 \end{cases}$$

This now becomes a problem of  $n$  equations and  $m$  unknowns, so to solve we will rewrite the equations as an augmented matrix.

$$\left[ \begin{array}{ccccc|c} 1 & 1 & 1 & 0 & 0 & 4 \\ 1 & 0 & 0 & 1 & 0 & 5 \\ 0 & 1 & 0 & 0 & 1 & 3 \end{array} \right]$$

If we perform Gaussian elimination on this matrix with the solution we get

$$\left[ \begin{array}{ccccc|c} 1 & 0 & 0 & 1 & 0 & 5 \\ 0 & 1 & 0 & 0 & 1 & 3 \\ 0 & 0 & 1 & -1 & -1 & -4 \end{array} \right]$$

From here we have two free variables, we will call them  $t_1$  and  $t_2$  and the solution is:

$$\begin{cases} r_1 = 5 - t_1 \\ r_2 = 3 - t_2 \\ r_3 = -t_1 - t_2 + 4 \\ r_4 = t_1 \\ r_5 = t_2 \end{cases}$$

which produces an infinite number of solution, but we know that  $|r_i| \in \mathbb{N}$  so we can add the following restrictions on  $t_1$  and  $t_2$

$$\begin{cases} 0 \geq t_1 \geq 5 \\ 0 \geq t_2 \geq 3 \\ t_1 + t_2 \leq 4 \end{cases}$$

and if we plug in all acceptable values for  $t_1$  and  $t_2$  into the solution matrix we get the 14 different possible evolutions of that particular configuration.

This algorithm is not exceptionally efficient as Gaussian elimination is  $O(n^3)$  but as rules in a membrane do not change through the evolution of the P system we can solve the equation for a generic multiset and then simply plug in the values when calculating all possible evolutions. This will significantly reduce the amount of time required to calculate all possible evolutions.

### 3.4 Determining next level's nodes

Once we have calculated all possible evolutions of a particular configuration of a P system then the calculation of the next level of the computation tree is quite straight forward. We follow the steps:

1. Take one possible rule application sequence (calculated in section 4.3)
2. Given the particular input set apply the rule combination and get the output multiset
3. Take that multiset and do a cross product with the multisets of all of the other membranes available for the unique ID
4. Break up the resultant configuration and store each node in a unique key in the database
5. Repeat for all rule application sequences and possible cross products with different membranes

Following these steps we are able to compute all of the children nodes for a particular configuration of the P system.

### 3.5 The Map Reduce implementation

Developing a computation tree for a P system requires the calculation of all possible evolutions of each node in the tree recursively. As each node's possible evolution is absolutely independent of another, its calculation can be performed independently and most importantly in parallel. To facilitate this parallelism we use the Map construct of the Hadoop infrastructure, as it allows us to parallelize very naturally this calculation. As the calculation of the next level's nodes requires the aggregation of multiple membrane's possible evolutions, the Reduce construct is used to perform this task. Each MapReduce cycle calculates one more level of the computation tree, and as multiple calls are made to the MapReduce infrastructure the output of one cycle becomes the input for the next cycle. In other words the Map task implements one of the algorithms described in section 4.3 and stores the results under the Unique Id of the configuration.

The Reduce task receives all of the results from the MAP task for a particular configuration (a list whose cardinality is equal to the number of membranes

in a configuration). In the Reduce task a cross product between the possible configurations of each membrane and stored the products as the nodes of the next level. For example if we have 3 membranes and each membrane has 4 possible evolutions, then we would store 192 nodes in the computation tree (assuming all of the configurations produce objects in all of the membranes). The cross product of all of the possible evolutions is  $4 \times 4 \times 4$  which is 64. Each of those configuration has objects in all three membranes, but in the computation tree a node only represents one membrane, hence for each configuration there will be 3 nodes stored in the database; therefore  $64 \times 3 = 192$  nodes stored.

## 4 Experimental Results

We developed several P systems of varied size to determine the time required to generate a computation three of  $n$  levels for a particular P system. We also vary the number of servers in the cluster to be able to get an idea of scaling possibilities. The cluster was composed of 16 servers each with a single core 2 duo processor 4 GB of Ram and a single hard disk on a SATA bus. It is important to note a single server in a modern Hadoop cluster can be more power then the sum of the 16 machines we used, as such the following results need to be taken within that context. Most of the 16 nodes were configured identically with on server dedicated to administrating the Hadoop cluster and another for the NoSQL database. These two services (Hadoop MapReduce server and NoSQL storage server) were run concurrently on the same physical machines with on core for each process.

### 4.1 Testing different algorithms

We experimented with the different algorithms described in section 4.3 to notice the performance difference between the algorithms. The experiment tried to find all of the possible evolutions of a single membrane given a different number of rules. The multiset used for each experiment was a vector with one of each element in the alphabet ( $\mathcal{V}$ ). This experiment was performed on a single machine. These are the results:

| Number Of Rules | Time for General Case (ms) | Time for Equation Solving (ms) |
|-----------------|----------------------------|--------------------------------|
| 10              | 15                         | 86                             |
| 20              | 1146                       | 188                            |
| 30              | 1915509                    | 891                            |
| 40              | 8hrs +                     | 30444                          |
| 50              | ???                        | 7808309 (2.16 hrs)             |

To understand the results there are a couple of practical notes to consider for the Equation Solving algorithm.

1. The calculation of the reduced row echelon form matrix was not taken into consideration for the total execution time, since it is calculated only once at

the loading of the P system into the database, hence not relevant in repeated executions.

2. Although the example in section 3.3 showed lower and upper bounds for each free variable more complex execution usually only provide lower bounds for the free variables. As such it required the calculation of theoretical upper bounds, and then doing a linear search through all combinations of free variables to check if they comply with restrictions imposed by the equations in the reduced matrix.
3. The use of the Equation Solving algorithm has a static set-up time which is why for small search sets the General Case algorithm is more efficient as there is no set-up required for that algorithm.

## 4.2 Testing Number of Nodes

We used some P systems with varying number of rules to test how many nodes we could store in the database and how much space would be required. All experiments were performed with simple mirroring of data across the cluster.

Our first test used a simple P system with 4 membranes, 3 objects and 5 rules per membrane. This P system had a high number of possible evolutions per membrane. Our experiment of running this P system had:

| Number Of Nodes | Storage Space | Execution time |
|-----------------|---------------|----------------|
| 65471486        | 84.9 GB       | 16.54 hrs      |

This experiment did not finish as Hadoop time-out started to disrupt the execution.

We did run the computation again without clearing the database to see if more nodes could be stored. The results were:

| Number Of Nodes | Storage Space | Execution time |
|-----------------|---------------|----------------|
| 77186334        | 105.4 GB      | 5.52 hrs       |

This execution was exited by the system due to database issues. We believe it might be configuration issues or networking issues but error logs did not provide any useful information.

We did also use a much larger P system with 5 membranes, 10 objects and 30 rules per membrane and the following results were achieved:

| Number Of Nodes | Storage Space | Execution time |
|-----------------|---------------|----------------|
| 40445334        | 90 GB         | 10.52 hrs      |

This execution also timed out due to system resource limitations.

These experiments were designed to show the size of the computation trees which could be stored in the database. The number of nodes stored is not the total number of nodes which exist in the computation tree but the number of nodes stored before system stability issues interfered. We can extract from those results we have about 770 thousand nodes per gigabyte, and 450 thousand nodes per gigabyte for the respective P systems. The storage requirements do not grow



perfectly proportionally with the number of objects in the multiset as metadata describing the edges of the tree is also stored in the database which is not affected by the alphabet size.

### 4.3 Testing with different numbers of servers

We will also vary the cluster size from 2 to 16 servers with the same P system to test the scaling factor use a P system of 3 membranes 10 objects and 25 rules per membrane. We observed the following results:

| Tree Level | Number Of Nodes | 2 Servers | 4 Servers | 8 Servers | 16 Servers |
|------------|-----------------|-----------|-----------|-----------|------------|
| 1          | 18411           | 2m 51s    | 1m 22s    | 1m 43s    | 1m 18s     |
| 2          | 1438047         | 55m 34s   | 24m 12s   | 12m 44s   | 7m 56s     |

This experiment really demonstrates the scaling factor of Hadoop where doubling the number of nodes effectively doubles the performance of the application. The results of the experiments were significant despite the cluster being composed of low power machines. Hadoop clusters have scaled to thousands of servers where each server was significantly more powerful than the machines used for these tests. This experiment also shows the potential variance of the system. The first line of the table required to derivation of a single node (the root node) hence no parallelism was possible. All results for the first derivation should be the same as the number of servers does not matter for a non parallelisable task. The variance in the numbers is because of unpredictable elements in the Hadoop infrastructure.

### 4.4 Variance in results

Hadoop (as an infrastructure) is designed for large scale deployment of a distributed system (100 - 1000 of servers), and there is a high potential for server failure, server slowdowns, and data loss, situations which Hadoop is designed to deal with. These include data replication across servers (to deal with server loss) and speculative execution to deal with individual server performance issues. As exact execution path are both unpredictable, the timing results provided in this article come with a potential error factor. Repeated experiments will produce the same results but with different execution times, although these deltas are usually within an acceptable margin.

## 5 Practical Uses

To demonstrate a practical use of this application we developed a tool which would perform (Context Dependent Rule Coverage) CDRC test oracle discovery using the computation tree stored on the database. CDRC is a testing strategy where all possible sequences of 2 rules which can sequentially occur during the simulation of a Psystem are tested. For more details on CDRC please see article [5]. The process of discovering tree walks, which cover context dependent rules, is as followed:

1. Go through all of the rules and discover the CDRC rule pairs. Store these inside the NoSQL database
2. Run a Map task which take as input a unique configuration at a specific level of the computation, the goes through all children nodes of that configuration and tests the applied rules of the two steps.
3. Each evolution of the P system (which covers a CDRC rule pair) is stored and the CDRC pair is removed from the database as it is discovered.

## 6 Previous Work

Many attempts have previously been made to generate P system simulators which use parallel computing with varying levels of success. These are all enumerated in: [8]. Possibly the most successful attempts have used speciality hardware to enable the simulation. These include GPU, FPGA, and Micro Controller implementations. These simulators usually run much faster (per server) then the one developed in this article, but they have two major drawbacks: they require speciality equipment and they are limited to the storage and processing capacity of a single device. Our solution is able to scale to multiple commodity machines extending the storage capacity to very large data sets. At the same time a possible integrated approach between Hadoop and GPU approach would be very interesting to attempt as it might enable a best of both worlds implementations were the drawbacks of both system can be counterbalanced.

There have also been attempts at using clustered computing, using either c++ and Message Passing Interface, or Java and remote method calls. Both methods showed great potential but were limited by the communication overhead of the implementations. Our approach uses a distributed database to enable our communication and a slightly different approach to simulation (given tour multi path approach). Although Hadoop is a more rigid infrastructure then the other used, its rigidity also mitigates some of the issues faced with bespoke clustering technologies.

## 7 Conclusion

In this article we have show how big data technologies can be used to massively extend the reach of our P system simulators and calculators. The use of these technologies constituted several conceptual elements:

1. The use of a NoSQL database to store the computation tree of a P system
2. The use of a Hadoop Map task to compute all possible evolutions of a membrane
3. Two different algorithms which can be used to compute the possible evolutions of a membrane
4. The use of a Hadoop Reduce task to simulate membrane communication with the context of developing a computation tree

5. The implementation of this code which scale to computing and storing millions of nodes of a computation tree within a distributed storage to allow sub second access to the data even on low grade hardware.
6. The explanation and implementation of a possible use of the computation tree in Context Dependent Rule Coverage testing.

We can now extend the use of Hadoop and NoSQL to empower P system to simulate real world problems and possibly find solutions as we now have a viable strategy for potentially unlimited scaling.

Further work will now be performed to extend the application both from a technology perspective and a P system perspective. We will extend the technology to allow for other NoSQL database to underpin the system as to allow for the use of server rental services. We will also try to extend in the type of P system which can be simulated including conditional rules. We will also look at using this technology for different practical purposes including different testing strategies.

For access to the source code for this application go to GitHub at URL: <https://github.com/alexciobanu/psystem>

**Acknowledgments** We would like to thank Cristi Stefan of University of Pitesti who has enabled all of the experiments done in the development of this code. He worked tirelessly to maintain the equipment and ensure it was available for experimentation.

This work was partially supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0688. We are indebted to the anonymous reviewers for their valuable comments and suggestions

## References

1. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
2. Păun, G., Rozenberg, G., Salomaa, A.: *The Oxford Handbook of Membrane Computing*. Oxford University Press (2010)
3. Jeffrey Dean and Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters, *Sixth Symposium on Operating System Design and Implementation* (2004)
4. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber: Bigtable: A Distributed Storage System for Structured Data, *Seventh Symposium on Operating System Design and Implementation* (2006)
5. R. Lefticaru, F. Ipate, M. Gheorghe: Model checking based test generation from P systems using P-lingua. *Romanian Journal of Information Science and Technology*, 13(2): 153-168, 2010. Special Issue on Membrane Computing containing selected papers from BWMC (2010)

6. Richelle Ann B. Juayong<sup>1</sup>, Francis George C. Cabarle<sup>1</sup>, Henry N. Adorna<sup>1</sup>, Miguel A. Martinez-del-Amor<sup>2</sup>: On the Simulations of Evolution-Communication P Systems with Energy without Antiport Rules for GPUs. 10th Brainstorming Week on Membrane Computing proceeding pg. 267 - 289 (2012)
7. L. Diez Dolinski, R. Nunez Hervas, M. Cruz Echeandia, and A. Ortega: Distributed Simulation of P Systems by Means of Map-Reduce: First Steps with Hadoop and P-Lingua. IWANN 2011, Part I, LNCS 6691, pp. 457-464 (2011)
8. Miguel Angel Martinez del Amor: Accelerating Membrane Systems Simulators using High Performance Computing with GPU. PHD thesis University of Seville (2013)
9. <http://docs.oracle.com/cd/NOSQL/html/index.html> Visited May 10 2013
10. [www.p-lingua.org](http://www.p-lingua.org) Visited May 10 2013

# On Counter Machines versus dP Automata<sup>\*</sup>

Erzsébet Csuhaj-Varjú<sup>1</sup> and György Vaszil<sup>2</sup>

<sup>1</sup> Department of Algorithms and Their Applications, Faculty of Informatics  
Eötvös Loránd University  
Pázmány Péter sétány 1/c, 1117 Budapest, Hungary  
`csuhaj@inf.elte.hu`

<sup>2</sup> Department of Computer Science, Faculty of Informatics  
University of Debrecen  
P.O. Box 12, 4010 Debrecen, Hungary  
`vaszil.gyorgy@inf.unideb.hu`

**Abstract.** Continuing the study of connections between classical and P automata variants, we show that dP automata, i.e., distributed systems of P automata, where the input multiset is mapped to the set of strings consisting of all permutations of its elements, are as powerful as the class of distributed systems of special restricted counter machine acceptors. These variants of counter machines read multisets (represented as sets of all permutations of their elements) and manipulate counters in a conventional manner.

## 1 Introduction

P automata are purely communicating P systems accepting strings in an automaton-like fashion. In the standard case they are based on antiport systems with promoters or inhibitors. The concept was introduced in [3, 4]; for a summary on P automata the interested reader is referred to Chapter 6, [13]. Elements of the language (over some alphabet) of a P automaton are obtained by some mapping of the multiset sequences which enter the system through the skin membrane during an accepting computation.

Studying simple, non-erasing mappings, it was shown that if the rules of the P automaton are applied sequentially, then the accepted language class is strictly included in the class of languages accepted by one-way Turing machines with a logarithmically bounded workspace (1LOGSPACE), or if the rules are applied in the maximally parallel manner, then the class of context-sensitive languages is obtained [1]. If the input mapping is defined in such a way that it maps a multiset to the set of strings consisting of all permutations of its elements, i.e., mapping  $f_{perm}$  is used, then a class of languages is obtained that is strictly included in the class of so-called restricted logarithmic space Turing machines [6]. (In the case of nondeterministic restricted logarithmic space Turing machines

---

<sup>\*</sup> Supported in part by the Hungarian Scientific Research Fund, “OTKA”, grant no. K75952, and by the European Union through the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project which is co-financed by the European Social Fund.

the actual workspace available for computation is in logarithmic accordance with the already consumed input.) To prove the statement, special variants of counter machines, called RCMA (restricted counter machine acceptors) and SRCMA (special restricted counter machine acceptors) were introduced making it possible to read multisets (represented as sets of all permutations of their elements) and manipulating counters in a conventional manner.

Motivated by communication complexity questions, the notion of a distributed P automaton (a dP automaton, in short) was introduced in [12]. Such a system consists of a finite number of component P automata which have their separate inputs and which may communicate with each other by means of special antiport-like rules. A string accepted by a dP automaton is defined as the concatenation of the strings accepted by the individual components during a computation performed by the system [12]. The generic variant of dP automata ([12]) uses the mapping  $f_{perm}$  to define its language, that is, a string accepted by a component P automaton is the concatenation of strings which are permutations of the objects of the multisets imported by the skin membrane during an accepting computation; all combinations are considered.

In the last two years, dP automata were studied in detail (see, for example [12], [8], [14], and [15,16]). It was shown that using the mapping  $f_{perm}$ , dP automata are strictly more powerful than P automata (with  $f_{perm}$ ), but the language family accepted by them is strictly included in the family of context-sensitive languages.

Investigations have been made to compare P and dP automata classes to classical or well-known classes of acceptors as well. Connections between dP automata and non-deterministic multi-head finite automata were studied in [5], based on the concepts of agreement languages of dP automata and the notion of a two-way dP automaton. The strong agreement language consists of all words which can be accepted in such a way that all components accept the same sequence of multisets. In case of weak agreement languages the accepted multiset sequences can be different, only the equality of the images of all accepted sequences is required. In [5], it was shown how the languages of non-deterministic one-way and two-way multi-head finite automata can be obtained as the agreement languages of one-way and two-way finite dP automata. (A dP automaton is finite if the number of its configurations is a finite number.)

Continuing this line of research, in this paper we show that the classes of concatenated and agreement languages of dP automata with mapping  $f_{perm}$  and working in the nondeterministic maximally parallel mode, are equal to the classes of concatenated and agreement languages of distributed systems of special restricted counter machine acceptors.

## 2 Preliminaries and Definitions

We assume the reader to be familiar with the basics of formal language theory and membrane computing; for details consult [18] and [13].

An alphabet is a finite non-empty set of symbols. For an alphabet  $V$ , we denote by  $V^*$  the set of all strings over  $V$ ; if the empty string,  $\lambda$ , is not included, then we use notation  $V^+$ . The length of a string  $x \in V^*$  is denoted by  $|x|$ . For any symbol  $a \in V$ ,  $|x|_a$  denotes the number of occurrences of the symbol  $a$  in  $x$ ; and for any set of symbols  $A \subseteq V$ , the number of occurrences of symbols from  $A$  in  $x$  is denoted by  $|x|_A$ .

A finite multiset over an alphabet  $V$  is a mapping  $M : V \rightarrow \mathbb{N}$  where  $\mathbb{N}$  is the notation for the set of non-negative integers;  $M(a)$  is said to be the multiplicity of  $a$  in  $M$ .  $M$  can also be represented by any permutation of a string  $w = a_1^{M(a_1)} a_2^{M(a_2)} \dots a_n^{M(a_n)} \in V^*$ , where if  $M(x) \neq 0$ , then there exists  $j$ ,  $1 \leq j \leq n$ , such that  $x = a_j$ . The set of all finite multisets over an alphabet  $V$  is denoted by  $V^\circ$ , and we use the notation  $V^*$  for denoting the set of nonempty (finite) multisets. The empty multiset is denoted by  $\lambda$  as in the case of the empty string. If confusion may arise, we make explicit whether we speak of a string or a multiset.

A P system is a structure of hierarchically embedded membranes (a rooted tree), each membrane (node) having a unique label and enclosing a region containing a multiset of objects. The outmost membrane (the root of the tree), called the skin membrane, is unique and usually labeled with 1. Each region (membrane) is associated with a set of rules over multisets of objects which are used for changing the configuration of the P system.

An antiport rule is of the form  $(u, in; v, out)$ ,  $u, v \in V^\circ$  for a finite set of objects  $V$ . If such a rule is applied in a region, then the objects of  $u$  enter from the parent region and in the same step, objects of  $v$  leave to the parent region. If only  $(u, in)$  or  $(u, out)$  is indicated, then we speak of symport rules. (Note that the meaning of the “in” tag in these rules is different from the meaning of the target indicator “in” in the rules of type  $u \rightarrow (v, in)$  used in other types of P systems.) Antiport rules can be associated with promoters or inhibitor multisets of objects, denoted by  $(u, in; v, out)|_z$ , or  $(u, in; v, out)|_{\bar{z}}$ ,  $z \in V^\circ$ . In the first case the rule can only be applied if the objects of the promoter multiset  $z$  are all present in the given region, in the second case, the rule can be applied if no element of  $z$  is present. Analogously, promoters or inhibitors can be added to symport rules as well. The environment is supposed to contain an unlimited supply of objects, thus if an antiport rule (with promoters or inhibitors) is to be applied in the skin region, then the requested multiset is always available to enter the system from the environment.

A *P automaton* (of degree  $k$ ) is a membrane system  $\Pi = (V, \mu, w_1, \dots, w_k, P_1, \dots, P_k)$  with object alphabet  $V$ , membrane structure  $\mu$ , initial contents (multisets) of the  $i$ th region  $w_i \in V^\circ$ ,  $1 \leq i \leq k$ , and sets of antiport rules with promoters or inhibitors  $P_i$ ,  $1 \leq i \leq k$ . Furthermore,  $P_1$  must not contain any rule of the form  $(a, in)$  where  $a$  is an object from  $V$ .

The configurations of the P automaton (the actual  $k$ -tuple of multisets of objects over  $V$  in the regions) can be changed by transitions. The transition is performed by applying rules according to the working mode of the P automaton. For simplicity, we consider only the non-deterministic maximally parallel

(working) mode, where as many rules are applied simultaneously in the regions at the same step as possible. Thus, a transition in the P automaton  $\Pi$  is  $(v_1, \dots, v_k) \in \delta_\Pi(u_0, u_1, \dots, u_k)$ , where  $\delta_\Pi$  denotes the transition relation,  $u_1, \dots, u_k$  are the contents of the  $k$  regions,  $u_0$  is the multiset entering the system from the environment, and  $v_1, \dots, v_k$ , respectively, are the contents of the  $k$  regions after performing the transition. A sequence of transitions starting from the initial configuration  $(w_1, \dots, w_k)$  is a computation.

In this way, there is a sequence of multisets which enter the system from the environment during the steps of any computation. If the computation is accepting, that is, if it halts, then this multiset sequence is called an accepted multiset sequence.

From any accepted multiset sequence over  $V$ , a string of the accepted language, that is, a string over some alphabet  $\Sigma$  is obtained by the application of a mapping  $f : V^\circ \rightarrow 2^{\Sigma^*}$ , mapping each multiset to a finite set of strings.

Let  $\Pi$  be a P automaton as above, and let  $f$  be a mapping  $f : V^\circ \rightarrow 2^{\Sigma^*}$  for some finite alphabet  $\Sigma$ . The *language* over  $\Sigma$  accepted by  $\Pi$  with respect to  $f$  is defined as

$$L(\Pi, f, \Sigma) = \{f(v_1) \dots f(v_s) \mid v_1, \dots, v_s \text{ is an accepted multiset sequence of } \Pi\}.$$

In [8] the authors consider P automata with  $f_{perm}$ . Since in this case  $\Sigma$  does not differ from  $V$ , we denote the accepted language by  $L(\Pi, f_{perm})$ . The class of languages accepted by  $\Pi$  automata defined by mapping  $f_{perm}$  is denoted by  $\mathcal{L}(\text{PA}, f_{perm})$ . We note that the first appearance of  $f_{perm}$  is in [7], where the so-called analyzing P system, a closely related concept to the P automaton was introduced, almost at the same time as [3].

A finite collection of P automata forms a distributed P automaton, a dP automaton, in short, introduced in [12]. A *dP automaton* (of degree  $n \geq 1$ ) is a construct  $\Delta = (V, \Pi_1, \dots, \Pi_n, R)$ , where  $V$  is the alphabet of objects;  $\Pi_i = (V, \mu_i, w_{i,1}, \dots, w_{i,k_i}, P_{i,1}, \dots, P_{i,k_i})$  is a P automaton of degree  $k_i \geq 1$ ,  $1 \leq i \leq n$ , called the  $i$ th component of the system;  $R$  is a finite set of rules of the form  $z_i | (s_i, u/v, s_j) | z_j$ ,  $1 \leq i, j \leq n$ ,  $i \neq j$ ,  $uv \in V^\Phi$ , called the set of inter-component communication (shortly, communication) rules of  $\Delta$ ;  $s_k$ ,  $1 \leq k \leq n$  denotes the skin membrane of  $\Pi_k$ . The multisets  $z_i, z_j$  are promoters (or inhibitors) associated to the rule which can be applied if region  $s_i$  and  $s_j$  contain (or not contain) the elements of the multisets  $z_i, z_j$ , respectively.

We say that  $\Delta$  accepts the  $n$ -tuple  $(\alpha_1, \dots, \alpha_n)$ , where  $\alpha_i$ ,  $1 \leq i \leq n$ , is a sequence of multisets over  $V$ , if the component  $\Pi_i$ , starting from its initial configuration, using the symport/antiport rules (with promoters or inhibitors) as well as the inter-component communication rules in the non-deterministic maximally parallel way, takes from the environment the multiset sequence  $\alpha_i$ ,  $1 \leq i \leq n$ , and  $\Delta$  eventually halts, i.e., enters an accepting configuration.

Analogously to single P automaton, we may associate a language to a dP automaton by using a mapping from the object multisets to an alphabet of symbols. In this paper we study systems with the mapping  $f_{perm}$ , see above. We note that in [3] the mapping to define the alphabet of the language of the dP automaton was considered in a more general manner.



The (*concatenated*) *language* of  $\Delta$  (introduced in [12]) with respect to the mapping  $f_{perm}$ , is defined as

$$L_{concat}(\Delta, f_{perm}) = \{w_1 \dots w_n \in V^* \mid w_i \in f_{perm}(v_{i,1}) \dots f_{perm}(v_{i,s_i}) \text{ and} \\ \alpha_i = v_{i,1} \dots v_{i,s_i}, 1 \leq i \leq n, \text{ for an } n\text{-tuple of} \\ \text{accepted multiset sequences } (\alpha_1, \dots, \alpha_n)\}.$$

In [5] two variants of languages based on agreement of the components were introduced, namely, the weak and strong agreement languages. The strong agreement language consists of all words which can be accepted in such a way that all components accept the same sequence of multisets. In weak agreement languages, however, the accepted multiset sequences can be different, only the equality of the images of all accepted sequences is required. Note that in the special case of  $f_{perm}$ , the two types of agreement languages coincide, since in general (considering multisets from  $V^\circ$  containing at least two different symbols  $a, b \in V$ ,  $a \neq b$  of the underlying alphabet), the sets of words obtained as the images of two multiset sequences under the permutation mapping are equal only if the multiset sequences themselves are also equal. Thus, to obtain a “weaker” requirement, similarly to the weak agreement languages for more general input mappings in [5], we will use here a variant of the notion defined as follows.

The (*weak*) *agreement language* with respect to the mapping  $f_{perm}$  is defined as

$$L_{agree}(\Delta, f_{perm}) = \{w \in V^* \mid w \in f_{perm}(v_{i,1}) \dots f_{perm}(v_{i,s_i}) \text{ for all } 1 \leq i \leq n, \\ \text{where } \alpha_i = v_{i,1} \dots v_{i,s_i}, 1 \leq i \leq n, \\ \text{and } (\alpha_1, \dots, \alpha_n) \text{ is an } n\text{-tuple of accepted multiset} \\ \text{sequences of } \Delta\}.$$

In the case of  $L_{concat}(\Delta, f_{perm})$ , the words accepted by the components are concatenated to obtain the words of the language accepted by the dP automaton. In the case of the agreement language  $L_{agree}(\Delta, f_{perm})$ , those words are accepted by the dP automaton which can be obtained as the image of the accepted multiset sequence of all of the components, or in other words, the language accepted by the distributed automaton is the intersection of the sets of words  $L_i$  obtained as the images of the accepted multiset sequences  $\alpha_i$  of an  $n$ -tuple of sequences  $(\alpha_1, \dots, \alpha_n)$  accepted by the system in a computation.

The classes of concatenated and weak agreement languages accepted by dP automata and defined by mapping  $f_{perm}$  are denoted by  $\mathcal{L}_{concat}(\text{dPA}, f_{perm})$  and  $\mathcal{L}_{agree}(\text{dPA}, f_{perm})$ .

*Example 1.* Let  $\Delta$  be a dP automaton  $\Delta = (\{a, b, c, d\}, \Pi_1, \Pi_2, \emptyset)$  with  $\Pi_i = (\{a, b, c, d\}, [ ]_1, d, P_i)$ ,  $1 \leq i \leq 2$ , where  $P_1 = \{(ab, in; d, out), (c, in; a, out)\}$ ,  $P_2 = \{(a, in; d, out), (bc, in; a, out)\}$ .

The systems  $\Delta$  has only one computation where  $\Pi_1$  and  $\Pi_2$  accept the following sequences of two multisets: In the sequence accepted by  $\Pi_1$ , the first multiset contains a symbol  $a$  and a symbol  $b$ , the second contains a symbol  $c$ .

In the sequence accepted by  $\Pi_2$ , the first multiset contains a symbol  $a$ , the second contains a symbol  $b$  and a symbol  $c$ . Thus,  $\Delta$  accepts the pair of sequences of multisets  $(\{a, b\}\{c\}, \{a\}\{b, c\})$ . (We enumerated the elements between curly brackets, as in the usual set notation.)

Then, the concatenated language of  $\Delta$  is  $L_{concat}(\Delta, f_{perm}) = \{abcabc, bacabc, abcacb, bacacb\}$ , while the agreement language is  $L_{agree}(\Delta, f_{perm}) = \{abc\}$ .

In the following we recall some notions concerning complexity classes used to characterize classes of languages accepted by P automata. We start with a notion from [1].

A nondeterministic Turing machine with a one-way input tape is *restricted logarithmic space bounded* if for every accepted input of length  $n$ , there is an accepting computation where the number of nonempty cells on the work-tape(s) is bounded by  $O(\log d)$  where  $d \leq n$  is the number of input tape cells already read, that is, the *distance* of the reading head from the left end of the one-way input tape. The class of languages accepted by such machines is denoted by  $\text{rLOGSPACE}$ .

The following two variants of counter machines were introduced in [6]: A *restricted  $k$ -counter machine acceptor*  $M$ , an RCMA in short, is a (nondeterministic) counter machine with  $k$  counters (holding non-negative integers) and a one-way read only input tape. Thus,  $M = (Q, \Sigma, k, \delta, q_0, F)$  for some  $k \geq 1$ , where  $Q$  is the set of internal states,  $\Sigma$  is the input alphabet,  $q_0 \in Q$  is the initial state,  $F \subseteq Q$  is the set of final states, and  $\delta : Q \times \Sigma^* \times C^k \rightarrow 2^{Q \times D^k}$ , where  $C = \{\text{zero}, \text{nonzero}\}$ , denoting the two types of observations the machine can make on its counters,  $D = \{\text{increment}, \text{decrement}, \text{none}\}$  denoting the operations the machine can execute on its counters. (Note that  $\delta$  is finitely defined, that is, defined for a finite subset of  $\Sigma^*$ , and a counter can be incremented/decremented by one at any computational step.) Moreover,

- the transition relation is defined in such a way that the reading head is able to read a finite multiset of symbols in one computational step in the following sense:  $\delta(q, x, \alpha) = \delta(q, y, \alpha)$  for each  $x, y \in \Sigma^*$  which represent the same multiset. Moreover,
- the sum of the values stored in the counters can only increase as much in one computational step as the number of symbols read in that same step, that is, for all  $(q', \beta) \in \delta(q, x, \alpha)$  we have  $|\beta|_{\text{increment}} - |\beta|_{\text{decrement}} \leq |x|$ .

A *special restricted  $k$ -counter machine acceptor*, an SRCMA in short, is a restricted  $k$ -counter machine acceptor  $M = (Q, \Sigma, k, \delta, q_0, F)$ , but in addition, the transition relation  $\delta$  is defined in such a way, that if the length of the string  $x$  read in one computational step is  $l$ , then the sum of the values stored in the counters can only increase at most as much as  $l - 1$  in the same computational step. Thus,

- for all  $(q', \beta) \in \delta(q, x, \alpha)$ , we have  $|\beta|_{\text{increment}} - |\beta|_{\text{decrement}} \leq |x| - 1$ .

The classes of languages accepted by RCMA and SRCMA are denoted by  $\mathcal{L}(\text{RCMA})$  and  $\mathcal{L}(\text{SRCMA})$ .

### 3 Distributed Systems of Counter Machine Acceptors and dP Automata

A *distributed system of special restricted counter machine acceptors*, a dSRCMA in short, is a system  $\mathcal{M} = (\Sigma, M_1, \dots, M_n, \delta_{\mathcal{M}})$  for some  $n \geq 1$ , where  $\Sigma$  is an alphabet,  $M_i = (Q_i, \Sigma, k, \delta_i, q_{i,0}, F_i)$  for  $1 \leq i \leq n$  are SRCMA, the components of the system, and  $\delta_{\mathcal{M}}$  is the communication relation, where if we denote  $Q = \bigcup_{i=1}^n Q_i$ , then  $\delta_{\mathcal{M}} : (Q \times C^k)^2 \rightarrow 2^{(Q \times D^k)^2}$  and, as above,  $C = \{zero, nonzero\}$ ,  $D = \{increment, decrement, none\}$ .

When a dSRCMA works, each component processes its own input in a parallel and synchronized manner. The components may use their own transition relations  $\delta_i$ ,  $1 \leq i \leq n$ , or when appropriate, they may communicate as described by  $\delta_{\mathcal{M}}$ . The relation  $\delta_{\mathcal{M}}$  governs the communication of the components as follows. Let us assume that the internal control and the counters of  $M_{i_1}$  and  $M_{i_2}$  for some  $1 \leq i_1, i_2 \leq n$  are in the configurations  $(q_1, \alpha_1)$  and  $(q_2, \alpha_2)$ ,  $q_1 \in Q_{i_1}$ ,  $q_2 \in Q_{i_2}$ , and  $\alpha_i \in C^k$ ,  $1 \leq i \leq 2$ , respectively. Now, if  $(q'_1, \beta_1, q'_2, \beta_2) \in \delta(q_1, \alpha_1, q_2, \alpha_2)$ , then the components change their internal states to  $q'_1$  and  $q'_2$ , respectively, and update their counter contents according to  $\beta_i \in D^k$ ,  $1 \leq i \leq 2$ , respectively. It is important to note, that  $\delta_{\mathcal{M}}$  is defined in such a way that the sum of the counter contents cannot increase during a communication step, that is, taking  $\beta_1$  and  $\beta_2$  from above, it holds that  $|\beta_1 \beta_2|_{increment} \leq |\beta_1 \beta_2|_{decrement}$ .

Let  $(w_1, \dots, w_n)$  be the  $n$ -tuple of words accepted by the components of the dSRCMA system  $\mathcal{M}$ . The *concatenated language* and the *agreement language* of  $\mathcal{M}$  are defined as

$$L_{concat}(\mathcal{M}) = \{w_1 \dots w_n \in \Sigma^* \mid (w_1, \dots, w_n) \text{ is an accepted } n\text{-tuple of words of } \mathcal{M}\},$$

and

$$L_{agree}(\mathcal{M}) = \{w \in \Sigma^* \mid (w, \dots, w) \text{ is an accepted } n\text{-tuple of words of } \mathcal{M}\}.$$

The classes of concatenated languages and agreement languages accepted by dSRCMA systems are denoted by  $\mathcal{L}_X(\text{dSRCMA})$  for  $X \in \{concat, agree\}$ .

Now we are going to show that dSRCMA systems and dP automata characterize the same class of languages.

**Lemma 1**  $\mathcal{L}_X(\text{dPA}, f_{perm}) \subseteq \mathcal{L}_X(\text{dSRCMA})$  for any  $X \in \{concat, agree\}$ .

*Proof.* Let  $L = L_X(\Delta, f_{perm})$ , for  $X \in \{concat, agree\}$ , and let  $\Delta$  be the dP automaton  $\Delta = (\Sigma, \Pi_1, \dots, \Pi_n, R)$ ,  $\Pi_i = (\Sigma, \mu_i, w_{i,1}, \dots, w_{i,m_i}, P_{i,1}, \dots, P_{i,m_i})$ , for  $1 \leq i \leq n$ . We construct a dSRCMA system  $\mathcal{M} = (\Sigma, M_1, \dots, M_n, \delta_{\mathcal{M}})$  such that  $L = L_X(\mathcal{M})$ .

The components  $M_i = (Q_i, \Sigma, k, \delta_i, q_{i,ini}, F_i)$  of  $\mathcal{M}$  are able to simulate the computations of the components of  $\Delta$  by keeping track of the number of different

objects in the different regions of  $\Pi_i$ .  $M_i$  has three counters for each symbol-region pair, these are called storage counters, temporary counters, and assistant counters; three additional counters for each symbol-component pair, plus three additional ones for each symbol and the environment, these are called output counters, input<sub>1</sub> counters, input<sub>2</sub> counters. In addition,  $M_i$  has  $d_i$  additional counters called input assistant counters where  $d_i$  is the maximal number of objects which can enter the skin membrane of  $\Pi_i$  from the environment or from an other component by the application of one antiport rule ( $d_i = \max(\{|v| \mid (u, out; v, in)|_z \in P_{i,1} \cup \{z_i \mid ((i, 1), u/v, (j, 1))|_{z_j} \in R \mid 1 \leq j \leq m\}\})$ ). Apart from these, the components may need a number of assistant counters in order to be able to perform basic arithmetic operations and to check the equality of counter values during the computation. In order to have an equal number of counters in each component, we can take the maximum of the sum of the values defined above as the number  $k$  of counters in any component. These counters are initially empty, so the number of objects in the initial configuration of  $M_i$  is recorded in the components  $(c_{i,1}, \dots, c_{i,k})$  of the internal state  $q_{i,ini} = (q_{i,0}, c_{i,1}, \dots, c_{i,k}) \in Q'_i \times \mathbb{N}^k$ .

The simulation of a computational step  $(v_1, \dots, v_{m_i}) \in \delta_{\Pi_i}(u_0, u_1, \dots, u_{m_i})$  of  $\Pi_i$  by  $M_i$  can be described as follows. First  $M_i$  nondeterministically chooses symport/antiport rules (with promoters or inhibitors) from the sets  $P_{i,j}$ ,  $1 \leq j \leq m_i$ , of  $\Pi_i$  or from the communication rules in  $R$ , then updates the counters which keep track of the configuration of  $\Pi_i$  according to the chosen rules. The storage counters corresponding to the region and the objects which leave the region are decremented with the necessary amount of objects, and the number of objects entering the region are added to the corresponding temporary counters. If objects are exchanged between the skin regions of different components by the use of a communication rule of  $R$ , then the dSRCMA system also uses a communicating transition (described by  $\delta_{\mathcal{M}}$ ) to increase the temporary counters corresponding to the exchanged symbols and the skin regions of the two components. If an object leaves to the environment from the skin region of  $\Pi_i$ , then the corresponding output counter of  $M_i$  is incremented.

(Note that the “counter components” of the internal states are also taken into account: their value and the value of the corresponding “real” counter together represent the number of objects in  $\Pi_i$ . When such an “internal counter” is decremented, then the increment of the necessary temporary counter also takes place in the corresponding “internal” version of that counter. This way this nondeterministic rule choosing and configuration modifying phase of the computation of  $M_i$  does not increase the overall sum of the values stored in the different counters.)

When this phase is finished,  $M_i$  checks whether the configuration change implied by the rules chosen above corresponds to the maximally parallel mode of rule application. This means that  $M_i$  must check the applicability of rules in each region, which can be done one by one, using the corresponding assistant counters to store the numbers which are subtracted from various counters during the process in order to be able to easily restore the original configuration when

the checking of the applicability of a rule fails. The check also includes the skin region, to make sure that the multiset leaving to the environment is also maximal.

After the checking of the maximality of the chosen rule set,  $M$  realizes the configuration change by updating the storage counters using the values from the temporary counters, and by simulating the entering of objects from the environment (corresponding to the ones that leave the skin region), which can be done as follows. The number and type of objects which are supposed to leave to the environment are recorded in the output counters of the component. First  $M_i$  chooses antiport rules  $(u, out; v, in)|_z \in P_{i,1}$  and decrements the output counters corresponding to the objects of  $u$  while incrementing the input assistant counters of the component. Now  $M_i$  reads  $|v|$  symbols from its input tape, and records them in the  $input_1$  counters, and also records the symbols of  $v$  (from the chosen antiport rule) in the  $input_2$  counters. This process can be repeated a number of times, and when it is finished, the SRCMA component  $M_i$  correctly simulated the entering of objects into the skin region of  $\Pi_i$  from the environment, if two conditions are satisfied: first, the output counters should be empty, and second, each  $input_1$  and  $input_2$  counter corresponding to the same symbol should hold the same value. The second requirement corresponds to the fact that the same multiset of objects was read from the tape of  $M_i$  (although, possibly in a different order) as can be imported from the environment into the skin membrane of  $\Pi_i$  using the antiport rules that were chosen previously by  $M_i$ .

After completing this phase of the computation,  $\mathcal{M}$  can start the simulation of the next computational step of  $\Pi$  in the same way as described above. Before continuing with the simulation,  $\mathcal{M}$  can check whether the current configuration is final or not, and decide to proceed or to stop accordingly. (A configuration is final if it is halting, thus, if no rule can be applied in any of the regions.)

Note that the input reading operations do not violate the requirement that the sum of the numbers stored in the counters of an SRCMA can only increase in a computational step as much as  $c - 1$ , where  $c$  is the number of symbols read in that step. This holds because at least one symbol left the simulated system, thus, at least once one of the output counters were decremented, and at the same time, one of the input assistant counters was incremented. This means that after decrementing the input assistant counter, it is possible to increment the  $input_1$  and  $input_2$  counters altogether by the value of  $|v|$ . This is sufficient, because we can store any value  $c = 2 \cdot j + l$ ,  $l \in \{0, 1\}$  by storing  $j$  in the counter and keeping track of  $l$  in the state of the finite control, thus, by increasing the sum of the overall counter contents by  $|v|$ , we can store two numbers which are both less or equal to  $|v|$ .

We have seen that the words obtained by permuting the elements of the multisets in the multiset sequences accepted by the components of the dP automaton  $\Pi$  coincided with the words which can be accepted by the components of the dSRCMA system  $\mathcal{M}$ . This means that  $\mathcal{L}_X(\text{dPA}, f_{perm}) \subseteq \mathcal{L}_X(\text{dSRCMA})$  for any  $X \in \{\text{concat}, \text{agree}\}$ .  $\square$

**Lemma 2**  $\mathcal{L}_X(\text{dSRCMA}) \subseteq \mathcal{L}_X(\text{dPA}, f_{perm})$  for any  $X \in \{\text{concat}, \text{agree}\}$ .

*Proof.* We show how a dSRCMA system  $\mathcal{M} = (\Sigma, M_1, \dots, M_n, \delta_{\mathcal{M}})$  with  $M_i = (Q_i, \Sigma, k, \delta_i, q_{i,0}, F_i)$ ,  $1 \leq i \leq n$ , can be simulated by a dP automaton. Let the transitions defined by  $(\bigcup_{i=1}^n \delta_i) \cup \delta_{\mathcal{M}}$  be labeled in a one-to-one manner by the set  $lab(\mathcal{M})$ , and let the simulating dP automaton be defined as  $\Delta = (V, \Pi_1, \dots, \Pi_n, R)$ . For any  $M_i = (Q_i, \Sigma, k, \delta_i, q_{i,0}, F_i)$ ,  $1 \leq i \leq n$ , we define  $\Pi_i = (V, \mu_i, w_{i,1}, \dots, w_{i,k+2}, P_{i,1}, \dots, P_{i,k+2})$  as follows.

The alphabet is  $V = \Sigma \cup \{q_0, C, D, E, F\} \cup \{B_{i,t}, t_1, t_2, t_3, t_4 \mid 1 \leq i \leq 6, t \in lab(\mathcal{M})\} \cup \{A_i A'_i \mid 3 \leq i \leq k+2\}$ , the membrane structure is  $\mu_i = [ [ ]_{i,2} [ ]_{i,3} \dots [ ]_{i,k+2} ]_{i,1}$ , and the rule sets with the initial membrane contents are as follows. (For easier readability, instead of the string notation, we denote the initial multisets by enumerating their elements between curly brackets, as in the usual set notation.)

$$w_{i,1} = \{q_0, C, D\},$$

$$P_{i,1} = \{(a, out; u, in)|_{t_1} \mid a \in \Sigma, t \in lab(\delta_i) \text{ is a transition of } M \text{ which reads a string representing } u \text{ from the input tape}\}$$

$$w_{i,2} = \{a, B_{j,t}, t_1, (t_2)^k, (t_3)^k, (t_4)^k \mid 1 \leq j \leq 6, t \in lab(\delta_i)\} \text{ where } a \text{ is some element of } \Sigma \text{ and } (t_j)^k \text{ denotes } k \text{ copies of the object } t_j,$$

$$P_{i,2} = \{(t_1 a, out; q_0 D, in) \mid a \in \Sigma, t \in lab(\delta_i) \text{ labels a transition from } q_{i,0} \in Q_i\} \cup \\ \{(B_{1,t} D (t_2)^k, out; t_1, in) \mid t \in lab(\delta)\} \cup \{(a, out)|_D \mid a \in \Sigma\} \cup \\ \{(B_{2,t} (t_3)^k, out; B_{1,t}, in), (B_{3,t} (t_4)^k, out; B_{2,t}, in), \\ (B_{4,t}, out; B_{3,t} (t_2)^k, in), (B_{5,t}, out; (t_3)^k B_{4,t}, in), \\ (B_{6,t}, out; (t_4)^k B_{5,t} C a, in), (s_1 a, out; B_{6,t} D, in) \mid t, s \in lab(\delta) \text{ where } s \text{ is a transition which can follow } t, a \in \Sigma\} \cup \\ \{(E, out; B_{6,t}, in) \mid t \in lab(\delta_i) \text{ is a transition leading to a final state of } M\} \cup \\ \{(a, in)|_C, (C, out) \mid a \in \Sigma\},$$

and for  $3 \leq j \leq k+2$ , let

$$w_{i,j} = \{A_j, A'_j, F, F\},$$

$$P_{i,j} = \{(A_j, out; t_2, in), (A'_j, out)|_{t_2}, (A_j A'_j, in), (F, in; F, out)\} \cup \\ \{(t_2 a, out; t_3, in), (t_2 F, out), (t_3, out; t_4, in), (t_4, out) \mid t \in lab(\delta_i) \text{ is a transition which decrements the value of counter } j-2\} \cup \\ \{(t_2, out; t_3, in), (t_3, out; t_4, in), (t_4, out; a, in) \mid t \in lab(\delta_i) \text{ is a transition which increments the value of counter } j-2\} \cup \\ \{(F a, out)|_{t_2}, (t_2, out; t_3, in), (t_3, out; t_4, in), (t_4, out) \mid t \in lab(\delta_i) \text{ is a transition which requires that the value of counter } j-2 \text{ is zero}\}.$$

Let also

$$R = \{(t_2 t_4 | ((i, 1), u/\lambda, (j, 1)) |_{t_2 t_4} \mid \text{ where } t \in \delta(\mathcal{M}) \text{ labels a transition}$$

which results in the increase of the sum of the counter  
 contents of  $\Pi_j$  by  $|u| \cup$   
 $\{_{t_2 t_4} | ((i, 1), \lambda/v, (j, 1)) |_{t_2 t_4} \mid \text{ where } t \in \delta(\mathcal{M}) \text{ labels a transition}$   
 which results in the increase of the sum of the counter  
 contents of  $\Pi_i$  by  $|v| \cup$ .

Each of the components  $\Pi_i$  of the system defined above has a skin region (region  $(i, 1)$ ), a region representing the finite control (region  $(i, 2)$ ), and  $k$  regions corresponding to the  $k$  counters of  $M_i$  (regions  $(i, j)$ ,  $3 \leq j \leq k+2$ , referred to as the counter regions). The counter regions represent the values stored in the counters of  $M_i$  with objects from  $\Sigma$ , region  $(i, j)$  contains as many such objects as the values stored in counter  $j-2$ . The object  $q_0$  present in the skin region in the initial configuration is exchanged for a symbol  $t_1$  for a transition symbol  $t \in \text{lab}(\delta_i)$  denoting a transition from the initial state.

The simulation of a computational step of  $M_i$  starts by having one terminal object  $a \in \Sigma$ , and a transition symbol  $t_1$  for some transition  $t \in \text{lab}(\delta_i) \cup \text{lab}(\delta_{\mathcal{M}})$  in the skin membrane. If  $t \in \text{lab}(\delta_i)$ , then the terminal object  $a$  is used by a rule  $(a, \text{out}; u, \text{in})|_{t_1}$  to import a multiset  $u \in \Sigma^\circ$  which is read by  $M_i$  during the transition  $t$ . Otherwise, if  $t \in \text{lab}(\delta_{\mathcal{M}})$  no symbols are imported from the environment. Now the transition symbol is moved back to region  $(i, 2)$ , and  $k$  copies of  $t_2$  (corresponding to the same transition, but indexed with 2) are exported to the skin region together with all the copies of objects from  $\Sigma$  which are not used inside the counter regions (these are stored in region  $(i, 2)$  until they are needed). In the next six steps, the values stored in the  $k$  counter regions are modified as necessary while the symbol  $B_{1,t}$  is changed to  $B_{6,t}$ , increasing its index by one in every step. If a counter needs to be decremented or checked for being zero, then the objects  $t_2$  enter and take with them a terminal object to the skin region or perform the zero check as necessary. Meanwhile  $k$  copies of  $t_3$  are released from region  $(i, 2)$  which continue the process by changing to  $t_4$  and then bringing in terminal objects to the counter regions when the counter in question needs to be incremented during transition  $t$ . If  $t \in \text{lab}(\delta_{\mathcal{M}})$  is a communication transition, then the a number of objects which are necessary to maintain the values of the counters as required by  $t$  are also transferred between the components using the communication rules of  $R$ . Such a rule can be applied only in the step when both  $t_2$  and  $t_4$  are present in the skin region.

After the modification of the counter values, the remaining terminal objects are transported back to region  $(i, 2)$ , and the symbol  $s_1 \in \text{lab}(\delta_i) \cup \text{lab}(\delta_{\mathcal{M}})$  for the next transition appears, together with exactly one terminal object  $a \in \Sigma$ , so the simulation of the next computational step of  $M_i$  can start in the same manner.

The simulation finishes when, after executing a transition leading to a final state of  $M_i$ , the symbol  $E$  is exported from region  $(i, 2)$  to the skin region and the component halts.

Note that the components of the SRCMA system read multisets in the sense that whenever  $(q', \beta) \in \delta(w, q, \alpha)$  for some  $w \in \Sigma^*$ , then also  $(q', \beta) \in \delta(\bar{w}, q, \alpha)$

where  $\bar{w}$  is any permutation of  $w$ . This means that the components of the dP automaton described above accept the same words as the components of the dSRCMA system, thus, they also accept the same concatenated or agreement languages.  $\square$

Combining the two lemmas above, we obtain the following

**Corollary 3**  $\mathcal{L}_X(\text{dPA}, f_{perm}) = \mathcal{L}_X(\text{dSRCMA})$ , for  $X \in \{\text{concat}, \text{agree}\}$ .

## 4 Conclusion

In this paper we have shown that dP automata with mapping  $f_{perm}$  are as powerful as the class of distributed systems of special restricted counter machine acceptors. Observing the proof and the concept of dSRCMA, the reader may easily notice that dSRCMA realize multi-head SRCMA in some sense, i.e., the (weak) agreement language of dP automata corresponds to the language of a multi-head SRCMA. We plan research in this direction, i.e., on the relation between one-way and two-way multi-head RCMA and SRCMA and languages of one-way and two-way dP automata in the future.

## References

1. E. Csuhaj-Varjú, O. H. Ibarra, and Gy. Vaszil. On the computational complexity of P automata. In C. Ferretti, G. Mauri, and C. Zandron, editors, *10th International Workshop on DNA Computing*, volume 3384 of *Lecture Notes in Computer Science*, pages 76–89. Springer, 2005.
2. E. Csuhaj-Varjú, M. Oswald, and Gy. Vaszil. P automata. In [13], chapter 6, pages 144–167.
3. E. Csuhaj-Varjú and Gy. Vaszil. P automata. In Gh. Păun and C. Zandron, editors, *Pre-Proceedings of the Workshop on Membrane Computing WMC-CdeA 2002, Curtea de Argeş, Romania, August 19-23, 2002*, pages 177–192. Pub. No. 1 of MolCoNet-IST-2001-32008, 2002.
4. E. Csuhaj-Varjú and Gy. Vaszil. P automata or purely communicating accepting P systems. In Gh. Păun, G. Rozenberg, A. Salomaa, and C. Zandron, editors, *Membrane Computing*, volume 2597 of *Lecture Notes in Computer Science*, pages 219–233. Springer, Berlin, 2003.
5. E. Csuhaj-Varjú and Gy. Vaszil. Finite dP Automata versus multi-head Finite Automata. In M. Gheorghe, Gh. Păun, G. Rozenberg, A. Salomaa, and S. Verlan, editors, *Membrane Computing*, volume 7184 of *Lecture Notes in Computer Science*, pages 120–138. Springer, Berlin, 2012.
6. E. Csuhaj-Varjú and Gy. Vaszil. On the power of P automata. In A. Dennunzio, L. Manzoni, G. Mauri, and A. E. Porreca, editors, *Unconventional Computation and Natural Computation 2013, Lecture Notes in Computer Science*, Springer, Berlin, to appear.
7. R. Freund and M. Oswald. A short note on analysing P systems. *Bulletin of the EATCS*, 78:231–236, 2002.



8. R. Freund, M. Kogler, Gh. Păun, and M. J. Pérez-Jiménez. On the power of P and dP automata. *Annals of Bucharest University Mathematics-Informatics Series*, LVIII:5–22, 2009.
9. O.H. Ibarra. Membrane hierarchy in P systems. *Theoretical Computer Science*, 334(1–3):115–129, 2005.
10. B. Monien. Two-way multi-head automata over a one-letter alphabet. *RAIRO - Informatique théorique/Theoretical Informatics* 14(1):67–82, 1980.
11. G. Păun. *Membrane Computing: An Introduction*, Natural Computing Series, Springer-Verlag, Berlin, 2002.
12. Gh. Păun and M. J. Pérez-Jiménez. Solving problems in a distributed way in membrane computing: dP systems. *International Journal of Computers, Communication and Control*, V(2):238–250, 2010.
13. G. Păun, G. Rozenberg and A. Salomaa. Eds.: *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
14. Păun, G., Pérez-Jiménez, M. J.: P and dP automata: A survey. In: *Rainbow of Computer Science* (C. S. Calude, G. Rozenberg, A. Salomaa, Eds.), vol. 6570 of *Lecture Notes in Computer Science*, Springer, Berlin, 2011, 102–115.
15. Păun, G., Pérez-Jiménez, M. J.: An infinite hierarchy of languages defined by dP systems. *Theoretical Computer Science*, 431:4–12, 2012.
16. Păun, G., Pérez-Jiménez, M. J.: P automata revisited. *Theoretical Computer Science*, 454:222–230, 2012.
17. M. J. Pérez-Jiménez. A computational complexity theory in membrane computing. In Gh. Păun, M. J. Pérez-Jiménez, A. Riscos-Núñez, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 5957 of *Lecture Notes in Computer Science*, pages 125–148, Berlin Heidelberg, 2010. Springer.
18. G. Rozenberg and A. Salomaa. Eds.: *Handbook of Formal Languages*, Springer-Verlag, Berlin, 1997.
19. Gy. Vaszil. On the parallelizability of languages accepted by P automata. In J. Kelemen and A. Kelemenová, editors, *Computation, Cooperation, and Life*, volume 6610 of *Lecture Notes in Computer Science*, pages 170–178. Springer, Berlin Heidelberg, 2011.



# Model Checking Kernel P Systems

Ciprian Dragomir<sup>1</sup>, Florentin Ipate<sup>2,3</sup>, Savas Konur<sup>1</sup>, Raluca Lefticaru<sup>2,3</sup>, and  
Laurentiu Mierla<sup>3</sup>

<sup>1</sup> Department of Computer Science, University of Sheffield  
Regent Court, Portobello Street, Sheffield S1 4DP, UK  
`c.dragomir@sheffield.ac.uk`, `s.konur@sheffield.ac.uk`

<sup>2</sup> Department of Computer Science, University of Bucharest  
Str. Academiei nr. 14, 010014, Bucharest, Romania  
`florentin.ipate@ifsoft.ro`, `raluca.lefticaru@fmi.unibuc.ro`

<sup>3</sup> Department of Mathematics and Computer Science, University of Pitești  
Str. Târgu din Vale 1, 110040, Pitești, Romania  
`laurentiu.mierla@gmail.com`

**Abstract.** Recent research in membrane computing examines and confirms the anticipated modelling potential of kernel P systems in several case studies. On the one hand, this computational model is destined to be an abstract archetype which advocates the unity and integrity of P systems onto a single formalism. On the other hand, this envisaged convergence is conceived at the expense of a vast set of primitives and intricate semantics, an exigent context when considering the development of simulation and verification methodologies and tools. Encouraged and guided by the success and steady progress of similar undertakings, in this paper we directly address the issue of formal verification of kernel P systems by means of model checking and unveil a software framework, *kpWorkbench*, which integrates a set of related tools in support of our approach. A case study that centres around the well known *Subset Sum* problem progressively demonstrates each stage of the proposed methodology: expressing a kP system model in recently introduced *kP-Lingua*; the automatic translation of this model into a Promela (Spin) specification; the assisted, interactive construction of a set of LTL properties based on natural language patterns; and finally, the formal verification of these properties against the converted model, using the Spin model checker.

## 1 Introduction

Membrane computing, the research field introduced by Gheorghe Păun [20], studies computational models, called P systems, inspired by the functioning and structure of the living cell. In recent years, significant progress has been made in using various types or classes of P systems to model and simulate systems and problems from many different areas [4]. However, in many cases, the specifications developed required the ad-hoc addition of new features, not provided in the initial definition of the given P system class. While allowing more

flexibility in modelling, this has led to a plethora of P system variants, with no coherent integrating view, and sometimes even confusion with regard to what variant or functioning strategy is actually used.

The concept of *kernel P system (kP system)* [7] has been introduced as a response to these problems. It integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, thus, provides a framework for formally analyzing these models. The expressive power and efficiency of the newly introduced kP systems have been illustrated by a number of representative case studies [6], [13]. Furthermore, the kP model is supported by a modelling language, called *kP-Lingua*, capable of mapping the kernel P system specification into a machine readable representation.

Naturally, formal modelling has to be accompanied by formal verification methods. In the membrane system context, formal verification has been approached, for example, using rewriting logic and the Maude tool [1] or PRISM and the associated probabilistic temporal logic [10] for stochastic systems [3]. Several, more recent, successful attempts to apply model checking techniques on transition P systems also exist [16], [17], [14]. However, to the best of our knowledge, there is no integrated formal verification approach to allow formal properties to be specified in a language easily accessible to the non-specialist user and to be automatically verified in a transparent way.

This paper proposes precisely such an integrated verification approach, which allows formal properties, expressed in a quasi-natural language using predefined patterns, to be verified against a kP-Lingua representation of the model using model checking techniques and tools (in this case the model checker Spin and the associated modelling language Promela). Naturally, this approach is supported by adequate tools, which automatically convert the supplied inputs (natural language queries and kP-Lingua representation) into their model checking specific counterparts (LTL queries and Promela representation, respectively). The approach is illustrated with a case study, involving a kP system solving a well-known NP-complete problem, the Subset Sum problem.

The paper is structured as follows: Section 2 recalls the definition of a kernel P system - the formal modelling framework central to our examination. We then review, in section 3, some of the primary challenges of model checking applicable to kP system models and discuss the transformations such a model must undergo, in order to be exhaustively verified by Spin. We also present our implemented approach to achieve an automatic model conversion, targeting the process meta language, *Promela*. In section 4, we address the complementary requirement of specifying system properties as temporal logic formulae. The section also includes an array of EBNF formal definitions which describe the construction of LTL properties that relate to kP system state constituents, a guided process which employs selected natural language query patterns.

Section 5 applies our proposed methodology, exemplifies and demonstrates all stages of the process with a case study - an instance of the Subset Sum problem. Finally, we conclude our investigation and review our findings in section 6.

## 2 Kernel P systems

A kP system is made of compartments placed in a graph-like structure. A compartment  $C_i$  has a type  $t_i = (R_i, \sigma_i)$ ,  $t_i \in T$ , where  $T$  represents the set of all types, describing the associated set of rules  $R_i$  and the execution strategy that the compartment may follow. Note that, unlike traditional P system models, in kP systems each compartment may have its own rule application strategy. The following definitions are largely from [7].

**Definition 1.** A kernel P (kP) system of degree  $n$  is a tuple

$$k\Pi = (A, \mu, C_1, \dots, C_n, i_0),$$

where  $A$  is a finite set of elements called objects;  $\mu$  defines the membrane structure, which is a graph,  $(V, E)$ , where  $V$  are vertices indicating components, and  $E$  edges;  $C_i = (t_i, w_i)$ ,  $1 \leq i \leq n$ , is a compartment of the system consisting of a compartment type from  $T$  and an initial multiset,  $w_i$  over  $A$ ;  $i_0$  is the output compartment where the result is obtained.

Each rule  $r$  may have a **guard**  $g$  denoted as  $r \{g\}$ . The rule  $r$  is applicable to a multiset  $w$  when its left hand side is contained into  $w$  and  $g$  is true for  $w$ . The guards are constructed using multisets over  $A$  and relational and Boolean operators. For example, rule  $r : ac \rightarrow c \{ \geq a^3 \wedge \geq b^2 \vee \neg > c \}$  can be applied iff the current multiset,  $w$ , includes the left hand side of  $r$ , i.e.,  $ac$  and the guard is true for  $w$  - it has at least 3  $a$ 's and 2  $b$ 's or no more than a  $c$ . A formal definition may be found in [7].

**Definition 2.** A rule can have one of the following types:

- (a) **rewriting and communication rule**:  $x \rightarrow y \{g\}$ ,  
where  $x \in A^+$  and  $y$  has the form  $y = (a_1, t_1) \dots (a_h, t_h)$ ,  $h \geq 0$ ,  $a_j \in A$  and  $t_j$  indicates a compartment type from  $T$  – see Definition 1 – with instance compartments linked to the current compartment;  $t_j$  might indicate the type of the current compartment, i.e.,  $t_{l_i}$  – in this case it is ignored; if a link does not exist (the two compartments are not in  $E$ ) then the rule is not applied; if a target,  $t_j$ , refers to a compartment type that has more than one instance connected to  $l_i$ , then one of them will be non-deterministically chosen;
- (b) **structure changing rules**; the following types are considered:
  - (b1) **membrane division rule**:  $[x]_{t_{l_i}} \rightarrow [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}} \{g\}$ ,  
where  $x \in A^+$  and  $y_j$  has the form  $y_j = (a_{j,1}, t_{j,1}) \dots (a_{j,h_j}, t_{j,h_j})$  like in rewriting and communication rules; the compartment  $l_i$  will be replaced by  $p$  compartments; the  $j$ -th compartment, instantiated from the compartment type  $t_{i_j}$  contains the same objects as  $l_i$ , but  $x$ , which will be replaced by  $y_j$ ; all the links of  $l_i$  are inherited by each of the newly created compartments;
  - (b2) **membrane dissolution rule**:  $[x]_{t_{l_i}} \rightarrow \lambda \{g\}$ ;  
the compartment  $l_i$  and its entire contents is be destroyed together with

its links. This contrasts with the classical dissolution semantics where the inner multiset is passed to the parent membrane - in a tree-like membrane structure;

- (b3) **link creation rule**:  $[x]_{t_i}; []_{t_j} \rightarrow [y]_{t_i} - []_{t_j} \{g\};$   
the current compartment is linked to a compartment of type  $t_j$  and  $x$  is transformed into  $y$ ; if more than one instance of the compartment type  $t_j$  exists then one of them will be non-deterministically picked up;  $g$  is a guard that refers to the compartment instantiated from the compartment type  $t_i$ ;
- (b4) **link destruction rule**:  $[x]_{t_i} - []_{t_j} \rightarrow [y]_{t_i}; []_{t_j} \{g\};$   
is the opposite of link creation and means that the compartments are disconnected.

Each compartment can be regarded as an instance of a particular *compartment type* and is therefore subject to its associated rules. One of the main distinctive features of Kernel P systems is the execution strategy which is now statutory to types rather than unitary across the system. Thus, each membrane applies its type specific instruction set, as coordinated by the associated execution strategy.

An execution strategy can be defined as a sequence  $\sigma = \sigma_1 \& \sigma_2 \& \dots \& \sigma_n$ , where  $\sigma_i$  denotes an atomic component of the form:

- $\epsilon$ , an analogue to the generic *skip* instruction; *epsilon* is generally used to denote an *empty* execution strategy;
- $r$ , a rule from the set  $R_t$  (the set of rules associated with type  $t$ ). If  $r$  is applicable, then it is executed, advancing towards the next rule in the succession; otherwise, execution halts, pruning the remainder of the sequence;
- $(r_1, \dots, r_n)$ , with  $r_i \in R_t, 1 \leq i \leq n$  symbolizes a non-deterministic choice within a set of rules. One and only one applicable rule will be executed if such a rule exists, otherwise the atom is simply skipped. In other words the non-deterministic choice block is always applicable;
- $(r_1, \dots, r_n)^*$ , with  $r_i \in R_t, 1 \leq i \leq n$  indicates the arbitrary execution of a set of rules in  $R_t$ . The group can execute zero or more times, arbitrarily but also depending on the applicability of the constituent rules;
- $(r_1, \dots, r_n)^\top$ ,  $r_i \in R_t, 1 \leq i \leq n$  represents the maximally parallel execution of a set of rules. If no rules are applicable, then execution proceeds to the subsequent atom in the chain.

The execution strategy itself is a notable asset in defining more complex behaviour at the compartment level. For instance, priorities can be easily expressed as sequences of maximally parallel execution blocks:  $(r_1)^\top \& (r_2)^\top \& \dots \& (r_3)^\top$  or non-deterministic choice groups if single execution is required. Together with composite guards, they provide an unprecedented modelling fluency and plasticity for membrane systems. Whether such macro-like concepts and structures are preferred over traditional modelling with simple but numerous compartments in complex arrangements is a debatable aspect.

### 3 kP system models and the Spin model checker

Formal verification of P systems has become an increasingly investigated subject, owing to a series of multilateral developments which have broadened its application scope and solidified some domain specific methodologies. Although there have been several attempts that successfully demonstrated model checking techniques on P systems ([16], [17], [14]), the analysis is always bound to an array of constraints, such as specific P system variants with a limited feature set and a very basic set of properties. Nevertheless, there are notable advancements which have paved the path towards a more comprehensive, integrated and automated approach we endeavour to present in this paper.

The task of P system model checking is perhaps a most inviting and compelling one due to the many onerous challenges it poses. On the one hand we are confronted with the inherent shortcomings of the method itself, which have a decisive impact on the tractability of some models and, in the best case, the efficiency or precision of the result is severely undermined. Speaking generally, but not inaccurately, model checking entails an exhaustive, *strategic* exploration of a model's state space to assert the validity of a logically defined property. Hence, the state space is of primary concern and we can immediately acknowledge 1. the requirement for models to have a finite state space and 2. the proportionality between the state space *size* and the stipulated computational resources, which ultimately determines the feasibility of the verification process.

On the other hand, the complex behaviour of certain computational models translates to elaborate formal specifications, with intricate semantics and more often than not, a vast set of states. However, it is the tireless state explosion problem that diminishes the applicability of model checking to concurrent systems, a rather ironical fact, since such systems are now the primary target for exhaustive verification.

We shall not delve any further into general aspects since our focus is not the vivisection of a methodology, but rather the introduction of a robust, integrated and automated approach that constellates around kernel P systems and overtly addresses the predominant challenges of model checking emphasised so far.

The three most conspicuous features that typify membrane systems are 1. a structured, distributed computational environment; 2. multisets of objects as atomic terms in rewriting rules and 3. an *execution strategy* according to which the rules are applied. We recall that kP systems explicitly associate an instruction set to an array of compartments employing the *type - instance* paradigm. As it turns out, this distinction is highly relevant in mapping a formal state transition system, where a system state is conveyed compositionally, as the union of individual states attributed to instances (in our case), or disjoint volatile components in more generic terms. Thus, a kP system state  $S$  is an aggregate of  $S_C$ , the set of compartment states and  $\mu$  which denotes the membrane structure as a set of interconnections between compartments. A compartment state is identified by its associated multiset configuration at a particular computational step, together with the membrane type the compartment it subject to. The following set like expression exemplifies a kernel P system state for three compartments  $c_1, c_2$  and

$c_3$ , of types  $t_1, t_2, t_2$ , having configurations  $2a\ b, a\ 2c$  and *empty* respectively. The second fragment is a set of pairs which symbolize links between compartments:  $c_1$  is connected to both  $c_2$  and  $c_3$ , who do not share a link in-between.

$$(\{(c_1, t_1, \{2a, b\}), (c_2, t_2, \{a, 2c\}), (c_3, t_2, \{\})\}, \{(c_1, c_2), (c_1, c_3)\})$$

Since kP systems feature a dynamic structure by preserving structure changing rules such as membrane division, dissolution and link creation/destruction, a state defined in this expansive context is consequently variable in size. This is not unnatural for a computational model, however it does become an issue when conflicting with the requirement of a fixed sized pre-allocated data model imposed by most model checker tools. The instinctive solution is to bound the expansion of these collections to a certain maximum based on the algorithmic necessities. For instance, an initial analysis of the problem we are modelling can provide relevant details about the number of steps required for a successful execution, the number of divisions that may occur and the maximum number of links generated.

One of the most fruitful advantages of model checking is the fact it can be completely automated. The principal insight is that both the system's state space (commonly referred to as *global reachability graph*) and the correctness claim specified as a temporal logic formula can be converted to non-deterministic finite automata. The product of the two automata is another NDFA whose accepted language is either empty in which case the correctness claim is not satisfied, or non-empty if the system exhibits precisely the behaviour specified by the temporal logic statement. There are numerous implementations of this stratagem boasting various supplementary features, a survey of which is beyond the scope of this study. The model checker extensively adopted in formal verification research on membrane systems is Spin. Developed by Gerard J. Holzmann in the 1990s, Spin is now a leading verification tool used by professional software engineers and has an established authority amidst model checkers. Among plentiful qualities, Spin is particularly suited for modelling concurrent and distributed systems by means of interleaving atomic instructions. For a more comprehensive description of the tool, we refer the reader to [11].

A model checker requires an unambiguous representation of its input model, together with a set of correctness claims expressed as temporal logic formulae. Spin features a high level modelling language, called Promela, which specializes in concise descriptions of concurrent processes and inter-process communication supporting both rendezvous and buffered message passing. Another practical and convenient aspect of the language is the use of discrete primitive data types as in the C programming language. Additionally, custom data types and single dimensional arrays are also supported, although in restricted contexts only.

The kernel P systems specification is an embodiment of elementary components shared by most variants, complemented by innovative new features, promoting a versatile modelling framework without transgressing the *membrane computing* paradigm. Characterised by a rich set of primitives, kP systems offer many



high level functional contexts and building blocks such as the exhaustive and arbitrary execution of a set of rules, complex guards and the popular concept of membrane division - powerful modelling instruments from a user centric perspective. An attempt, however, to equate such a complex synthesis of related abstractions to a mainstream specification is a daunting and challenging task. It is perhaps evident that users should be entirely relieved of this responsibility, and all model transformations should be handled automatically. It is precisely this goal which motivates the development of **kpWorkbench**, a basic framework which integrates a set of translation tools that bridge several target specifications we employ for kernel P system models. The pivotal representation medium is, however, the newly introduced kP-Lingua, a language designed to express a kP system coherently and intuitively. kP-Lingua is described in detail in [7], which includes an EBNF grammar of its syntax. We exemplify kP-Lingua in our dedicated case study, presented in section 5 of this paper.

One of the fundamental objectives in devising a conversion strategy is to establish a correspondence with respect to data and functional modules between the two specifications. In some cases, a direct mapping of entities can be identified:

- A **multiset of objects** is encoded as an integer array, where an index denotes the object and the value at that index represents the multiplicity of the object;
- A **compartment type** is translated into a data type definition, a structure consisting of native elements, the multiset of objects and links to other compartments, as well as auxiliary members such as a temporary storage variable, necessary in order to simulate the inherent parallelism of P systems.
- A **compartment** is an instance of a data type definition and a set of compartments is organised into an array of the respective type;
- A **set of rules** is organised according to an **execution strategy** is mapped by a Proctype definition - a Promela process;
- A **guard** is expressed as a composite conditional statement which is evaluated inside an *if* statement;
- A **rule** is generally converted into a pair of instructions which manage subtraction and addition on compartment multisets, but can also process structural elements such as compartments and links;
- **Exhaustive and arbitrary execution** are resolved with using the *do* block;
- **Single non-deterministic execution** is reflected by an *if* statement with multiple branches; we note that Promela evaluates *if* statements differently than most modern programming languages: if more than one branch evaluates to true, then one is non-deterministically chosen.

It is not, however, the simplicity and limpidity of these projections that prevail, especially when dealing with a computational model so often described as unconventional. Rather, concepts such as maximal parallelism and membrane

division challenge the mainstream modelling approach of sequential processes and settle on contrived syntheses of clauses. These artificial substitutes operate as auxiliary functions and therefore require abstraction from the global state space generated by a model checker tool. Spin supports the hiding of mediator instruction sets by enveloping code into *atomic* or *d\_step* blocks. Although this is a very effective optimisation, we are still faced with the problem of instruction interleaving, the de facto procedure which reconciles parallel and sequential computation. It is not this forced simulation of parallelism that obstructs a natural course for P system verification with Spin, but rather the inevitable inclusion of states generated by interleaved atomic instructions or ensembles of instructions.

In our approach we overcome this obstacle with a hybrid solution, involving both the model in question and the postulated properties. Firstly, we collapse individual instructions (to atomic blocks) to the highest degree permitted by Spin, minimizing the so-called *intermediate state space* which is irrelevant to a P system computation; and secondly, we appoint the states relevant to our model explicitly, using a global flag (i.e. a boolean variable), raised when all processes have completed a computational step. Hence, we make a clear distinction between states that are pertinent to the formal investigation and the ones which should be discarded. This contrast is in turn reflected by the temporal logic formulae, which require adjustment to an orchestrated context where only a narrow subset of the global state space is pursued. The technique is demonstrated in our case study of section 5.

While the approach is a practical success, its efficacy is still a questionable matter. Although a substantial set of states is virtually *neglected* when asserting a correctness claim, the complete state space is nevertheless generated (i.e. including the superfluous states) and each state examined: if the state is flagged as a genuine P system state, then it is queried further, otherwise it is *skipped*. In terms of memory usage, the implications are significant and certainly not to be underestimated, particularly when the model exhibits massively parallel and non-deterministic behaviour.

We conclude the section with an informal synopsis of the *kP system - Promela* translation strategy and the rationale behind some of its noteworthy particularities:

- While each compartment type is represented by a Promela process definition, a *Scheduler* process is employed to launch and coordinate the asynchronous execution of procedures per compartment. The following pseudo-code illustrates the managerial role played by our scheduler:

```
process Scheduler {
  while system is not halted {
    for each type T_i {
      for each compartment C_j of type T_i {
        appoint process P(T_i) to compartment C_j;
      }
    }
  }
}
```

```

    start all appointed processes;
    wait until all appointed processes finish;
    state = step_complete;
    print configuration;
    state = running;
  }
}
```

- Each compartment consists of two multisets of objects, one which rules operate on and *consume* objects from; and the second which temporarily stores the produced or communicated objects. Before the end of each computational step, the content of the auxiliary multiset is committed to the primary multiset, which also denotes the compartment's configuration. This interplay is required to simulate a parallel execution of the system.

## 4 Queries on kernel P systems

A much debated aspect of model checking based formal verification is specifying and formulating a set of properties whose correctness is to be asserted. Since model checking is essentially an exhaustive state space search, there is a persistent and irreconcilable concern over the limitations of this method when investigating the behaviour of concurrent models, generative of an astronomical state spaces. More precisely, the complexity of the model itself has a great subversive impact on the property gamut which can be employed such that the procedure remains feasible given reasonable computational resources.

It is not just the inherent limitations of this technique which must be taken into consideration, but also the effort and tenacity required to formally express specific queries concisely and faithfully into prescribed logical frameworks. Amir Pnueli's seminal work on temporal logic [19] was a major advance in this direction, enabling the elegant representation of time dependent properties in deductive systems. Essential adverbial indicators such as *never* and *eventually* have a diametric correspondent in temporal logic, as operators which relate system states in terms of reachability, persistence and precedence, supporting more powerful queries in addition to simple state equivalence assertions and basic invariance. Exploiting the potential of these logics, as evident as it may seem, can still be problematic and laborious under certain circumstances.

Firstly, devising a temporal logic formula for a required property is a cumbersome and error-prone process even for the experienced. It is often the case that the yielded expressions, although logically valid, are counter-intuitive and abstruse, having little to tell about the significance of the property itself. As with any abstraction that is based on pure logical inference, it is devoid of meaning outside the logical context. To clearly emphasize our affirmations, consider the following example:

$$\begin{aligned} G \ (vm\_functional = true \ \wedge \ vm\_coin > 0 \rightarrow \\ F \ (vm\_dispensed\_drink > 0 \ \wedge \ F \ (vm\_coin = 0)) \vee \\ F \ (vm\_functional = false)) \end{aligned}$$

is a faithful LTL (linear time temporal logic) representation of a property which can be phrased as “a vending machine, if functional, will always dispense a drink after having accepted coins and will either become dysfunctional or its coin buffer will be depleted.” Although we have used intuitive variable names, it is not immediately apparent what this expression stands for, requiring a thorough understanding of the LTL specification together with effort and insight to accurately decipher its meaning.

The second notable issue we wish to evince is the correctness of the formula itself which can often be questionable even if the property is of moderate complexity and is syntactically accepted by a model checker tool. How can one prove that a temporal logic expression is indeed a valid representation of a property we wish to verify? Is this a genuine concern we should address, or is it acceptable to assume the faithfulness of temporal logic expressions to specific queries, as formulated by expert and non-expert users?

In response to these controversies, we propose a strategy that facilitates a *guided construction* of relevant LTL properties and automates the translation to their formal equivalent. It is the Natural Language Query (NLQ) builder that was developed to support this methodology. The tool features a rich set of *natural language patterns*, presented to users as sequences of GUI (graphical user interface) form elements: labels, text boxes and drop-down lists. Once the required values have been selected or directly specified and the template populated, NLQ automatically converts the natural language statement to its temporal logic correspondent. The translation from an informal to a formal representation is based on an interpreted grammar which accompanies each natural language pattern.

In table 1, we illustrate a selection of patterns whose instantiation generates properties suitable for kP system models and their formal verification. Table 2 depicts the EBNF based grammar according to which, *state formulae* are derived, with reference to kernel P system components.

In order to verify kP systems modeled in kP-Lingua using Spin model checker, properties specified in LTL should be reformulated in Spin language for the corresponding Promela model. In Table 3, we give LTL formulae of the patterns shown in Table 1, and their corresponding translations in Spin language for the Promela specification. Each LTL formula described for P systems in general (and kP systems in our case) should be translated to Spin using a special predicate, **pInS**, showing that the current Spin state represents a P system configuration (the predicate is true when a computation step is completed) or represents an intermediate state (it is false if intermediary steps are executed) [14, 17].

The idea of capturing recurring properties into categories of patterns was initiated by Dwyer et al. in their seminal paper of 1999 [5]. This study surveyed more than five hundred temporal properties and established a handful of pattern classes. In [8], this mapping was extended to include additional time related

|              |  |
|--------------|--|
| Pattern      | ::= Occurrence   Order   |
| Occurrence   | ::= Next   Existence   Absence   Universality  <br>Recurrence   Steady-State                             |
| Order        | ::= Until   Precedence   Response  |
| Next         | ::= stateFormula <i>'will hold in the next state'</i> .  |
| Existence    | ::= stateFormula <i>'will eventually hold'</i> .   |
| Absence      | ::= stateFormula <i>'never holds'</i> .  |
| Universality | ::= stateFormula <i>'always holds'</i> .   |
| Recurrence   | ::= stateFormula <i>'holds infinitely often'</i> .   |
| Steady-State | ::= stateFormula <i>'will hold in the long run (steady state)'</i> .                                     |
| Until        | ::= stateFormula <i>'will eventually hold, until then'</i> stateFormula<br><i>'holds continuously'</i> . |
| Response     | ::= stateFormula <i>'is always followed by'</i> stateFormula.  |
| Precedence   | ::= stateFormula <i>'is always preceded by'</i> stateFormula.  |

**Table 1.** Grammar for query patterns.

patterns and their associated observer automata. This was further supplemented with real-time specification patterns in [15].

A unified pattern system was introduced in [2], adding new real-time property classes. Probabilistic properties were similarly catalogued based on a survey of 200 properties [9], and provisioned with a corresponding structured grammar.

An analogous undertaking can also be observed in [18], where an array of query templates which target biological models was proposed.

Although the NLQ builder is based on an extensive set of patterns investigated in above mentioned literature, the templates relevant to our formal examination of kP system models represent a small subset of this collection; particularly we only employ patterns which generate temporal properties.

## 5 Case study: the Subset Sum problem

In this section we demonstrate the proposed methodology with a case study, the subject of which is the well known Subset Sum problem.

The Subset Sum problem is stated as follows:

*Given a finite set  $A = \{a_1, \dots, a_n\}$ , of  $n$  elements, where each element  $a_i$  has an associated weight,  $w_i$ , and a constant  $k \in \mathbb{N}$ , it is requested to determine whether or not there exists a subset  $B \subseteq A$  such that  $w(B) = k$ , where  $w(B) = \sum_{a_i \in B} w_i$ .*

The Subset Sum problem is representative for the NP complete class because it portrays the underlying necessity to consider *all combinations* of distinct elements of a finite set, in order to produce a result. Consequently, such a problem requires exponential computational resources (assuming  $P \neq NP$ ), either in the

|                      |   |
|----------------------|---|
| stateFormula         | ::= statePredicate   statePredicate <i>'does not hold'</i>  <br>stateFormula <i>'and'</i> stateFormula  <br>stateFormula <i>'or'</i> stateFormula   |
| statePredicate       | ::= numericExpression relationalOperator numericExpression  |
| numericExpression    | ::= objectCount   localObjectCount   compartmentCount  <br>linkCount   linkToCount   numericLiteral   |
| linkCount            | ::= <i>'the number of links from'</i> compartmentQuery<br><i>'to'</i> compartmentQuery  |
| linkToCount          | ::= <i>'the number of links to'</i> compartmentQuery  |
| compartmentQuery     | ::= <i>'all compartments'</i>   <i>'compartments'</i> compartmentCondition  |
| compartmentCondition | ::= <i>'of type'</i> typeLabel   <i>'of type other than'</i> typeLabel  <br><i>'linked to'</i> compartmentQuery  <br><i>'not linked to'</i> compartmentQuery  <br>localObjectCount relationalOperator numericExpression  <br>linkToCount relationalOperator numericExpression |
| localObjectCount     | ::= <i>'the number of objects'</i> localObjectCondition   |
| objectCount          | ::= <i>'the number of objects'</i> objectCondition  |
| localObjectCondition | ::= <i>'with label'</i> objectLabel  <br><i>'with label different than'</i> objectLabel  <br>localObjectCondition <i>'and'</i> localObjectCondition  <br>localObjectCondition <i>'or'</i> localObjectCondition  |
| objectCondition      | ::= localObjectCondition  <br><i>'in'</i> compartmentQuery  <br><i>'not in'</i> compartmentQuery  <br>objectCondition <i>'and'</i> objectCondition  <br>objectCondition <i>'or'</i> objectCondition   |
| relationalOperator   | ::= <i>'is equal to'</i>   <i>'is not equal to'</i>   <i>'is greater than'</i>  <br><i>'is less than'</i>   <i>'is greater than or equal to'</i>  <br><i>'is less than or equal to'</i>   |
| numericLiteral       | ::= ? {0-9} ?   |

**Table 2.** EBNF based grammar for state formulae.

| Pattern      | Informal Formula   | LTl formula                           | Spin formula  |
|--------------|--|---------------------------------------|---|
| Next         | $p$ will hold in the next state                                | $X p$                                 | $X(!pInS \cup (p \ \&\& \ pInS))$                             |
| Existence    | $p$ will eventually hold                                       | $F p$                                 | $<>(p \ \&\& \ pInS)$   |
| Absence      | $p$ never holds  | $\neg(F p)$                           | $!(<>(p \ \&\& \ pInS))$                                      |
| Universality | $p$ always holds   | $G p$                                 | $\Box (p \    \ !pInS)$                                       |
| Recurrence   | $p$ holds infinitely often                                     | $G F p$                               | $\Box (<>(p \ \&\& \ pInS) \    \ !pInS)$                     |
| Steady-State | $p$ will hold in the steady state                              | $F G p$                               | $<>(\Box (p \    \ !pInS) \ \&\& \ pInS)$                     |
| Until        | $p$ will eventually hold,<br>until then $q$ holds continuously | $p \cup q$                            | $(p \    \ !pInS) \cup (q \ \&\& \ pInS)$                     |
| Response     | $p$ is always followed by $q$                                  | $G (p \rightarrow F q)$               | $\Box ((p \rightarrow <> (q \ \&\& \ pInS)) \    \ !pInS)$    |
| Precedence   | $p$ is always preceded by $q$                                  | $\neg(\neg p \cup (\neg p \wedge q))$ | $!(\Box (p \    \ !pInS) \cup (!p \ \&\& \ q \ \&\& \ pInS))$ |

**Table 3.** LTL formulae and translated Spin specifications of the property patterns

temporal (number of computational steps) or spatial (memory) domain, or both. The Subset Sum problem explicitly denominates combinations of integers as subsets of the initial set  $A$ , or more accurately, the set of weights respective to  $A$ . It is therefore transparent that the number of all combinations which can be generated and evaluated is the cardinality of the power set of  $A$ , that is  $2^n$ . Since our elements are in fact integers, optimisations have been considered, leveraging the intrinsic order relation between numbers, coupled with efficient sorting algorithms to avoid generating all possible subsets [12]. This did not, however, manage to reduce the complexity of the problem to a non-exponential order.

P system variants endowed with *membrane division* proved to be ideal computational frameworks for solving NP complete problems efficiently. The insightful strategy, often referred to as *trading space for time*, can be envisaged as the linear generation of an exponential computational space (compartments) together with the linear distribution (replication) of constituent data (multiset of objects). The topic is very popular in the community and was subject to extensive investigation; while the underlying principle is pertinent to our study, we shall illustrate it more sharply as applied, using a kernel P system model to solve the Subset Sum problem:

Consider the kP system

$$k\Pi = (\{a, x, step, yes, no, halt, r_1, \dots, r_n\}, \mu, (Main, \{a\}), (Output, \{step\}))$$

with  $\mu$  represented by a link between the two instances of type *Main* and *Output* respectively.

The rules for compartments of type *Main* are:

- $R_i$ :  $a \longrightarrow [a, r_i][w_i x, a, r_i]\{\neg r_i\}, 1 \leq i \leq n$
- $R_{n+1}$ :  $a \longrightarrow (yes, halt)_{Output} \{= kx\}$
- $R_{n+2}$ :  $a \longrightarrow \lambda \{> kx\}$

where

- $n$  is the number of elements in set  $A$ , that is the cardinal of  $A$ ;
- $r_i$  with  $1 \leq i \leq n$  is an object which flags the execution of a membrane division rule, prohibiting multiple applications of the same addition;

- $w_i$  is the weight of the  $i$ th element in the set  $A$ , with  $1 \leq i \leq n$ ;
- $k$  is the constant we refer to, when assessing the sum of the values in a subset; if  $\sum w_i = k$ , then a solution has been found;

The execution strategy  $\sigma(Main)$  unfolds as follows:

$$\sigma(Main) = (R_{n+1}, R_{n+2}) \& (R_{1..n})$$

Thus, each step a compartment of type *Main* performs two preliminary evaluations: if the number of  $x$  objects is precisely  $k$ , then a *yes* and a *halt* object are sent to the output membrane. We recall the specialised *halt* object as a universal, model independent and convenient means of halting a computation for kernel P systems: when such an object is encountered in any of the system's compartments, the execution stops at the completion of the computational step. This is generally preferred to specifying halting conditions which relate to configurations or system states particular to the modelled problem.

If the multiplicity of  $x$  is greater than  $k$ , a condition assessed with the guard  $> kx$ , the compartment is dissolved, pruning a fruitless search path. Otherwise, a division rule is selected non-deterministically, splitting the compartment in two and adding  $w_i x$ s to the current multiplicity of  $x$  in one of the newly created regions, while preserving the weight of  $x$  in the other. Both compartments also receive a  $r_i$  object which marks the execution of the  $i$ th rule. This will be prevented from executing a second time by the guard  $\neg r_i$ . The object  $a$  is auxiliary and recurs in every compartment of type *Main*.

There is only one compartment of type *Output* which persists throughout the execution, playing the role of an output membrane, as its name plainly indicates: either it receives a *yes* object if a solution is found, or it generates a *no* object if the computation does not halt after  $n + 1$  steps. The two rules which correlate with this behaviour are:

- $R_1 : \text{step} \longrightarrow 2\text{step}$
- $R_2 : (n + 2)\text{step} \longrightarrow \text{no}, \text{halt}$

The rules are executed sequentially:

$$\sigma(Output) = (R_1 \& R_2)$$

*Remark 1.* The illustrated algorithm is a faithful **linear time** solution to the Subset Sum problem: it computes an answer to the stipulated enquiry in **maximum**  $n + 2$  **steps**, where  $n$  is the cardinality of the set  $A$  of elements.

*Remark 2.* The algorithm will generate the sums of all subsets of  $A$  in linear time using membrane division; the process is interrupted when a solution is found and computation halts at this stage. A notable difference to the skP (simple kernel P) system based solution presented in [13], is the use of non-deterministic choice in the selection of division rules. This rather unconventional approach facilitates the generation of subset sums that is irrespective of the order of elements in  $A$ . Evidently, the artifice owes its merit to the commutativity of integer addition.



*Remark 3.* The kP system model requires a total of:  $n + 6$  distinct objects,  $n + 4$  rules of which  $n + 1$  employ basic guards and a maximum of  $2^n + 1$  compartments.

*Remark 4.* Although we have extensively referred to integer weights (of the elements in  $A$ ) throughout this section, it is important to note that we can not directly represent negative numbers as object multiplicities alone (some encoding can be devised for this purpose). Since the only mathematical operation required is addition, which is a monotonically increasing function, a simple translation to the positive domain can be mapped on the set of weights  $w(A)$ , which in turn makes this issue irrelevant.

We next demonstrate the implementation of our kP system model in kP-Lingua, highlighting some of the most prominent features of its syntax. The illustrated model maps an instance of the Subset Sum problem with  $n = 7$  elements:  $w(A) = \{3, 25, 8, 23, 5, 14, 30\}$  and  $k = 55$ .

```

type Main {
  choice {
    = 55x: a -> {yes, halt} (Output) .
    > 55x: a -> # .
  }
  choice {
    !r1: a -> [a, r1] [3x, a, r1] .
    !r2: a -> [a, r2] [25x, a, r2] .
    !r3: a -> [a, r3] [8x, a, r3] .
    !r4: a -> [a, r4] [23x, a, r4] .
    !r5: a -> [a, r5] [5x, a, r5] .
    !r6: a -> [a, r6] [14x, a, r6] .
    !r7: a -> [a, r7] [30x, a, r7] .
  }
}

type Output {
  step -> 2 step .
  9 step -> no, halt .
}

{a} (Main) - {step} (Output) .

```

The code comprises of two type definitions, *Main* and *Output*, together with the instantiation of two, linked, compartments of the respective types. The first two rules are guarded by  $\{= 55x\}$  and  $\{> 55x\}$  respectively, and organized in a *choice* block since they are mutually exclusive and each may execute once and only once. Indeed, enclosing these rules in a maximally parallel grouping would result in equivalent behaviour. A guard always relates to the multiset contained in the compartment it evaluates in and terminates with a colon; the  $->$  symbol denotes the transition of a non-empty multiset on the left hand side

to a rewrite-communication outcome (objects *yes*, *halt* into the compartment of type Output), or a *single* structure changing element (# which symbolises membrane dissolution). Next, the choice block is applied as a non-deterministic selection of *one* of the rules it envelopes: there are seven division rules, which resemble the addition of a value from  $w(A)$ . Each rule is prefixed by a guard  $!r_i$ , in order to prevent its subsequent application which would equate to multiple additions of the same number.

Type *Output* lists two rewriting rules which execute successively and non-repetitively. The first rule *increments* the number of *step* objects in the compartment, updating the step count as the computation unfolds. The second rule will only execute if we have reached the 9th step and no *halt* object was received from any of the *Main* compartments, effectively pronouncing a negative answer to the problem.

The kP-Lingua implementation is a compact and intuitive representation of the formally described model presented earlier. The specification is next translated into Spin's modelling language, Promela, a fully automated process accomplished by a *kP-Lingua parser* and *kP system - Promela* model converter, constituent tools of *kP Workbench*. We document this stage of our approach with several fragments of the rather cryptic Promela encoding, as generated by our converter.

```
#define AO_SIZE 9
#define A1_SIZE 4

typedef C0 {
    int x[AO_SIZE] = 0;
    int xt[AO_SIZE] = 0;
    int c1Links[1];
    int c1LCount = 0;
    int c1LSize = 0;
    bit isComputing = 0;
    bit isDissolved = 0;
    bit isDivided = 0;
}

typedef C1 {
    int x[A1_SIZE] = 0;
    int xt[A1_SIZE] = 0;
    int c0Links[100];
    int c0LCount = 0;
    int c0LSize = 0;
    bit isComputing = 0;
}

int step = 0;
bit halt = 0;
```

```

C0 m0[20];
int m0Count = 0;
int m0Size = 0;
C1 m1[1];
int m1Count = 0;
int m1Size = 0;

int m0DissolvedCount = 0;
int stepsExp = 1;

```

In table 4 we elucidate the constituent elements of the above printed data structures and variable declarations.

|                         |   |
|-------------------------|---|
| <b>A0_SIZE, A1_SIZE</b> | The size of the alphabet for each type of compartment;  |
| <b>C0, C1</b>           | The compartment types <i>Main</i> and <i>Output</i> respectively;   |
| <b>x, xt</b>            | The arrays which store multiplicities of objects encoded as indices;                                      |
| <b>c1Links[1]</b>       | The array of links to compartments of type C1;  |
| <b>isComputing</b>      | A flag indicating whether a process is running on this instance or not;                                   |
| <b>isDissolved</b>      | A flag indicating whether the compartment is dissolved or not;  |
| <b>isDivided</b>        | Indicates if the compartment was divided<br>(and henceforth considered non-existent);                     |
| <b>m0, m1</b>           | The arrays which store compartments of type<br>C0 ( <i>Main</i> ) and C1 ( <i>Output</i> ), respectively; |
| <b>m0[0].x[2]</b>       | The object with index 2 in the 0th compartment of type C0;  |
| <b>m1[0].x[0]</b>       | Multiplicity of object <i>step</i> in compartment 0 of type <i>Output</i> ;                               |
| <b>m1[0].x[1]</b>       | Multiplicity of object <i>yes</i> in the output compartment;  |
| <b>m1[0].x[2]</b>       | Multiplicity of object <i>no</i> in the output compartment;   |
| <b>m1[0].x[3]</b>       | Multiplicity of object <i>halt</i> in the output compartment;   |
| <b>m0DissolvedCount</b> | The number of dissolved compartments of type <i>Main</i> ;  |
| <b>stepsExp</b>         | A number updated each step with the value of $2^{step}$ .   |

**Table 4.** Interpretation of variable expressions generated in Promela

The second key requirement for the model checking methodology we exemplify in this section is the provision of LTL formulae the validity of which is to be asserted against the model. As methodically described in the previous section, a set of properties is generated by instantiating various natural language patterns. These are appointed as templates to be completed by the user with model variables or numeric constants, interactively, through a graphical user interface. Several screenshots which illustrate the Natural Language Query (NLQ) builder, integrated into kpWorkbench are supplied in the Appendix.

Table 5 lists an array of ten properties we have compiled and derived from natural language patterns for the Subset Sum example. These properties have been successfully verified with Spin on a Core i7 980X based machine, with 24GB RAM and running Windows 8 Professional Edition.

Devising a set of properties assisted by the NLQ tool becomes an intuitive, effortless and streamlined task, however, there may be cases when a generated

natural language statement does not reflect the meaning of the property in its entirety, although it is logically equivalent. This may lead to shallow interpretations if the formal representation is not consulted and ultimately to oversights of relevant implications of the property. For example, in Table 5, the property *a ‘yes’ result is eventually observed within no more than three step* is as a fabricated form of *there exists a non-deterministic execution strategy that yields an affirmative result to the problem in no more than three steps*. The second expression is significantly more elevate and meaningful in comparison with its generated counterpart which clearly describes the underlying LTL formulae, but requires a deeper understanding of the model for an accurate interpretation.

| Property | Pattern      | Natural Language Statement and Spin formula  |
|----------|--------------|--|
| 1        | Until        | The computation will eventually halt.<br>$\text{halt} == 0 \text{ U } \text{halt} > 0$<br>$(\text{m1}[0].\text{x}[3] == 0 \text{    } !\text{pInS}) \text{ U } (\text{m1}[0].\text{x}[3] > 0 \text{ \&\& } \text{pInS})$   |
| 2        | Until        | The computation will halt within $n + 2$ steps.<br>$(\text{halt} == 0 \text{ \&\& } \text{steps} < n + 2) \text{ U } (\text{halt} > 0 \text{ \&\& } \text{steps} \leq n + 2)$<br>$(\text{m1}[0].\text{x}[3] == 0 \text{ \&\& } \text{m1}[0].\text{x}[0] < n+2 \text{    } !\text{pInS}) \text{ U } (\text{m1}[0].\text{x}[3] > 0 \text{ \&\& } \text{m1}[0].\text{x}[0] \leq n+2 \text{ \&\& } \text{pInS})$ |
| 3        | Until        | The computation will eventually halt with either a ‘yes’ or ‘no’ result.<br>$\text{halt} == 0 \text{ U } (\text{halt} > 0 \text{ \&\& } (\text{yes} > 0 \text{    } \text{no} > 0))$<br>$(\text{m1}[0].\text{x}[3] == 0 \text{    } !\text{pInS}) \text{ U } (\text{m1}[0].\text{x}[3] > 0 \text{ \&\& } (\text{m1}[0].\text{x}[1] > 0 \text{    } \text{m1}[0].\text{x}[2] > 0) \text{ \&\& } \text{pInS})$ |
| 4        | Until        | At least one membrane division will eventually take place (before a result is obtained).<br>$(\text{yes} == 0 \text{ \&\& } \text{no} == 0) \text{ U } \text{m0Count} > 1$<br>$(\text{m1}[0].\text{x}[1] == 0 \text{ \&\& } \text{m1}[0].\text{x}[2] == 0) \text{    } !\text{pInS} \text{ U } \text{m0Count} > 1 \text{ \&\& } \text{pInS}$   |
| 5        | Existence    | A ‘yes’ result is eventually observed within no more than three steps.<br>$\text{F } (\text{yes} > 0 \text{ \&\& } \text{steps} \leq 3)$<br>$\langle \rangle (\text{m1}[0].\text{x}[1] > 0 \text{ \&\& } \text{m1}[0].\text{x}[0] \leq 3 \text{ \&\& } \text{pInS})$   |
| 6        | Existence    | A ‘yes’ result is eventually observed within more than three steps.<br>$\text{F } (\text{yes} > 0 \text{ \&\& } \text{steps} > 3)$<br>$\langle \rangle (\text{m1}[0].\text{x}[1] > 0 \text{ \&\& } \text{m1}[0].\text{x}[0] > 3 \text{ \&\& } \text{pInS})$  |
| 7        | Existence    | A result (‘yes’ or ‘no’) is eventually obtained without any membrane dissolutions.<br>$\text{F } (\text{yes} > 0 \text{    } \text{no} > 0) \text{ \&\& } \text{m0DissolvedCount} == 0$<br>$\langle \rangle ((\text{m1}[0].\text{x}[1] > 0 \text{    } \text{m1}[0].\text{x}[2] > 0) \text{ \&\& } \text{m0DissolvedCount} == 0 \text{ \&\& } \text{pInS})$  |
| 8        | Existence    | A ‘yes’ result is eventually obtained with membrane dissolution occurring.<br>$\text{F } \text{yes} > 0 \text{ \&\& } \text{m0DissolvedCount} > 0$<br>$\langle \rangle (\text{m1}[0].\text{x}[1] > 0 \text{ \&\& } \text{m0DissolvedCount} > 0 \text{ \&\& } \text{pInS})$   |
| 9        | Universality | The number of compartments in use is always equal to $2^{\text{stepcount}}$ .<br>$\text{G } \text{m0Count} + 1 == \text{TwoToTheNumberOfSteps}$<br>$[] (\text{m0Count} + 1 == \text{TwoToTheNumberOfSteps} \text{    } !\text{pInS})$  |
| 10       | Absence      | There will never be a negative answer for this example.<br>$!\text{F } \text{no} > 0$<br>$!(\langle \rangle (\text{m1}.\text{x}[2] > 0 \text{ \&\& } \text{pInS}))$  |

**Table 5.** List of properties derived from natural language patterns using NLQ and their generated LTL equivalent

## 6 Conclusions

The approach to kernel P system model checking presented in this paper is a powerful synthesis of concepts and ideas, materialised into an aggregate of software tools and template data sets. The investigation permeates two innovative leaps, namely the kP system computational model in the context of membrane computing and the use of natural language patterns to generate temporal logic properties in the field of model checking. After establishing a model equivalence relation together with a procedural translation from a generic representation to a notation required by Spin, non-specialist users can benefit from the standard features offered by the model checker. The often intricate and abstruse process of constructing temporal logic formulae has also been abstracted to natural language statements and interactive visual representation through graphical user interface (GUI) elements. Another consequential advantage of significance is the correctness guarantee conferred by an automatic model conversion and formula generation.

Our case study illustrated in section 5, demonstrates the feasibility of this approach with its illustrious qualities, but also exposes the potential limitations of the method: on one hand, the notorious state space explosion problem is an inexorable fact that circumscribes the model checking of concurrent and non-deterministic systems; on the other hand, some generated properties, products of composite natural language patterns, are devoid of meaning and can possibly lead to shallow or inaccurate interpretations and even confusion.

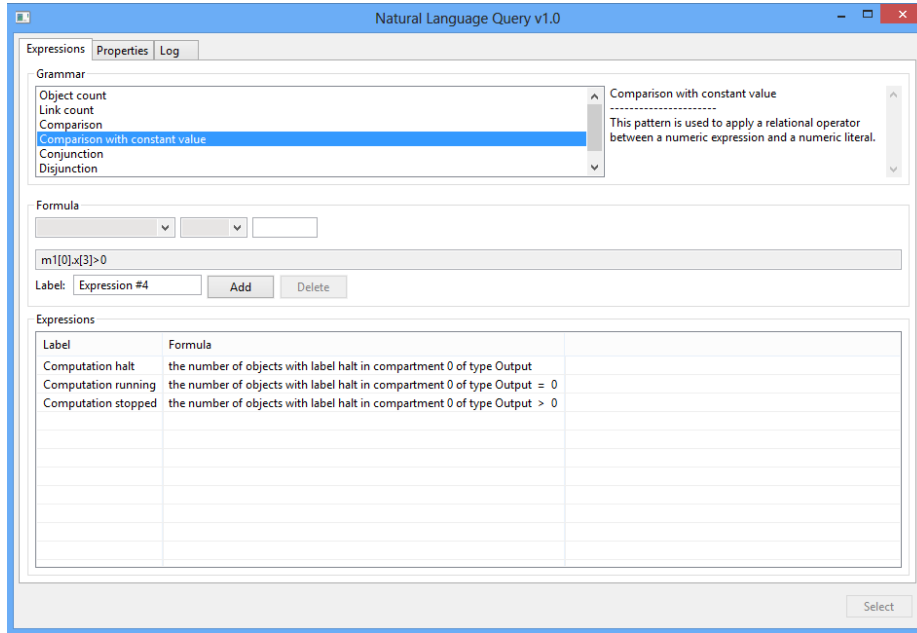
Evidently, a more consistent qualitative evaluation of the methodology, involving several other case studies is required to highlight its potential and limitations more generally. It would be interesting to see the outcome of future investigations in this newly established context.

## References

1. Andrei, O., Ciobanu, G., Lucanu, D.: A rewriting logic framework for operational semantics of membrane systems. *Theoretical Computer Science* 373(3), 163–181 (2007)
2. Bellini, P., Nesi, P., Rogai, D.: Expressing and organizing real-time specification patterns via temporal logics. *J. Syst. Softw.* 82(2), 183–196 (Feb 2009)
3. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: *WMC 2007. LNCS*, vol. 4860, pp. 138–159. Springer (2007)
4. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): *Applications of Membrane Computing*. Natural Computing Series, Springer (2006)
5. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: *Proceedings of the 21st international conference on Software engineering*. pp. 411–420. ICSE '99, ACM (1999)
6. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M., Turcanu, A., Valencia, L., Garcia-Quismondo, M., Mierlă, L.: 3-col problem modelling using simple kernel P systems. *International Journal of Computer Mathematics*, online version (to appear)

7. Gheorghe, M., Ipate, F., Dragomir, C., Mierla, L., Valencia-Cabrera, L., Garcia-Quismondo, M., Perez-Jimenez, M.: Kernel P systems. Eleventh Brainstorming Week on Membrane Computing pp. 97–124 (2013)
8. Gruhn, V., Laue, R.: Patterns for timed property specifications. *Electron. Notes Theor. Comput. Sci.* 153(2), 117–133 (2006)
9. Grunske, L.: Specification patterns for probabilistic quality properties. In: *Proceedings of the 30th international conference on Software engineering*. pp. 31–40. ICSE '08, ACM (2008)
10. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: *Proc. TACAS'06*. LNCS, vol. 3920, pp. 441–444. Springer (2006)
11. Holzmann, G.J.: The model checker SPIN. *IEEE Transactions on Software Engineering* 23(5), 275–295 (1997)
12. Horowitz, E.; Sahni, S.: Computing partitions with applications to the knapsack problem. *Journal of the Association for Computing Machinery* 21, 277–292 (1974)
13. Ipate, F., Lefticaru, R., Mierla, L., Cabrera Valencia, L., Han, H., Zhang, G., Dragomir, C., Perez Jimenez, M., Gheorghe, M.: Kernel P systems: Applications and implementations. In: *Proceedings of The Eighth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2013, Advances in Intelligent Systems and Computing*, vol. 212, pp. 1081–1089. Springer Berlin Heidelberg (2013)
14. Ipate, F., Lefticaru, R., Tudose, C.: Formal verification of P systems using Spin. *International Journal of Foundations of Computer Science* 22(1), 133–142 (2011)
15. Konrad, S., Cheng, B.: Real-time specification patterns. In: *Proceedings of 27th International Conference on Software Engineering*. pp. 372 – 381 (2005)
16. Lefticaru, R., Ipate, F., Valencia-Cabrera, L., Turcanu, A., Tudose, C., Gheorghe, M., Jiménez, M.J.P., Niculescu, I.M., Dragomir, C.: Towards an integrated approach for model simulation, property extraction and verification of P systems. *Tenth Brainstorming Week on Membrane Computing vol. I*, 291–318 (2012)
17. Lefticaru, R., Tudose, C., Ipate, F.: Towards automated verification of P systems using Spin. *International Journal of Natural Computing Research* 2(3), 1–12 (2011)
18. Monteiro, P.T., Ropers, D., Mateescu, R., Freitas, A.T., de Jong, H.: Temporal logic patterns for querying dynamic models of cellular interaction networks. *Bioinformatics* 24(16), i227–i233 (2008)
19. Pnueli, A.: The temporal logic of programs. In: *Proceedings of the 18th Annual IEEE Symposium on Foundations of Computer Science*. pp. 46–57. IEEE Computer Society Press (1977)
20. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)

## Appendix A: Screenshots illustrating our NLQ tool



[illegible]The screenshot shows the "Natural Language Query v1.0" application window. It has three tabs at the top: "Expressions", "Properties", and "Log". The "Expressions" tab is active. Inside this tab, there's a section titled "Select a Category" with a dropdown menu showing "until". Below this is a "Patterns" section containing a list box with one item: "expression 'p' will eventually hold, until then expression 'q' holds continuously". To the right of the list box is a "Description..." label. Further down is a "Formula" section where two dropdown menus are set to "Computation running" and "Computation stopped", respectively, separated by the text "' will eventually hold, until then expression '". Below the dropdowns is a text input field containing the formula "( m1[0].x[3]==0 || !pinS) U ( m1[0].x[3]>0 && pinS)". Underneath the formula is a "Label" field with "Property #1" and two buttons, "Add" and "Delete". At the bottom of the "Expressions" tab is a "Properties" section which contains a table with two columns: "Label" and "Formula". The table is currently empty. A "Select" button is located at the bottom right corner of the application window.



# Purely Catalytic P Systems: Two Catalysts Can Be Sufficient for Computational Completeness

Rudolf Freund

Faculty of Informatics, Vienna University of Technology  
Favoritenstr. 9, 1040 Vienna, Austria  
Email: [rudi@emcc.at](mailto:rudi@emcc.at)

**Abstract.** Whether purely catalytic P systems with only two catalysts can already be computational complete, is still an open problem. Here we establish computational completeness by using specific variants of additional control mechanisms. At each step using only multiset rules from one set of a finite number of sets of rules allows for obtaining computational completeness with two catalysts and only one membrane. If the available sets of rules change periodically with time, computational completeness can be obtained with two catalysts in one membrane, too. Moreover, we show computational completeness for purely catalytic P systems with two mobile catalysts and two membranes.

## 1 Introduction

P systems with catalytic rules were already considered in the originating papers for membrane systems, see [8]. In [3] two catalysts were shown to be sufficient for getting universality/computational completeness (throughout this paper, with these notions we will indicate that all recursively enumerable sets of (vectors of) non-negative integers can be generated). Since then, it has become one of the most challenging open problems in the area of P systems, whether or not one catalyst might already be enough to obtain computational completeness. Similar questions also arise in the context of purely catalytic P systems: purely catalytic P systems (working in the maximally parallel mode) with only one catalyst simply correspond with sequential P systems with only one membrane, hence, to multiset rewriting systems with context-free rules, and therefore can only generate linear sets; purely catalytic P systems working in the maximally parallel mode are computationally complete with (at most) three catalysts in only one membrane as shown in [3]; the question whether two catalysts are sufficient or not is another interesting question still open since the beginning of the membrane systems area.

Using additional control mechanisms as, for example, priorities or promoters/inhibitors, P systems with only one catalyst can be shown to be computationally complete, e.g., see Chapter 4 of [10]. On the other hand, additional features for the catalyst may be taken into account; for example, we may use

bi-stable catalysts (catalysts switching between two different states) or mobile catalysts (catalysts able to cross membranes).

P systems with mobile catalysts were introduced in [5], and their universality with one catalyst was proved with using three membranes and target indications of the forms *here*, *out*, and *in<sub>j</sub>*. In [4], this result was improved by replacing the target indications *in<sub>j</sub>* with the weaker one *in*. Here we will prove that two mobile catalysts “traveling” between two membranes yield computational completeness in the case of purely catalytic P systems.

Recently, several variants of P systems using only one catalyst together with control mechanisms for choosing the rules applicable in a computation step have been considered: for example, in [6] the rules are labeled with elements from an alphabet  $H$  and in each step a maximal multiset of rules having the same label from  $H$  is applied. In [4], a short proof for the universality of these *P systems with label selection* with only one catalyst in a single membrane was given. In this paper, we will show a similar result for purely catalytic P systems with only two catalysts in a single membrane.

Regular control languages were considered already in [6] for the maximally parallel derivation mode, whereas in [1] universality was proved for the sequential mode: there even only non-cooperative rules were needed in one membrane for time-varying P systems to obtain universality (in time-varying systems, the set of available rules varies periodically with time, i.e., the regular control language is of the very specific form  $W = (U_1 \dots U_p)^*$ , allowing to apply rules from a set  $U_i$  in the computation step  $pn + i$ ,  $n \geq 0$ ;  $p$  is called the *period*), but a bounded number of steps without applying any rule had to be allowed. In [4], *time-varying P systems* using the maximally parallel derivation mode in one membrane with only one catalyst were shown to be computationally complete with a period of six and the usual halting when no rule can be applied. We here prove a similar result for purely catalytic P systems with only two catalysts in a single membrane.

## 2 Prerequisites

The set of integers is denoted by  $\mathbb{Z}$ , the set of non-negative integers by  $\mathbb{N}$ . An *alphabet*  $V$  is a finite non-empty set of abstract *symbols*. Given  $V$ , the free monoid generated by  $V$  under the operation of concatenation is denoted by  $V^*$ ; the elements of  $V^*$  are called strings, and the *empty string* is denoted by  $\lambda$ ;  $V^* \setminus \{\lambda\}$  is denoted by  $V^+$ . Let  $\{a_1, \dots, a_n\}$  be an arbitrary alphabet; the number of occurrences of a symbol  $a_i$  in a string  $x$  is denoted by  $|x|_{a_i}$ ; the *Parikh vector* associated with  $x$  with respect to  $a_1, \dots, a_n$  is  $(|x|_{a_1}, \dots, |x|_{a_n})$ . The *Parikh image* of a language  $L$  over  $\{a_1, \dots, a_n\}$  is the set of all Parikh vectors of strings in  $L$ , and we denote it by  $Ps(L)$ . For a family of languages  $FL$ , the family of Parikh images of languages in  $FL$  is denoted by  $PsFL$ ; for families of languages of a one-letter alphabet, the corresponding sets of non-negative integers are denoted by  $NFL$ .

A (finite) multiset over the (finite) alphabet  $V$ ,  $V = \{a_1, \dots, a_n\}$ , is a mapping  $f : V \rightarrow \mathbb{N}$  and represented by  $\langle f(a_1), a_1 \rangle \cdots \langle f(a_n), a_n \rangle$  or by any string  $x$  the Parikh vector of which with respect to  $a_1, \dots, a_n$  is  $(f(a_1), \dots, f(a_n))$ . In the following we will not distinguish between a vector  $(m_1, \dots, m_n)$ , its representation by a multiset  $\langle m_1, a_1 \rangle \cdots \langle m_n, a_n \rangle$  or its representation by a string  $x$  having the Parikh vector  $(|x|_{a_1}, \dots, |x|_{a_n}) = (m_1, \dots, m_n)$ . Fixing the sequence of symbols  $a_1, \dots, a_n$  in the alphabet  $V$  in advance, the representation of the multiset  $\langle m_1, a_1 \rangle \cdots \langle m_n, a_n \rangle$  by the string  $a_1^{m_1} \cdots a_n^{m_n}$  is unique. The set of all finite multisets over an alphabet  $V$  is denoted by  $V^\circ$ .

The family of regular and recursively enumerable string languages is denoted by  $REG$  and  $RE$ , respectively. For more details of formal language theory the reader is referred to the monographs and handbooks in this area as [2] and [11].

A *register machine* is a tuple  $M = (m, B, l_0, l_h, P)$ , where  $m$  is the number of registers,  $P$  is the set of instructions bijectively labeled by elements of  $B$ ,  $l_0 \in B$  is the initial label, and  $l_h \in B$  is the final label. The instructions of  $M$  can be of the following forms:

- $l_1 : (ADD(j), l_2, l_3)$ , with  $l_1 \in B \setminus \{l_h\}$ ,  $l_2, l_3 \in B$ ,  $1 \leq j \leq m$ .  
Increase the value of register  $j$  by one, and non-deterministically continue with instruction  $l_2$  or  $l_3$ . This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$ , with  $l_1 \in B \setminus \{l_h\}$ ,  $l_2, l_3 \in B$ ,  $1 \leq j \leq m$ .  
If the value of register  $j$  is zero then continue with instruction  $l_3$ , otherwise decrease the value of register  $j$  by one and continue with instruction  $l_2$ . The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$ .  
Stop the execution of the register machine.

A *configuration* of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction to be executed. Computations start by executing the first instruction of  $P$  (labeled with  $l_0$ ), and terminate with reaching the *HALT*-instruction.

Register machines provide a simple universal computational model [7]. In the generative case as we need it later, we start with empty registers, use the first two registers for the necessary computations and take as results the contents of the  $k$  registers 3 to  $k+2$  in all possible halting computations; during a computation of  $M$ , only the registers 1 and 2 can be decremented. In the following, we shall call a specific model of P systems *computationally complete* or *universal* if and only if for any (generating) register machine  $M$  we can effectively construct an equivalent P system  $\Pi$  of that type simulating each step of  $M$  in a bounded number of steps and yielding the same results.

## 2.1 P Systems

The basic ingredients of a (cell-like) P system are the membrane structure, the objects placed in the membrane regions, and the evolution rules. The *membrane*

*structure* is a hierarchical arrangement of membranes. Each membrane defines a *region/compartment*, the space between the membrane and the immediately inner membranes; the outermost membrane is called the *skin membrane*, the region outside is the *environment*, also indicated by (the label) 0. Each membrane can be labeled, and the label (from a set *Lab*) will identify both the membrane and its region. The membrane structure can be represented by a rooted tree (with the label of a membrane in each node and the skin in the root), but also by an expression of correctly nested labeled parentheses. The *objects* (multisets) are placed in the compartments of the membrane structure and usually represented by strings, with the multiplicity of a symbol corresponding to the number of occurrences of that symbol in the string. The *evolution rules* are multiset rewriting rules of the form  $u \rightarrow v$ , where  $u$  is a multiset of objects from a given set  $O$  and  $v = (b_1, tar_1) \dots (b_k, tar_k)$  with  $b_i \in O$  and  $tar_i \in \{here, out, in\}$  or  $tar_i \in \{here, out\} \cup \{in_j \mid j \in Lab\}$ ,  $1 \leq i \leq k$ . Using such a rule means “consuming” the objects of  $u$  and “producing” the objects  $b_1, \dots, b_k$  of  $v$ ; the *target indications* *here*, *out*, and *in* mean that an object with the target *here* remains in the same region where the rule is applied, an object with the target *out* is sent out of the respective membrane (in this way, objects can also be sent to the environment, when the rule is applied in the skin region), while an object with the target *in* is sent to one of the immediately inner membranes, non-deterministically chosen, whereas with  $in_j$  this inner membrane can be specified directly. In general, we omit the target indication *here*.

Formally, a (cell-like) P system is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$$

where  $O$  is the alphabet of objects,  $\mu$  is the membrane structure (with  $m$  membranes),  $w_1, \dots, w_m$  are multisets of objects present in the  $m$  regions of  $\mu$  at the beginning of a computation,  $R_1, \dots, R_m$  are finite sets of evolution rules, associated with the regions of  $\mu$ , and  $f$  is the label of the membrane region from which the outputs are taken ( $f = 0$  indicates that the output is taken from the environment).

If a rule  $u \rightarrow v$  has at least two objects in  $u$ , then it is called *cooperative*, otherwise it is called *non-cooperative*. In *catalytic P systems* we use non-cooperative as well as *catalytic rules* which are of the form  $ca \rightarrow cv$ , where  $c$  is a special object which never evolves and never passes through a membrane (both these restrictions can be relaxed), but it just assists object  $a$  to evolve to the multiset  $v$ . In a *purely catalytic P system* we only allow catalytic rules. In both catalytic and purely catalytic P systems, we replace  $O$  by  $O, C$  in order to specify those objects from  $O$  which are the catalysts in the set  $C$ .

The evolution rules are used in the (non-deterministic) *maximally parallel* way, i.e., in any computation step of  $\Pi$  we choose a multiset of rules from the sets  $R_1, \dots, R_m$  in such a way that no further rule can be added to it so that the obtained multiset would still be applicable to the existing objects in the membrane regions  $1, \dots, m$ .

The membranes and the objects present in the compartments of a system at a given time form a *configuration*; starting from a given *initial configuration*

and using the rules as explained above, we get *transitions* among configurations; a sequence of transitions forms a *computation*. A computation is *halting* if it reaches a configuration where no rule can be applied. With a halting computation we associate a *result*, in the form of the number of objects present in region  $f$  in the halting configuration. The set of vectors of non-negative integers and the set of (Parikh) vectors of non-negative integers obtained as results of halting computations in  $\Pi$  are denoted by  $N(\Pi)$  and  $Ps(\Pi)$ , respectively.

The family of sets  $Y(\Pi)$ ,  $Y \in \{N, Ps\}$ , computed by P systems with at most  $m$  membranes and cooperative rules and with non-cooperative rules is denoted by  $YOP_m(coop)$  and  $YOP_m(ncoo)$ , respectively. It is well known that for any  $m \geq 1$ ,  $YREG = YOP_m(ncoo) \subset NOP_m(coop) = YRE$ , see [8].

The family of sets  $Y(\Pi)$ ,  $Y \in \{N, Ps\}$ , computed by (purely) catalytic P systems with at most  $m$  membranes and at most  $k$  catalysts is denoted by  $YOP_m(cat_k)$  ( $YOP_m(pcat_k)$ ); from [3] we know that, with the results being sent to the environment, we have  $YOP_1(cat_2) = YOP_1(pcat_3) = YRE$ .

If we allow catalysts to move from one membrane region to another one, then we speak of *P systems with mobile catalysts*. The families of sets  $N(\Pi)$  and  $Ps(\Pi)$  computed by P systems with at most  $m$  membranes and  $k$  mobile catalysts are denoted by  $NOP_m(mcat_k)$  and  $PsOP_m(mcat_k)$ , respectively. If we only allow catalytic rules, the corresponding families of sets  $N(\Pi)$  and  $Ps(\Pi)$  are denoted by  $NOP_m(mpcat_k)$  and  $PsOP_m(mpcat_k)$ , respectively.

For all the variants of P systems of type  $X$ , we may consider to label all the rules in the sets  $R_1, \dots, R_m$  in a one-to-one manner by labels from a set  $H$  and to take a set  $W$  containing subsets of  $H$ . Then a *P system with label selection* is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, H, W, f)$$

where  $\Pi' = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$  is a P system as defined above,  $H$  is a set of labels for the rules in the sets  $R_1, \dots, R_m$ , and  $W \subseteq 2^H$ . In any transition step in  $\Pi$  we first select a set of labels  $U \in W$  and then apply a non-empty multiset  $R$  of rules such that all the labels of these rules in  $R$  are in  $U$  in the maximally parallel way, i.e., the set  $R$  cannot be extended by any further rule with a label from  $U$  so that the obtained multiset of rules would still be applicable to the existing objects in the membrane regions  $1, \dots, m$ . The family of sets  $N(\Pi)$  and  $Ps(\Pi)$  computed by P systems with label selection with at most  $m$  membranes and rules of type  $X$  is denoted by  $NOP_m(X, ls)$  and  $PsOP_m(X, ls)$ , respectively.

Another method to control the application of the labeled rules is to use control languages (see [6] and [1]). A *controlled P system* is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, H, L, f)$$

where  $\Pi' = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$  is a P system as defined above,  $H$  is a set of labels for the rules in the sets  $R_1, \dots, R_m$ , and  $L$  is a string language over  $2^H$  from a family  $FL$ . Every successful computation in  $\Pi$  has to follow a control word  $U_1 \dots U_n \in L$ : in transition step  $i$ , only rules with labels in  $U_i$  are

allowed to be applied (but again in the maximally parallel way, i.e., we have to apply a multiset  $R$  of rules with labels in  $U_i$  which cannot be extended by any rule with a label in  $U_i$  such that the resulting multiset would still be applicable), and after the  $n$ -th transition, the computation halts; we may relax this end condition, and then we speak of *weakly controlled P systems*. If  $L = (U_1 \dots U_p)^*$ ,  $\Pi$  is called a *(weakly) time-varying P system*: in the computation step  $pn + i$ ,  $n \geq 0$ , rules from the set  $U_i$  have to be applied;  $p$  is called the *period*. The family of sets  $Y(\Pi)$ ,  $Y \in \{N, Ps\}$ , computed by (weakly) controlled P systems and (weakly) time-varying P systems with period  $p$ , with at most  $m$  membranes and rules of type  $X$  as well as control languages in  $FL$  is denoted by  $YOP_m(X, C(FL))$  ( $YOP_m(X, wC(FL))$ ) and  $YOP_m(X, TV_p)$  ( $YOP_m(X, wTV_p)$ ), respectively.

In the following sections we will prove several computational completeness results for purely catalytic P systems with only two catalysts: first we will give a proof for P systems with label selection, then for time-varying P systems, and finally for P systems with mobile catalysts.

Before we are going to show these special computational completeness results, we exhibit some general features of purely catalytic P systems:

*Remark 1.* For purely catalytic P systems with only one catalyst collecting the results in the environment, we only have to consider the minimal membrane structure  $[ ]_1$  with the catalyst being in the skin membrane: in order to allow for sending out the results from the skin membrane, the catalyst has to be there; any symbol that would be sent to an inner membrane would be lost anyway for being used in the further steps of a computation; hence, we can simply omit these symbols to even be generated and therefore can omit all inner membranes, too.

As a consequence of these observations we infer that the maximal parallelism in the case of purely catalytic P systems with only one catalyst reduces to the sequential application of rules where in each step of a computation exactly one rule is applied:

**Lemma 1.** *For purely catalytic P systems with only one catalyst collecting the results in the environment, even combined with some additional control mechanism, we get the same results as for sequential P systems with the minimal membrane structure  $[ ]_1$  (eventually equipped with the same additional control mechanism).*

As is well known, sequential P systems with the minimal membrane structure  $[ ]_1$  (or even any other more complicated membrane structure) correspond with multiset rewriting systems using only context-free rules, i.e., we have:

**Theorem 1.** *For any  $m \geq 1$  and any  $Y \in \{N, Ps\}$ ,*

$$YREG = YOP_m(ncoo) = YOP_m(pcat_1) = YOP_m(pcat_1, ls).$$

*Proof.* We only have to show  $YOP_m(pcat_1, ls) \subseteq YOP_m(pcat_1)$ . Yet the additional control mechanism of allowing to select only rules with the same label in each computation step does not increase the generative power, as in every step we anyway can only use exactly one rule, i.e., label selection in the case of purely catalytic P systems with only one catalyst does not yield additional power.  $\square$

*Remark 2.* For purely catalytic P systems with only two catalysts collecting the results in the environment, we only have to consider the membrane structures  $[ ]_1$  and  $[ [ ]_2 ]_1$ : as in Remark 1 we can argue that at least one catalyst has to be in the skin membrane. If only one catalyst is in the skin membrane and the second catalyst is in an inner membrane, then this second one has to be in a membrane directly connected with the skin region, otherwise it would not be able to contribute to the computations in the P system.

*Remark 3.* Except for the case of purely catalytic P systems with mobile catalysts, for all other cases of control mechanisms considered in this paper (and for many others considered in the literature so far, too) we can restrict ourselves to the minimal membrane structure  $[ ]_1$  when dealing with purely catalytic P systems with only two catalysts: instead of letting the second catalyst work in a second membrane, we put it into the skin membrane and let it work on primed symbols: symbols originally sent into the inner membrane now get primed; on the other hand, instead of sending out symbols from the second membrane, we just generate unprimed symbols with this second catalyst. This procedure of using marked symbols within the skin membrane instead of using inner membranes is well known in the P systems area as “flattening procedure”, even for any number of inner membranes.

*Remark 4.* For P systems with mobile catalysts, the case with the minimal membrane structure  $[ ]_1$  corresponds with the original question whether the inclusion  $YOP_1(pcat_2) \subseteq YRE$  is strict or not; therefore, in this case the solution being optimal with respect to the membrane structure we could aim for proving computational completeness was getting a result for the membrane structure  $[ [ ]_2 ]_1$ , and as will be shown in Section 5, two mobile catalyst moving within this simple membrane structure  $[ [ ]_2 ]_1$  are already sufficient for obtaining computational completeness.

### 3 Computational Completeness of P Systems with Label Selection

Whereas with Theorem 1 we have shown that  $YREG = YOP_m(pcat_1, ls)$  for any  $m \geq 1$ , in [4] it was shown that  $YOP_1(cat_1, ls) = YRE$ ,  $Y \in \{N, Ps\}$ ; the following theorem establishes the corresponding result for purely catalytic P systems with two catalysts.

**Theorem 2.**  $YOP_1(pcat_2, ls) = YRE$ ,  $Y \in \{N, Ps\}$ .

*Proof.* We only prove the inclusion  $PsRE \subseteq PsOP_1(pcat_2, ls)$ . Let us consider a register machine  $M = (n + 2, B, l_0, l_h, I)$  with only the first and the second register ever being decremented, and let  $A = \{a_1, \dots, a_{n+2}\}$  be the set of objects for representing the contents of the registers 1 to  $n + 2$  of  $M$ . We construct the following P system:

$$\begin{aligned} \Pi &= (O, \{c_1, c_2\}, [\ ]_1, c_1 c_2 d l_0, R_1, H, W, 0), \\ O &= A \cup B \cup \{c_1, c_2, d, \#\}, \\ H &= \{l, l' \mid l \in B\} \cup \{l_x \mid x \in \{1, 2, d, \#\}\}, \end{aligned}$$

and the sets of labels in  $W$  and the rules for  $R_1$  are defined as follows:

**A.** Let  $l_i : (\text{ADD}(r), l_j, l_k)$  be an ADD instruction in  $I$ . If  $r > 2$ , then the (labeled) rules

$$l_i : c_1 l_i \rightarrow c_1 l_j (a_r, out), \quad l'_i : c_1 l_i \rightarrow c_1 l_k (a_r, out),$$

are introduced, and for  $r \in \{1, 2\}$ , we introduce the rules

$$l_i : c_1 l_i \rightarrow c_1 l_j a_r, \quad l'_i : c_1 l_i \rightarrow c_1 l_k a_r.$$

In both cases, we define  $\{l_i, l'_i\}$  to be the corresponding set of labels in  $W$ . The contents of each register  $r$ ,  $r \in \{1, 2\}$ , is represented by the number of objects  $a_r$  present in the skin membrane; any object  $a_r$  with  $r > 2$  is immediately sent out into the environment.

**B.** The simulation of a SUB instruction  $l_i : (\text{SUB}(r), l_j, l_k)$ , for  $r \in \{1, 2\}$ , is carried out by the following rules and the corresponding sets of labels in  $W$ : For the case that the register  $r$ ,  $r \in \{1, 2\}$ , is not empty we take the (labeled) rules

$$l_i : c_1 l_i \rightarrow c_1 l_j, \quad l_r : c_2 a_r \rightarrow c_2, \quad l_d : c_2 d \rightarrow c_2 \#,$$

(if no symbol  $a_r$  is present, i.e., if the register  $r$  is empty, then the trap symbol  $\#$  is introduced) and for the case that the register  $r$  is empty, we introduce the rules

$$l'_i : c_1 l_i \rightarrow c_1 l_k, \quad l'_r : c_2 a_r \rightarrow c_2 \#$$

(if at least one symbol  $a_r$  is present, i.e., if the register  $r$  is not empty, then the trap symbol  $\#$  is introduced); the corresponding sets of labels to be taken into  $W$  are  $\{l_i, l_r, l_d\}$  and  $\{l'_i, l'_r\}$ , respectively. In both cases, the simulation of the SUB instruction works correctly if we have made the right choice.

**C.** We also add the labeled rule  $l_\# : \# \rightarrow \#$  to  $R_1$  and  $\{\#\}$  to  $W$ , hence, the computation cannot halt once the trap symbol  $\#$  has been generated.

In sum, we have shown  $Ps(M) = Ps(\Pi)$ , which completes the proof.  $\square$



## 4 Computational Completeness of Time-Varying P Systems

In [4] it was shown that  $YOP_1(cat_1, \alpha TV_6) = YRE$ ,  $Y \in \{N, Ps\}$ ,  $\alpha \in \{\lambda, w\}$ ; the following theorem establishes the corresponding result for purely catalytic P systems with two catalysts.

**Theorem 3.**  $YOP_1(pcat_2, \alpha TV_6) = YRE$ ,  $Y \in \{N, Ps\}$ ,  $\alpha \in \{\lambda, w\}$ .

*Proof.* We only prove the inclusion  $PsRE \subseteq PsOP_1(pcat_2, \alpha TV_6)$ . Let us consider a register machine  $M = (n+2, B, l_0, l_h, I)$  with only the first and the second register ever being decremented. Again, we define  $A = \{a_1, \dots, a_{n+2}\}$  and divide the set of labels  $B \setminus \{l_h\}$  into three disjoint subsets:

$$\begin{aligned} B_+ &= \{l_i \mid l_i : (\text{ADD}(r), l_j, l_k) \in I\}, \\ B_{-r} &= \{l_i \mid l_i : (\text{SUB}(r), l_j, l_k) \in I\}, \quad r \in \{1, 2\}; \end{aligned}$$

moreover, we define  $B_- = B_{-1} \cup B_{-2}$  as well as

$$B' = \{l, \tilde{l}, \hat{l} \mid l \in B \setminus \{l_h\}\} \cup \{l^-, l^0, \bar{l}^-, \bar{l}^0 \mid l \in B_-\}.$$

The main challenge in the construction for the time-varying P system  $\Pi$  is that the catalyst has to fulfill its task to erase an object  $a_r$ ,  $r \in \{1, 2\}$ , for both objects in the same membrane where all other computations are carried out, too; hence, at a specific moment in the cycle of period six, parts of simulations of different instructions have to be coordinated in parallel. The basic components of the time-varying P system  $\Pi$  are defined as follows (we here do not distinguish between a rule and its label):

$$\begin{aligned} \Pi &= (O, \{c_1, c_2\}, [\ ]_1, c_1 c_2 l_0, R_1 \cup \dots \cup R_6, R_1 \cup \dots \cup R_6, (R_1 \dots R_6)^*, 0), \\ O &= A \cup \{a'_1, a'_2\} \cup B' \cup \{c_1, c_2, h, l_h, \#\}. \end{aligned}$$

We now list the rules in the sets of rules  $R_i$  to be applied in computation steps  $6n+i$ ,  $n \geq 0$ ,  $1 \leq i \leq 6$ :

**R<sub>1</sub>:** in this step, the ADD instructions are simulated, i.e., for each  $l_i : (\text{ADD}(r), l_j, l_k) \in I$  we take

$c_1 l_i \rightarrow c_1 a_r \tilde{l}_j$ ,  $c_1 l_i \rightarrow c_1 a_r \tilde{l}_k$  (only in the sixth step of the cycle, from  $\tilde{l}_j$  and  $\tilde{l}_k$  the corresponding unmarked labels  $l_j$  and  $l_k$  will be generated); in order to obtain the output in the environment, for  $r \geq 3$ ,  $a_r$  has to be replaced by  $(a_r, out)$ ;

$c_1 l \rightarrow c_1 l^-$ ,  $c_1 l \rightarrow c_1 l^0$  initiate the simulation of a SUB instruction for register 1 labeled by  $l \in B_{-1}$ , i.e., we make a non-deterministic guess whether register  $r$  is empty (with introducing  $l^0$ ) or not (with introducing  $l^-$ );

$c_1 l \rightarrow c_1 \hat{l}$  marks a label  $l \in B_{-2}$  (the simulation of such a SUB instruction for register 2 will start in step 4 of the cycle);

$c_2\# \rightarrow c_2\#$  keeps the trap symbol  $\#$  alive guaranteeing an infinite loop once  $\#$  has been generated;

$c_2h \rightarrow c_2$  eliminates the auxiliary object  $h$  needed for simulating SUB instructions and eventually generated two steps before.

**R<sub>2</sub>**: in the second and the third step, the SUB instructions on register 1 are simulated, i.e., for all  $l \in B_{-1}$  we start with

$c_1a_1 \rightarrow c_1a'_1$  (if present, exactly one copy of  $a_1$  can be primed, but only if a label  $l^-$  for some  $l$  from  $B_{-1}$  is present) and

$c_2l^- \rightarrow c_2\bar{l}^-h, c_2l^- \rightarrow c_2\bar{l}^0$  for all  $l \in B_{-1}$ ;

all other labels  $\tilde{l}$  for  $l \in B_+ \cup B_{-2}$  block the catalyst  $c_1$  from erasing a copy of  $a_1$  by forcing the application of the corresponding rules  $c_1\tilde{l} \rightarrow c_1\tilde{l}$  for  $c_1$  in order to avoid the introduction of the trap symbol  $\#$  by the enforced application of a rule  $c_1\tilde{l} \rightarrow c_2\#$  with the catalyst  $c_2$ , i.e., we take

$c_1\bar{l} \rightarrow c_1\bar{l}, c_2\bar{l} \rightarrow c_2\#$  for all  $l \in B$ , and

$c_1\hat{l} \rightarrow c_1\hat{l}, c_2\hat{l} \rightarrow c_2\#$  for all  $l \in B_{-2}$ ;

$c_2\# \rightarrow c_2\#$  keeps the computation alive once the trap symbol has been introduced.

**R<sub>3</sub>**: for all  $l_i : (\text{SUB}(1), l_j, l_k) \in I$  we take

$c_1\bar{l}_i^0 \rightarrow c_1\bar{l}_k, c_2a'_1 \rightarrow c_2\#, c_2\bar{l}_i^0 \rightarrow c_2\#$  (zero test; if a primed copy of  $a_1$  is present, then the trap symbol  $\#$  is generated);

$c_2\bar{l}_i^- \rightarrow c_2\bar{l}_j, c_1a'_1 \rightarrow c_1, c_1h \rightarrow c_1\#$  (decrement; the auxiliary symbol  $h$  is needed to keep the catalyst  $c_1$  busy with generating the trap symbol  $\#$  if we have taken the wrong guess when assuming the register 1 to be non-empty);

$c_1\tilde{l} \rightarrow c_1\tilde{l}, c_2\tilde{l} \rightarrow \#$  for all  $l \in B$  (with these labels, we just pass through this step);

$c_1\hat{l} \rightarrow c_1\hat{l}, c_2\hat{l} \rightarrow \#$  for all  $l \in B_{-2}$  (these labels pass through this step to become active in the next step);

$c_2\# \rightarrow c_2\#$ .

**R<sub>4</sub>**: in the fourth step, the simulation of SUB instructions on register 2 is initiated by using

$c_1\hat{l} \rightarrow c_1l^-, c_1\hat{l} \rightarrow c_1l^0$  for all  $l \in B_{-2}$ , i.e., we make a non-deterministic guess whether register  $r$  is empty (with introducing  $l^0$ ) or not (with introducing  $l^-$ );

$c_1\tilde{l} \rightarrow c_1\tilde{l}, c_2\tilde{l} \rightarrow c_2\#$  for all  $l \in B$  (with all other labels, we already only pass through this step);

$c_2\# \rightarrow c_2\#$ ,

$c_2h \rightarrow c_2$  (if  $h$  has been introduced by  $c_2l^- \rightarrow c_2\bar{l}^-h$  in the second step for some  $l \in B_{-1}$ , the second catalyst now is free to erase  $h$ ).

**R<sub>5</sub>**: in the fifth and the sixth step, the SUB instructions on register 2 are simulated, i.e., for all  $l \in B_{-2}$  we start with

$c_1a_2 \rightarrow c_1a'_2$  (if present, exactly one copy of  $a_2$  can be primed) and

$c_2l^- \rightarrow c_2\bar{l}^-h, c_2l^- \rightarrow c_2\bar{l}^0$  for all  $l \in B_{-2}$ ;

$c_1\tilde{l} \rightarrow c_1\tilde{l}, c_2\tilde{l} \rightarrow c_2\#$  for all  $l \in B$ ;

$$c_2\# \rightarrow c_2\#.$$

**R<sub>6</sub>**: the simulation of SUB instructions  $l_i : (\text{SUB}(2), l_j, l_k) \in I$  on register 2 is finished by

$c_1\bar{l}_i^0 \rightarrow c_1l_k, c_2a'_2 \rightarrow c_2\#, c_2\bar{l}_i^0 \rightarrow c_2\#$  (zero test; if a primed copy of  $a_2$  is present, then the trap symbol  $\#$  is generated);

$c_2\bar{l}_i^- \rightarrow c_2l_j, c_1a'_2 \rightarrow c_1, c_1h \rightarrow c_1\#$  (decrement; the auxiliary symbol  $h$  is needed to keep the catalyst  $c_1$  busy with generating the trap symbol  $\#$  if we have taken the wrong guess when assuming the register 2 to be non-empty; if it is not used, it can be erased in the next step by using  $c_2h \rightarrow c_2$  in  $R_1$ );

$$c_1\bar{l} \rightarrow c_1l, c_2\bar{l} \rightarrow c_2\# \text{ for all } l \in B;$$

$$c_2\# \rightarrow c_2\#.$$

Without loss of generality, we may assume that the final label  $l_h$  in  $M$  is only reached by using a zero test on register 2; then, at the beginning of a new cycle, after a correct simulation of a computation from  $M$  in the time-varying P system  $\Pi$  no rule will be applicable in  $R_1$  (another possibility would be to take  $c_1\bar{l}_i^0 \rightarrow c_1$  instead of  $c_1\bar{l}_i^0 \rightarrow c_1l_h$  in  $R_6$ ).

At the end of the cycle, in case all guesses have been correct, the requested instruction of  $M$  has been simulated and the label of the next instruction to be simulated is present in the skin membrane. Only in the case that  $M$  has reached the final label  $l_h$ , the computation in  $\Pi$  halts, too, but only if during the simulation of the computation of  $M$  in  $\Pi$  no trap symbol  $\#$  has been generated; hence, we conclude  $Ps(M) = Ps(\Pi)$ .  $\square$

## 5 Computational Completeness of P Systems with Mobile Catalysts

Whereas in [4] it was shown that  $YOP_3(mcat_1) = YRE$ ,  $Y \in \{N, Ps\}$ , the following theorem establishes a similar result for purely catalytic P systems with two mobile catalysts proving  $YOP_2(mpcat_2) = YRE$ ; as we see, in the case of purely catalytic P systems with mobile catalysts there might be a trade-off between the number of membranes and the number of mobile catalysts, as their sum in both cases is four.

**Theorem 4.**  $YOP_2(mpcat_2) = YRE$ ,  $Y \in \{N, Ps\}$ .

*Proof.* We only prove the inclusion  $PsRE \subseteq PsOP_2(mpcat_2)$ . Let us consider a register machine  $M = (n+2, B, l_0, l_h, I)$  with only the first and the second register ever being decremented. Again we define  $A = \{a_1, \dots, a_{n+2}\}$  as the set of objects for representing the contents of the registers 1 to  $n+2$  of  $M$ ; the terminal symbols  $a_r$  for  $r > 2$  are collected in the environment, whereas the symbols  $a_1$  and  $a_2$  are stored in the innermost membrane 2. The skin region is used to take those symbols we need for guiding the simulation of a computation in  $M$ .

The set of labels  $B \setminus \{l_h\}$  is divided into three disjoint subsets:

$$\begin{aligned} B_+ &= \{l_i \mid l_i : (\text{ADD}(r), l_j, l_k) \in I\}, \\ B_{-r} &= \{l_i \mid l_i : (\text{SUB}(r), l_j, l_k) \in I\}, \quad r \in \{1, 2\}; \end{aligned}$$

moreover, we define  $B_- = B_{-1} \cup B_{-2}$  as well as

$$B' = \{l \mid l \in B\} \cup \{l', l'', l^-, l^0 \mid l \in B_-\}.$$

We now construct the following P system:

$$\begin{aligned} \Pi &= (O, \{c_1, c_2\}, \begin{bmatrix} & \\ & \end{bmatrix}_2, c_1 c_2 l_0 l'_0, R_1, R_2, 0), \\ O &= A \cup B' \cup \{c_1, c_2, \#\}. \end{aligned}$$

In general, for simulating an instruction labeled by  $l$  from  $M$ , we start with  $c_1 c_2 l$  in the skin region, whereas region 2 stores the contents of the registers 1 and 2 by the corresponding numbers of symbols  $a_1$  and  $a_2$ . The sets of rules are constructed as follows:

**A.** Let  $l_i : (\text{ADD}(r), l_j, l_k)$  be an ADD instruction in  $I$ . If  $r > 2$ , then the rules  $c_1 l_i \rightarrow c_1 l_j(a_r, out)$ ,  $c_1 l_i \rightarrow c_1 l_k(a_r, out)$  are introduced in  $R_1$ ; if  $r \in \{1, 2\}$ , in  $R_1$  we introduce the rules  $c_1 l_i \rightarrow c_1 l_j(a_r, in)$ ,  $c_1 l_i \rightarrow c_1 l_k(a_r, in)$ . In both cases, the catalyst  $c_2$  remains inactive.

**B.** The simulation of a SUB instruction  $l_i : (\text{SUB}(r), l_j, l_k)$ ,  $r \in \{1, 2\}$ , is carried out by the following rules:

In the first step, from  $R_1$  we use the rule  $c_r l_i \rightarrow l'_i(c_r, in)$  to send the corresponding catalyst  $c_r$  into membrane 2 for there to check the presence of a symbol  $a_r$  and eventually erase one copy of it. The second catalyst again remains inactive and stays in the skin region.

In the second step, in the skin region we use the rule  $c_{3-r} l'_i \rightarrow c_{3-r} l''_i$ . In the second membrane, the catalyst  $c_r$  will use the rule  $c_r a_r \rightarrow (c_r l_i^-, out)$  from  $R_2$  in case the register  $r$  is not empty, otherwise  $c_r$  has to stay in region 2.

In the third step, we may face two situations in the skin region: if  $c_r$  is present again after having decremented the number of symbols  $a_r$  in membrane 2, then we have to use the rules  $c_r l''_i \rightarrow c_r l_j$  and  $c_{3-r} l''_i \rightarrow c_{3-r}$ , thus already getting the configuration  $c_1 c_2 l_j$  in the skin region for starting the simulation of the instruction labeled by  $l_j$ . Moreover, the rule  $c_r l_i^- \rightarrow c_r \#$  guarantees that the two catalysts work together in a correct way, because otherwise, if  $c_r$  had to stay in membrane 2, in the skin region we have to use the rule  $c_{3-r} l''_i \rightarrow c_{3-r}(l_i^0, in)$ . In this case, we need another step to finish the simulation: in membrane 2 we now have the two symbol  $l_i^0$  allowing  $c_r$  to get out by using the rule  $c_r l_i^0 \rightarrow (c_r l_k, out)$ , which yields the configuration  $c_1 c_2 l_k$  in the skin region for starting the simulation of the instruction labeled by  $l_k$ .

Finally, the rule  $c_2 \# \rightarrow c_2 \#$  in  $R_1$  guarantees that the computation will never halt once the trap symbol  $\#$  has been introduced during the simulation of a SUB instruction.

In all cases, the simulation of the SUB instruction works correctly, and we return to a configuration with the two catalysts and  $l$  for some label from  $B$  in the skin region.

The computation in  $\Pi$  halts if and only if we reach the configuration with  $c_1 c_2 l_h$  in the skin region. In sum, we have the equality  $Ps(M) = Ps(\Pi)$ , which completes the proof.  $\square$

## 6 Final Remarks

Several new computational completeness results for purely catalytic P systems using only two catalysts together with some additional control mechanism were established in this paper, but the original problem of characterizing the sets of non-negative integers generated by purely catalytic P systems with only two catalysts still remains open. For other variants of additional control mechanisms, the case of purely catalytic P systems with two catalysts remains for future research, too.

**Acknowledgements.** This paper is a continuation of the results established together with Gheorghe Păun in [4], where we considered several variants of control mechanisms to obtain computational completeness for catalytic P systems with only one catalyst. I am very grateful to Gheorghe for all the interesting discussions on the topics elaborated in this paper, but especially for pushing me to again put some energy on considering these topics.

## References

1. A. Alhazov, R. Freund, H. Heikenwälder, M. Oswald, Yu. Rogozhin, S. Verlan, Sequential P systems with regular control. In: E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (Eds.): *Membrane Computing - 13th International Conference, CMC 2012*, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, LNCS 7762, Springer, 2013, 112–127.
2. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, 1989.
3. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science* **330**, 2005, 251–266.
4. R. Freund, Gh. Păun: How to obtain universality in P systems with one catalyst, to appear in *Proc. MCU 2013*.
5. S.N. Krishna, A. Păun: Results on catalytic and eolution-communication P systems. *New Generation Computing*, 22 (2004), 377–394.
6. K. Krithivasan, Gh. Păun, A. Ramanujan: On controlled P systems. *Fundamenta Informaticae*, to appear.
7. M. L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
8. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.*, 61 (2000), 108–143 (see also TUCS Report 208, November 1998, [www.tucs.fi](http://www.tucs.fi)).

9. Gh. Păun: Computing with membranes - a variant. *Intern. J. Found. Computer Sci.*, 11, 1 (2000), 167–182.
10. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
11. G. Rozenberg, A. Salomaa (Eds.): *Handbook of Formal Languages*, 3 volumes. Springer, 1997.
12. The P Systems Website: <http://ppage.psyste.ms.eu>.

# Solving SAT by P Systems with Active Membranes in Linear Time in the Number of Variables

Zsolt Gazdag

Department of Algorithms and their Applications  
Faculty of Informatics  
Eötvös Loránd University  
gazdagzs@inf.elte.hu

**Abstract.** In this paper we solve the SAT problem (the satisfiability problem of propositional formulas in conjunctive normal form) by a polynomially uniform family of P systems with active membranes in linear time in the number of propositional variables occurring in the input formula. In those polynomially uniform existing solutions which do not employ non-elementary membrane division or membrane creation the computation time depends also on the number of the clauses in the formula. In our solution we do not use non-elementary membrane division, but we use such membrane division rules where the labels of the involved membranes can change. We also use membrane creation rules, but compared to existing solutions with membrane creation rules our systems use asymptotically less objects and membranes during their computations.

**Keywords:** Membrane computing; P systems; SAT problem

## 1 Introduction

P systems with active membranes [9] are widely investigated variants of P systems [8]. These systems have the possibility of dividing elementary membranes which combined with the massive parallelism that is present in these systems can yield exponential workspace in linear time. This feature is frequently used in P system based efficient solutions of well known NP-complete problems such as the SAT problem. The SAT problem (satisfiability problem of propositional formulas) is probably the best known NP-complete decision problem where the question is whether a given propositional formula in conjunctive normal form (CNF) is satisfiable.

Solving SAT efficiently by P systems with active membranes is a subject of many papers in the literature (see e.g. [1], [2], [3], [4], [6], [7], [9], and [12]). These solutions differ, for example, in the types of the rules employed, the possibility of changing the labels of the membranes, and the use of the polarizations of the membranes. On the other hand, these solutions commonly work in a way where all possible truth valuations of the input formula are created and then a satisfying one (if it exists) is chosen.

The above mentioned works solve SAT by polynomially (semi-)uniform families of P systems. This means that the P systems in these families can be constructed in polynomial time by a deterministic Turing machine from the size of the input formula (in the uniform case) or from the formula itself (in the semi-uniform case). (For more details on polynomially (semi-)uniform families of P systems we refer to [11] or [12]). The size of the input formula is usually described by the number of distinct variables and the number of clauses in the formula. The P systems introduced in the above works can decide SAT in polynomial time in the size of the input formula. This means that the number of the computation steps of these systems usually depends also on the number of clauses. The only exceptions are the solutions of [4] and [6], where SAT is solved in linear time in the number of variables. However, this efficiency was achieved in these works by use of additional type of rules. In [4] the presented P systems employ non-elementary membrane division rules, while in [6] the P systems can create and dissolve membranes.

In [5] two families of polarizationless P systems were given which use neither non-elementary membrane division nor membranes creation, but still can solve SAT in linear time in the number of the variables in the input formula. These solutions implement a decision procedure which is strongly based on the well known resolution rule of propositional logic. However, the first solution is not polynomially uniform since its object alphabet is exponential in the number of the variables. The second solution, on the other hand, uses a polynomial time constructable family of P systems, but the P systems are constructed from the input formula, thus it is a semi-uniform solution.

In this paper we present a family of P systems that is based on these systems but do not have their drawbacks in the following sense. Our new solution is a polynomially uniform solution still capable to decide the satisfiability of a formula in linear time in the number of variables. On the other hand, in contrast to the solutions of [5], we could not avoid the use membrane creation. Because of this reason, the time efficiency of our solution is comparable to that of the solution of [6]. However, our P systems use asymptotically less objects and membranes during their computations. A more detailed comparison of the solution of [6] and our solution will be given after presenting the main result of this paper.

It also should be pointed out that our solution uses such elementary membrane division rules which can change the labels of the membranes involved. Although these rules are powerful ones, it seems that their relabelling feature can be traded for the use of polarizations of the membranes. We will discuss this question in the conclusions section.

The paper is organised as follows. In Section 2 we clarify the used notations and notions and give the necessary definitions and preliminary results. Section 3 contains our families of P systems, and Section 4 presents some concluding remarks.



## 2 Definitions

**Alphabets, Words, Multisets.** An *alphabet*  $\Sigma$  is a non-empty and finite set of symbols. The elements of  $\Sigma$  are called *letters*.  $\Sigma^*$  denotes the set of all finite *words* (or *strings*) over  $\Sigma$ , including the *empty word*  $\varepsilon$ . We will use *multisets* of objects in the membranes of a P system. As usual, these multisets will be represented by strings over the object alphabet of the P system. The set of natural numbers is denoted by  $\mathbb{N}$ . For a number  $n \in \mathbb{N}$ , we denote the set  $\{1, 2, \dots, n\}$  by  $[n]$ .

**The SAT Problem.** Let  $X = \{x_1, x_2, x_3, \dots\}$  be a recursively enumerable set of *propositional variables* (*variables*, to be short), and, for every  $n \in \mathbb{N}$ , let  $X_n := \{x_1, \dots, x_n\}$ . An *interpretation of the variables in  $X_n$*  (or just an *interpretation* if  $X_n$  is clear from the context) is a function  $\mathcal{I} : X_n \rightarrow \{\text{true}, \text{false}\}$ .

The variables and their negations are called *literals*. A *clause*  $C$  is a disjunction of finitely many pairwise different literals satisfying the condition that there is no  $x \in X$  such that both  $x$  and  $\bar{x}$  occur in  $C$ , where  $\bar{x}$  denotes the negation of  $x$ . The set of all clauses over the variables in  $X_n$  is denoted by  $\mathcal{C}_n$ . A *formula in conjunctive normal form* (CNF) is a conjunction of finitely many clauses. We denote the conjunction and the disjunction operator by  $\wedge$  and  $\vee$ , respectively. However, when it is more convenient, we will treat formulas in CNF as finite sets of clauses, where the clauses are finite sets of literals. A clause  $C \in \mathcal{C}_n$  is called a *complete clause* if, for every  $x \in X_n$ ,  $x \in C$  or  $\bar{x} \in C$ . Let *Form* be the set of all formulas in CNF over the variables in  $X$  and, for every  $n \in \mathbb{N}$ , let *Form<sub>n</sub>* be the set of those formulas in *Form* that have variables in  $X_n$ . It is easy to see that *Form* is a recursively enumerable set (notice that, for a given  $n \in \mathbb{N}$ , *Form<sub>n</sub>* is a finite set).

Let  $\varphi \in \text{Form}_n$  ( $n \in \mathbb{N}$ ) and let  $\mathcal{I}$  be an interpretation for  $\varphi$ . We say that  $\mathcal{I}$  *satisfies*  $\varphi$ , denoted by  $\mathcal{I} \models \varphi$ , if  $\varphi$  evaluates to *true* under the truth assignment defined by  $\mathcal{I}$ . Notice that  $\mathcal{I} \models \varphi$  if and only if, for every  $C \in \varphi$ ,  $\mathcal{I} \models C$ . We say that  $\varphi$  is *satisfiable* if there is an interpretation  $\mathcal{I}$  such that  $\mathcal{I} \models \varphi$ . The *SAT problem* (boolean satisfiability problem of propositional formulas in CNF) can be defined as follows:

*Given a formula  $\varphi$  in CNF, decide whether or not there is an interpretation  $\mathcal{I}$  such that  $\mathcal{I} \models \varphi$ .*

Let  $\varphi \in \text{Form}$ . The set of variables occurring in  $\varphi$ , denoted by  $\text{var}(\varphi)$ , is defined by  $\text{var}(\varphi) := \{x \in X \mid \exists C \in \varphi : x \in C \text{ or } \bar{x} \in C\}$ . Next we define an operation on a clause in  $\mathcal{C}_n$ . This operation is a key component in our method of solving the SAT problem by P systems. For a clause  $C \in \mathcal{C}_n$  and a set  $Y \subseteq X_n$  ( $n \in \mathbb{N}$ ) such that  $\text{var}(C) \cap Y = \emptyset$ , let  $C_Y$  be the following set of clauses. Assume that  $Y = \{x_{i_1}, \dots, x_{i_k}\}$  ( $k \leq n$ ,  $1 \leq i_1 < \dots < i_k \leq n$ ). Then let  $C_Y := \{C \cup \{l_1, \dots, l_k\} \mid j \in [k] : l_j \in \{x_{i_j}, \bar{x}_{i_j}\}\}$ . Intuitively,  $C_Y$  is the set of those clauses that can be created by adding, for every variable  $x \in Y$ ,  $x$  or  $\bar{x}$  to  $C$ . For example, if  $C = \{x_1, \bar{x}_2\}$  and  $Y = \{x_3\}$ , then  $C_Y = \{\{x_1, \bar{x}_2, x_3\}, \{x_1, \bar{x}_2, \bar{x}_3\}\}$ .

For a formula  $\varphi = \{C_1, \dots, C_m\} \in Form_n$  ( $m, n \in \mathbb{N}$ ), let  $\varphi' := \bigcup_{C \in \varphi} C_Y$ , where  $Y := X_n - \text{var}(C)$ .

The following statement claims that the satisfiability of a formula  $\varphi \in Form_n$  can be reduced to the question whether  $\varphi'$  contains every complete clause in  $\mathcal{C}_n$ .

**Proposition 1.** *For a formula  $\varphi \in Form_n$  ( $n \in \mathbb{N}$ ),  $\varphi$  is satisfiable if and only if  $|\varphi'| < 2^n$ .*

The formal proof of this statement can be found, for example, in [5]. We only note here that the correctness of this statement is based on the following observations. For a formula  $\varphi \in Form_n$ ,  $C \in \varphi$ , and  $x \in X_n - \text{var}(C)$ ,  $\varphi$  is satisfiable if and only if the formula  $(\varphi - C) \cup C_{\{x\}}$  is satisfiable. Moreover, trivially, a set of complete clauses is satisfiable if and only if it contains every complete clause in  $\mathcal{C}_n$ , for some  $n \in \mathbb{N}$ .

As an example consider the formula  $\varphi = (x_1 \vee \bar{x}_2) \wedge \bar{x}_1 \wedge x_2 \in Form_2$ . Let us denote the clauses of  $\varphi$  by  $C_1$ ,  $C_2$ , and  $C_3$ , respectively. Clearly  $\text{var}(C_1) = \{x_1, x_2\}$ ,  $\text{var}(C_2) = \{x_1\}$ , and  $\text{var}(C_3) = \{x_2\}$ . Thus, the clauses of  $\varphi'$  are  $C_1$ ,  $C_2 \cup \{x_2\}$ ,  $C_2 \cup \{\bar{x}_2\}$ ,  $C_3 \cup \{x_1\}$ , and  $C_3 \cup \{\bar{x}_1\}$ , i.e.,  $\varphi' = (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1 \vee x_2)$  (notice that the second clause of  $\varphi'$  can be created from both  $C_2$  and  $C_3$ ). As  $\varphi'$  contains every complete clause in  $\mathcal{C}_2$ , using Proposition 1 we can derive that  $\varphi$  is unsatisfiable.

Our P systems will be based on the above method of creation of complete clauses. In fact, we are going to implement Algorithm 1. In this algorithm the complete clauses are created iteratively, adding missing literals to the clauses of a formula step by step.

---

**Algorithm 1:** Deciding the satisfiability of a formula in CNF by creating complete clauses

---

**input** : A formula  $\varphi$  in CNF with  $n$  variables  
**output**: *yes* if  $\varphi$  is satisfiable, otherwise *no*

```

1 for  $i \leftarrow 1$  to  $n$  do
2    $\hat{\varphi} \leftarrow \emptyset$ ;
3   foreach  $C \in \varphi$  do
4     if  $x_i \notin \text{var}(C)$  then
5        $\hat{\varphi} \leftarrow \hat{\varphi} \cup C_{\{x_i\}}$ ;
6     else
7        $\hat{\varphi} \leftarrow \hat{\varphi} \cup C$ ;
8   end
9    $\varphi \leftarrow \hat{\varphi}$ ;
10 end
11 if  $|\varphi| = 2^n$  then
12    $\text{answer} \leftarrow \text{no}$ ;
13 else
14    $\text{answer} \leftarrow \text{yes}$ ;
15 return  $\text{answer}$ 

```

---

**Proposition 2.** *Let  $\varphi \in \text{Form}_n$  ( $n \in \mathbb{N}$ ). Then Algorithm 1 with input  $\varphi$  terminates with the answer yes if and only if  $\varphi$  is satisfiable.*

*Proof.* It is easy to see that after the  $n$ th iteration of the loop starting at line 1,  $\varphi$  equals to  $\varphi'$ . Thus the statement follows from Proposition 1.

**P Systems with Active Membranes.** We will use P systems with active membranes to solve SAT. In these P systems we will use such membrane division rules that can change the labels of the membranes involved. We will also use membrane creation and dissolution rules. On the other hand, we will not use the polarizations of the membranes, thus we leave out this feature from the definition of these systems. The following is the formal definition of the P systems we will use (see also [10]).

A (polarizationless) P system with active membranes is a construct  $\Pi = (O, H, \mu, w_1, \dots, w_m, R)$ , where:

- $m \geq 1$  (the *initial degree* of the system);
- $O$  is the *alphabet of objects*;
- $H$  is a finite set of *labels* for membranes;
- $\mu$  is a *membrane structure*, consisting of  $m$  membranes, labelled (not necessarily in a one-to-one manner) with elements of  $H$ ;
- $w_1, \dots, w_m$  are strings over  $O$ , describing the *multisets of objects* (every symbol in a string representing one copy of the corresponding object) placed in the  $m$  regions of  $\mu$ ;
- $R$  is a finite set of *developmental rules*, of the following forms:
  - (a)  $[a \rightarrow v]_h$ , for  $h \in H, a \in O, v \in O^*$   
(object evolution rules, associated with membranes and depending on the label 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)  $a[ ]_h \rightarrow [b]_h$ , for  $h \in H, a, b \in O$   
(communication rules, sending an object into a membrane; the label cannot be modified);
  - (c)  $[a]_h \rightarrow [ ]_h b$ , for  $h \in H, a, b \in O$   
(communication rules; an object is sent out of the membrane, possibly modified during this process; the label cannot be modified);
  - (d)  $[a]_h \rightarrow b$ , for  $h \in H, a, b \in O$   
(membrane dissolving rules; in reaction with an object, a membrane can be dissolved, while the object specified in the rule can be modified);
  - (e)  $a \rightarrow [b]_h$ , for  $h \in H, a, b \in O$   
(membrane creation rules; in reaction with an object a new membrane with label  $h$  can be created; the object  $a$  specified in the rule is replaced in the new membrane by the object  $b$ );
  - (f)  $[a]_{h_1} \rightarrow [b]_{h_2}[c]_{h_3}$ , for  $h_1, h_2, h_3 \in H, a, b, c \in O$   
(division rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with possibly different labels;

the object  $a$  specified in the rule is replaced in the two new membranes by (possibly new) objects  $b$  and  $c$  respectively, and the remaining objects are duplicated);

As usual,  $\Pi$  works in a *maximal parallel* manner:

- In one step, any object of a membrane that can evolve must evolve, but one object can be used by only one rule in (a)-(f);
- when some rules in (b)-(f) can be applied to a certain membrane, then one of them must be applied, but a membrane can be the subject of only one rule of these rules during each step.

We say that  $\Pi$  is a *recognizing P system* if

- $O$  has two designated objects *yes* and *no*, and every computation of  $\Pi$  halts and sends out to the environment either *yes* or *no*;
- $\Pi$  has a designated input membrane  $i_0$ ;
- for a word  $w$ , called the *input of  $\Pi$* ,  $w$  can be added to the system by placing it into the region  $i_0$  in the initial configuration.

A recognizing P system  $\Pi$  is called *deterministic* if it has only a single computation from its initial configuration to its unique halting configuration. It is called *confluent* if every computation of  $\Pi$  halts and sends out to the environment the same object (*yes* or *no*). A family  $\mathbf{\Pi} := (\Pi(i))_{i \in \mathbb{N}}$  of recognizing P systems called *polynomially uniform* if, for every  $n \in \mathbb{N}$ ,  $\Pi(n)$  can be constructed from  $n$  by a deterministic Turing machine in polynomial time in  $n$ .

We say that *SAT can be solved by a family  $\mathbf{\Pi} := (\Pi(i))_{i \in \mathbb{N}}$  of recognizing P systems* if, for a formula  $\varphi \in \text{Form}$  with size  $n$  ( $n \in \mathbb{N}$ ), starting  $\Pi(n)$  with a polynomial time encoding of  $\varphi$  in its input membrane,  $\Pi(n)$  sends out to the environment *yes* if and only if  $\varphi$  is satisfiable.

### 3 The Main Result

Here we present a polynomially uniform family  $\mathbf{\Pi} := (\Pi(i))_{i \in \mathbb{N}}$  of recognizing P systems that can solve SAT in linear time in the number of distinct variables in the input formula. As we have mentioned, the P systems in  $\mathbf{\Pi}$  implement Algorithm 1. We use the following encoding of formulas which is often used in the theory of P systems (see e.g. the definition of  $\text{cod}(\varphi)$  on page 314 in [11]). Let  $\varphi = C_1 \wedge \dots \wedge C_m$  be a formula. Then

$$\text{cod}(\varphi) := \bigcup_{j=1}^m (\{x_{j,i} \mid x_i \in C_j\} \cup \{\bar{x}_{j,i} \mid \bar{x}_i \in C_j\}).$$

Clearly, for every formula  $\varphi$  with  $m$  clauses and  $n$  variables,  $\text{cod}(\varphi) \subseteq O_{m,n}$ , where  $O_{m,n} := \{x_{j,i} \mid i \in [n], j \in [m]\} \cup \{\bar{x}_{j,i} \mid i \in [n], j \in [m]\}$ . We will use the size function  $\langle m, n \rangle := \frac{(n+m)(n+m+1)}{2} + n$  also used e.g. in [3] to represent the size of  $\varphi$ .

**Definition 1.** For every  $m, n \in \mathbb{N}$ , let  $\Pi(\langle m, n \rangle) := (O, H, \mu, w_{skin}, w_{aux}, w_1, R)$ , where:

- $O := O_{m,n} \cup \{yes, no, e\} \cup \{d_i^{(k)} \mid i \in [2n+1], k \in [7]\} \cup \{c_j^{(k)} \mid j \in [m], k \in [3]\}$ ;
- $H := \{skin, aux, 1, \dots, 2n+1\} \cup \{c_j \mid j \in [m]\}$ ;
- $\mu := [[ [ ]_1 ]_{aux}]_{skin}$ , where the input membrane is  $[ ]_1$ ;
- $w_{skin} := \varepsilon, w_{aux} := \varepsilon$  and  $w_1 := d_1^{(1)}$ ;
- $R$  is the set of the following rules (in some cases we also give explanations of the presented rules):
  - (a)  $[d_1^{(1)}]_1 \rightarrow [d_2^{(7)}]_2 [d_3^{(7)}]_3$  and  $[d_{2i+k}^{(1)}]_{2i+k} \rightarrow [d_{2i+2}^{(7)}]_{2i+2} [d_{2i+3}^{(7)}]_{2i+3}$ , for every  $i \in [n]$  and  $k \in \{0, 1\}$   
 (these rules are for duplicating and separating into new membranes those objects which encode literals of the input formula);
  - (b)  $[d_i^{(j)} \rightarrow d_i^{(j-1)}]_{2i+k}$ , for every  $2 \leq i \leq n$ ,  $2 \leq j \leq 7$ ,  $k \in \{0, 1\}$   
 (these rules are for counting the steps of the system between two applications of rules in (a));
  - (c)  $[\bar{x}_{j,i} \rightarrow c_j^n]_{2i}$  and  $[x_{j,i} \rightarrow c_j^n]_{2i+1}$ , for every  $i \in [n]$  and  $j \in [m]$ ,  
 $c_j \rightarrow [c_j^{(1)}]_{c_j}$ , for every  $j \in [m]$ ,  
 $[c_j^{(l)} \rightarrow c_j^{(l+1)}]_{c_j}$ , for every  $j \in [m]$  and  $l \in [2]$ ,  
 $x_{j,i} \rightarrow [x_{j,i}]_{c_j}$  and  $\bar{x}_{j,i} \rightarrow [\bar{x}_{j,i}]_{c_j}$ , for every  $i \in [n]$  and  $j \in [m]$ ,  
 $[x_{j,i} \rightarrow \varepsilon]_{c_j}$  and  $[\bar{x}_{j,i} \rightarrow \varepsilon]_{c_j}$ , for every  $i \in [n]$  and  $j \in [m]$ ,  
 $[c_j^{(3)}]_{c_j} \rightarrow c_j^{(3)}$ , for every  $j \in [m]$ ,  
 $[c_j^{(3)} \rightarrow \varepsilon]_{2i+k}$ , for every  $i \in [n]$ ,  $j \in [m]$  and  $k \in \{0, 1\}$   
 (these rules are used to select and erase specific objects in those membranes that are created by the rules in (a); the selection is done by introducing new objects ( $c_j$ ) that can create new membranes ( $[ ]_{c_j}$ ); these new membranes can select the corresponding objects which are then erased by  $\varepsilon$ -rules; finally the used axillary membranes and objects are erased also);
  - (j)  $[x_{j,i}]_{2n+k} \rightarrow e$  and  $[\bar{x}_{j,i}]_{2n+k} \rightarrow e$ , for every  $i \in [n]$  and  $k \in \{0, 1\}$   
 (these rules dissolve those membranes with label  $2n+k$  that contain a literal of a clause of  $\varphi$ ; during the dissolution of each such membrane an object  $e$  is introduced);
  - (k)  $[e]_{2n+k} \rightarrow [yes]_{2n+k}$  and  $[yes]_{2n+k} \rightarrow [ ]_{2n+k} yes$ , for every  $k \in \{0, 1\}$ ,  
 $[yes]_{aux} \rightarrow [ ]_{aux} yes$ ,  
 $[yes]_{skin} \rightarrow [ ]_{skin} yes$   
 (if there is a membrane with label  $2n+k$  which is not dissolved by the rules in (j), then the object  $e$  introduces the object  $yes$ ; the other rules are used to send  $yes$  out to the environment);
  - (l)  $[e]_{aux} \rightarrow [e]_{aux} [no]_{aux}$ ,  
 $[no]_{aux} \rightarrow [ ]_{aux} no$ ,  
 $[no]_{skin} \rightarrow [ ]_{skin} no$   
 (if every membrane with label  $2n+k$  could be dissolved by the rules in (j), then  $e$  is used to duplicate the membrane with label  $aux$  and to introduce the object  $no$ ; the other rules are used to send  $no$  out to the environment).

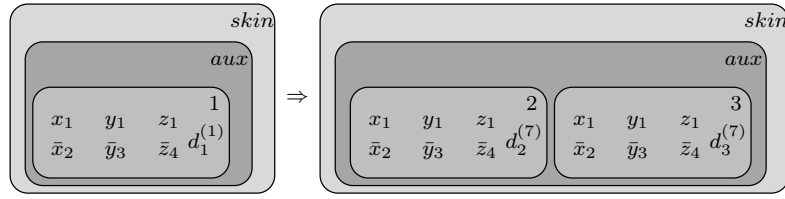
The computation of  $\Pi(\langle m, n \rangle)$ , for some  $m, n \in \mathbb{N}$ , when the membrane with label 1 contains  $\text{cod}(\varphi)$ , for a formula  $\varphi \in \text{Form}_n$  with  $m$  clauses can be described as follows:

- (1) During the first step the system duplicates the membrane with label 1 creating two new membranes with label 2 and 3, respectively (using the first rule in (a)). The object  $d_1^{(1)}$  involved by this rule is replaced in the two new membranes by the objects  $d_2^{(7)}$  and  $d_3^{(7)}$ , respectively. Those objects in membrane with label 1 which encode literals of the input formula are duplicated and distributed between the two new membranes. Thus, after the first step, every clause of the input formula are contained in both of the new membranes.
- (2) During the next six steps, the objects  $d_2^{(7)}$  and  $d_3^{(7)}$  are evolved to  $d_2^{(1)}$  and  $d_3^{(1)}$ , respectively; meanwhile the following happens:
  - (a) in membrane with label 2 (resp. with label 3), for every  $j \in [m]$ ,  $x_{j,1}$  (resp.  $\bar{x}_{j,1}$ ) introduces  $n$  pieces of the object  $c_j$ ;
  - (b) for every  $j \in [m]$ , every object  $c_j$  creates a membrane with label  $c_j$ ; these new membranes contain the object  $c_j^{(1)}$ ;
  - (c) for every  $j \in [m]$ , the objects  $x_{j,1}$  and  $\bar{x}_{j,1}$  are sent to the membranes with label  $c_j$ ; meanwhile the objects  $c_j^{(1)}$  evolve to  $c_j^{(2)}$ ;
  - (d) the objects  $x_{j,1}$  are erased and the objects  $c_j^{(2)}$  evolve to  $c_j^{(3)}$ ;
  - (e) the membranes with label  $c_j$  are dissolved;
  - (f) the objects  $c_j^{(3)}$  are erased.

In this way, those literals that belong to a clause containing  $\bar{x}_1$  are removed from the membrane with label 2. Likewise, those literals that belong to a clause containing  $x_1$  are removed from the membrane with label 3. Thus, those clauses of  $\varphi$  that contain  $x_1$  are placed to membrane 2, those that contain  $\bar{x}_1$  to membrane 3, and those that do not contain  $x_1$  or  $\bar{x}_1$  are placed to both membranes. This corresponds to the operation in line 5 of Algorithm 1.
- (3) After carrying out seven steps similar to the above ones  $n - 1$  times, the membrane system contains  $2^n$  membranes with label  $2n + k$  ( $k \in \{0, 1\}$ ). Each such membrane can contain objects which encode clauses of  $\varphi$ . Then those membranes with label  $2n + k$  that contain at least one object  $x_{j,i}$  ( $i \in [n], j \in [m]$ ) are dissolved by using the rules in (j). During the dissolution of each membrane an object  $e$  is introduced. So far the number of steps of the system is  $7n + 1$ . At this point the computation can continue in two different cases.
- (4) If every membrane with label  $2n + k$  is dissolved, then, using the first rule in (l), the system divides the membrane with label  $\text{aux}$ , and introduces the object  $\text{no}$ . In the last two steps, the object  $\text{no}$  is sent out to the environment, and the computation halts.
- (5) If there is at least one membrane with label  $2n + k$  that is not dissolved, then only the first rule in (k) can be applied, introducing the object  $\text{yes}$  (notice that the division rule in (l) cannot be applied as the membrane with label 2 is not elementary in this case). In the last three steps of the system, the object  $\text{yes}$  is sent out to the environment, and the computation halts.

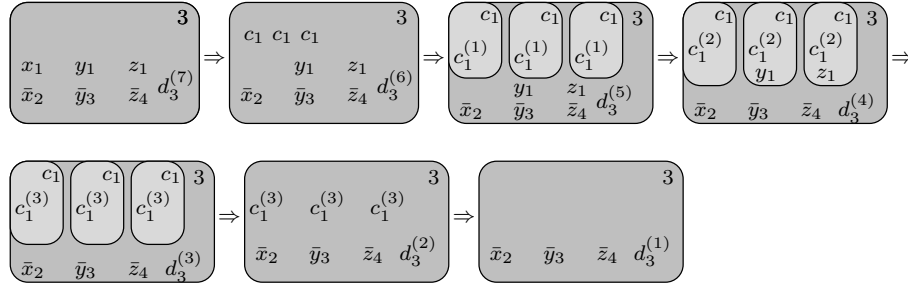
It is easy to see that the system halts after at most  $7n + 5$  steps sending out to the environment either *yes* or *no*. We demonstrate the above described work of our P systems by the following example.

*Example 1.* Let  $\varphi = (x_1 \vee x_2 \vee x_3) \wedge \bar{x}_1 \wedge \bar{x}_2 \wedge \bar{x}_3$ . For the better readability of the computation of  $\Pi(\langle 4, 3 \rangle)$  on this formula, we denote  $x_1, x_2$  and  $x_3$  by  $x, y$  and  $z$ , respectively. Thus,  $\text{cod}(\varphi) = \{x_1, y_1, z_1, \bar{x}_2, \bar{y}_3, \bar{z}_4\}$  (notice that here, for example,  $y_1$  denotes  $x_{1,2}$  meaning that the variable  $x_2$ , which is denoted by  $y$ , occurs in the first clause of  $\varphi$ ). Let  $\Pi(\langle 4, 3 \rangle)$  be the P system constructed in Definition 1 with  $\text{cod}(\varphi)$  in its input membrane. The initial configuration of  $\Pi(\langle 4, 3 \rangle)$  and its configuration after the first step can be seen on Figure 1.



**Fig. 1.** The first step of  $\Pi(\langle 4, 3 \rangle)$ .

Now we consider the next six steps of the system, but we show only the computation carried out in the membrane with label 3. The initial configuration of this membrane and its configuration after each step can be seen in Figure 2.



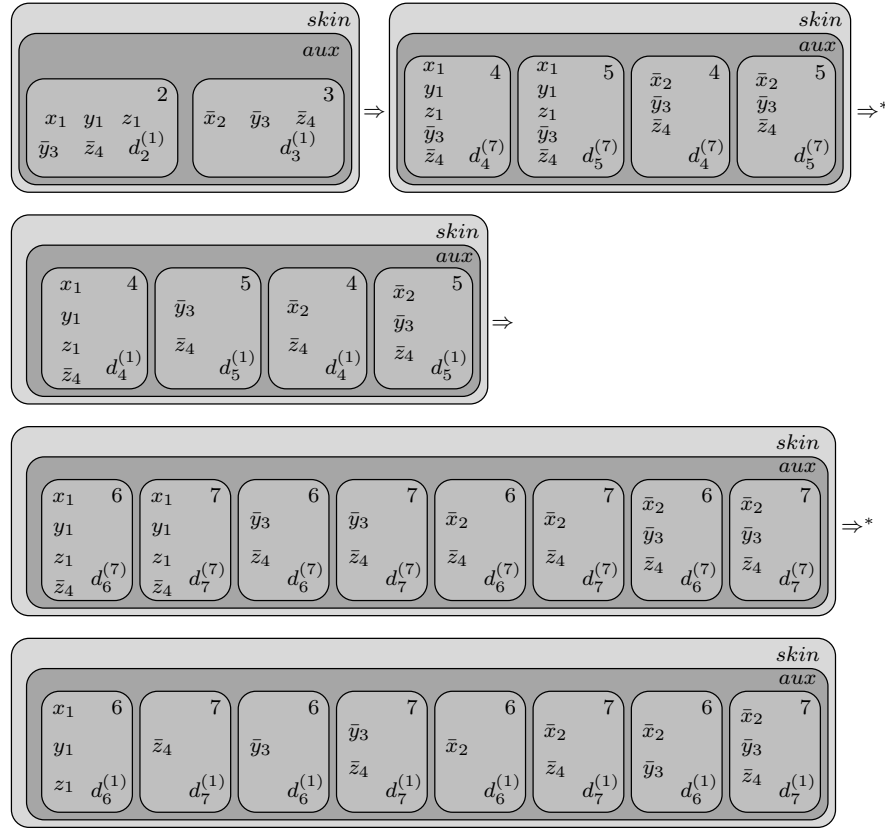
**Fig. 2.** The evolution of the membrane with label 3 using rules in (c).

In the following we shortly describe what is changed in this membrane after these steps:

- (1)  $x_1$  introduces three pieces of the object  $c_1$ ;
- (2) each  $c_1$  creates a membrane  $[ ]_{c_1}$  containing  $c_1^{(1)}$ ;

- (3)  $y_1$  and  $z_1$  are sent to a membrane with label  $c_1$  and the objects  $c_1^{(1)}$  evolve to  $c_1^{(2)}$ ;
- (4)  $y_1$  and  $z_1$  are erased and the objects  $c_1^{(2)}$  evolve to  $c_1^{(3)}$ ;
- (5) the membranes with label  $c_1$  are dissolved and their contents are released in membrane with label 3;
- (6) the objects  $c_1^{(3)}$  are erased.

The first configuration in Figure 3 is the configuration of the system after the first seven steps. The rest of the configurations in this figure describe the evolution of the system during the next fourteen steps.

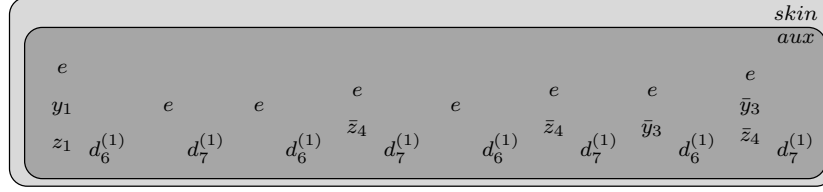


**Fig. 3.** The evolution of  $\Pi(\langle 4, 3 \rangle)$  until the last application of rules in (a)–(c).

Let us consider now the last configuration in Figure 3. Here only rules in (j) can be applied. These rules dissolve those membranes contained in membrane *aux* that have at least one object representing a literal of  $\varphi$ . The objects in-



volved by these dissolution rules are chosen non-deterministically. One possible configuration after the application of these rules can be seen in Figure 4.



**Fig. 4.** The configuration of  $\Pi(\langle 4, 3 \rangle)$  after the application of rules in (j).

The correctness of  $\Pi(\langle m, n \rangle)$  started with a formula  $\varphi \in \text{Form}_n$  containing  $m$  clauses ( $n, m \in \mathbb{N}$ ) follows from the following statement.

**Lemma 1.** *Let  $\varphi \in \text{Form}_n$  containing  $m$  clauses ( $n, m \in \mathbb{N}$ ). Starting  $\Pi(\langle m, n \rangle)$  with  $\varphi$  in its input membrane,  $\Pi(\langle m, n \rangle)$  sends out to the environment yes if and only if Algorithm 1 terminates with yes on input  $\varphi$ .*

*Proof.* First we define a bijection between the membranes with labels in  $[2n+1]$  and the set  $\{u \in \{0,1\}^* \mid |u| \leq n\}$  recursively. Let  $f(1) := \varepsilon$  and consider a membrane  $m$  with label  $2i+k$  ( $i \in [n-1], k \in \{0,1\}$ ). Assume that  $f(m) = u$  and let  $m_1$  and  $m_2$  be those membranes with label  $2(i+1)$  and  $2(i+1)+1$ , respectively, that are created from  $m$ . Then let  $f(m_1) := u0$  and  $f(m_2) := u1$ . Clearly  $f$  is a bijection and it in fact corresponds to a pre-order traversal of these membranes.

For a clause  $C \in \mathcal{C}_n$  and a word  $u \in \{0,1\}^*$  with  $|u| = i$ , we say that  $C$  contains the literals determined by  $u$  if the following holds. For every  $j \in [i]$ ,  $x_j \in C$  if the  $j$ th letter of  $u$  is 1 and  $\bar{x}_j \in C$  otherwise. Let moreover  $\varphi_i$  be the formula  $\hat{\varphi}$  after the  $i$ th iteration of the loop starting at line 1 in Algorithm 1. Finally, we call a membrane *quasi-empty* if it does not contain an object of the form  $x_{j,i}$  or  $\bar{x}_{j,i}$  ( $i \in [n], j \in [m]$ ). Then it is not difficult to see that the following holds:

- $\Pi(\langle m, n \rangle)$  sends out to the environment yes with input  $\text{cod}(\varphi)$  iff
- there is  $i \in [n]$  and a word  $u \in \{0,1\}^*$  with  $|u| = i$  such that there is a quasi-empty membrane  $m$  with  $f(u) = m$  iff
- there is  $i \in [n]$  and a word  $u \in \{0,1\}^*$  with  $|u| = i$  such that  $\varphi_i$  does not contain a clause  $C$  such that  $C$  contains the literals determined by  $u$  iff
- Algorithm 1 terminates with yes on input  $\varphi$ .

With this, we finished the proof of the lemma.

**Theorem 1.** *The SAT can be solved by a family  $\Pi := (\Pi(\langle m, n \rangle))_{m, n \in \mathbb{N}}$  of polarizationless recognizing P systems with the following properties:*

- (1)  $\Pi$  is polynomially uniform;
- (2) the elements of  $\Pi$  are confluent;
- (3) for a formula  $\varphi$  with  $n$  variables and  $m$  clauses, starting  $\Pi(\langle m, n \rangle)$  with  $\text{cod}(\varphi)$  in its input membrane,  $\Pi(\langle m, n \rangle)$  stops in linear number of steps in  $n$ .

*Proof.* The fact that  $\Pi$  solves SAT follows from Lemma 1 and Proposition 2. Property (1) follows from the fact that  $\Pi(\langle m, n \rangle)$  has polynomial size in  $\langle m, n \rangle$ , for every  $n, m \in \mathbb{N}$ . Property (2) follows from the following observation. Our system is non-deterministic only when it dissolves the membranes with label  $2n + k$  (using rules in (j)) and when the objects  $e$  chose membranes with label  $2n$  or  $2n + 1$  (first rules in (k)). Clearly these non-deterministic choices do not affect the output of the system. Property (3) can be seen using the discussion after Definition 1 concerning the computation of  $\Pi(\langle m, n \rangle)$ .

As we have mentioned before, there is a solution of SAT where the used P systems employ membrane creation and the number of computation steps of these systems is bounded by the number of variables of the input formula (see [6] or Section 12.6.1 in [11]). This solution, roughly, works in the following way. For a formula  $\varphi$  with  $n$  variables and  $m$  clauses, the P system first creates in  $2n$  steps  $2^n$  membranes, each of them corresponding to a possible evaluation of the variables in  $X_n$ . Meanwhile, the system stores in every membrane, by using new objects, those clauses of  $\varphi$  that are satisfied by the interpretation represented by the membrane. Finally, the system checks in constant steps, using again membrane creation, whether there is a membrane such that the objects in that membrane represent  $\varphi$  (i.e., the system decides if there is an interpretation that satisfies every clause in  $\varphi$ ). As it is stated in [6], the time complexity of this solution is  $\theta(n)$ . Moreover, it can be seen that the space complexity of this solution (i.e., the maximal number of objects and membranes presented in the system at the same time during its computation) is  $\theta(2^n m) = \theta(6^n)$ .

On the other hand, it can be seen that the space complexity of our solution is  $\theta(n4^n)$ . Indeed, the number of the objects corresponding to the literals of the input formula is bounded by  $4^n$ , and each of these objects can introduce  $n$  new objects (the objects introduced by the first rules in (c)). Moreover, these new objects are erased after a constant number of steps. It is also worth mentioning that the solution of [6] has  $\theta(6^n)$  space complexity whatever the input formula is. On the other hand, the space complexity of our solution can be asymptotically less than  $\theta(n4^n)$  in certain cases. For example, if the input formula has only complete clauses, then our P systems use only  $\theta(n2^n)$  objects and membranes.

In summary, while these solutions have the same time complexity, our solution has asymptotically less space complexity. On the other hand, the solution of [6] does not use membrane separation or membranes division rules, while our solution uses membrane division.

## 4 Conclusions

In this paper we presented a polynomially uniform family of P systems that can decide the satisfiability of a propositional formula in linear time in the number of the variables in the formula. The given P systems use the classical rules of P systems with active membranes and, in addition, membrane creation rules and such elementary membrane division rules which can change the label of the involved membranes.

It seems that the membrane label changing property in our solution, like in case of some existing solutions, can be traded for the use of polarizations. In our solution the membrane label changing is used for the following reason. When  $\Pi(\langle m, n \rangle)$  applies the first rule in (c), then the label of the membrane is used to select the applicable rules: for  $i \in [n]$ , in a membrane with label  $2i$ , only the objects  $\bar{x}_{j,i}$  can be subjects of rules, while in a membrane with label  $2i + 1$ , only the objects  $x_{j,i}$  can be rewritten. If there is no possibility of membrane label changing, we can use polarizations of the membranes to select whether a variable or its negation can be rewritten. To ensure that after the  $i$ th membrane division only the  $i$ th variable or its negation can be rewritten we can do the following. We can use  $n$  copies of the objects in  $O_{m,n}$  and add such rules that at every step when a membrane division happens a new copy of every object in  $O_{m,n}$  is introduced. Moreover, we can define the rules in (c) such that when the  $i$ th copies of the objects are in the membrane then only the literals containing the  $i$ th variable can be subjects of rules.

It is not clear, on the other hand, whether we can get rid of the membrane creation rules in our solution. This might be a subject of a further research.

## 5 Acknowledgements

The author gratefully acknowledges the many helpful suggestions of the anonymous referees.

## References

1. Alhazov, A.: Minimal parallelism and number of membrane polarizations. The Computer Science Journal of Moldova **18**(2), (2010) 149–170
2. Alhazov, A., Pan, L., Paun, G.: Trading polarizations for labels in P systems with active membranes. Acta Inf. **41**(2-3), (2004) 111–144
3. Cecilia, J. M., García, J. M., Guerrero G. D., Martínez-del-Amor, M. A., Pérez-Hurtado, I., Pérez-Jiménez, M. J.: Simulating a P system based efficient solution to SAT by using GPUs. J. Log. Algebr. Program. **79**(6), (2010) 317–325
4. Freund, R., Paun, G., Pérez-Jiménez, M.J.: Polarizationless P Systems with Active Membranes Working in the Minimally Parallel Mode. In: UC. (2007) 62–76
5. Gazdag, Zs., Kolonits, G.: A new approach for solving SAT by P systems with active membranes. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, Gy. (eds.) Membrane Computing - 13th International Conference, LNCS **7762**, (2013) 195–207

6. Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J., Romero-Campero, F.J.: A uniform solution to SAT using membrane creation. *Theor. Comput. Sci.* **371**(1-2), (2007) 54–61
7. Pan, L., Alhazov, A.: Solving HPP and SAT by P Systems with Active Membranes and Separation Rules. *Acta Inf.* **43**(2), (2006) 131–145
8. Paun, G.: Computing with membranes. *J. Comput. Syst. Sci.* **61**(1), (2000) 108–143
9. Paun, G.: P Systems with Active Membranes: Attacking NP-Complete Problems. *Journal of Automata, Languages and Combinatorics* **6**(1), (2001) 75–90
10. Paun, G.: Introduction to membrane computing. In: *Applications of Membrane Computing*, (2006) 1–42
11. Paun, G., Rozenberg, G., Salomaa, A.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA (2010), <http://portal.acm.org/citation.cfm?id=1738939>
12. Pérez-Jiménez, M.J., Jiménez, Á.R., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. *Natural Computing* **2**(3), (2003) 265–285

# Solving Hard Problems in Evolution-Communication P systems with Energy

Nestine Hope S. Hernandez, Richelle Ann B. Juayong, Henry N. Adorna

Algorithms & Complexity Lab  
Department of Computer Science  
University of the Philippines Diliman  
Diliman 1101 Quezon City, Philippines  
E-mail: {nshernandez,rbjuayong,hnadorna}@up.edu.ph

**Abstract.** In this paper, we present non-confluent solutions to some NP-complete problems using recognizer Evolution-Communication P systems with Energy (ECPe systems). We then evaluate the communication resources used in these systems using dynamical communication measures proposed for computations in ECPe systems. Specifically, we evaluate based on number of communication steps, communication rules and energy required for all communication.

**Keywords:** Membrane computing, recognizer P systems, Evolution-Communication P systems with Energy, communication complexity

## 1 Introduction

Evolution-Communication P systems with energy [1] (ECPe systems, for short) is a cell-like variant of P systems which is introduced to investigate communication on a system where communication is dependent on some ‘energy’ generated when objects evolve. In [1], dynamical communication resources are also proposed to evaluate the communication resources of solving problems in ECPe systems.

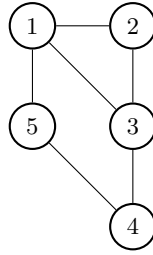
This study continues the works in [1] by examining the resources used in solving decision problems, specifically, Vertex Cover Problem (VCP), Independent Set problem (ISP) and 3-SAT Problem (3SP). We construct recognizer P systems (whose definitions are adapted from [7] and [8]) to non-confluently decide on these problems. We use the dynamical communication measures in [1] to determine the amount of communication steps, rules, and energy employed in solving such problems.

The content of this paper is arranged as follows: Section 2 formally defines the NP-complete problems we investigated, Section 3 discusses the formal definition of ECPe systems and how we can decide on problems using the idea of non-confluence and recognizer P systems. The main contribution of our work is provided in Section 4. Finally, our conclusions are given in Section 5.

## 2 Definitions of some NP-complete problems

We present formal definitions of the three NP-complete problems of interest for our study. Two of these problems use graphs as inputs while the remaining problem involves evaluation of boolean formula.

A graph is denoted by  $G = (V, E)$  where  $V$  is a set of vertices and  $E \subseteq V \times V$  is the set of edges. Note that in this paper, we only consider simple graphs, that is, graphs with no loops and parallel edges. Shown in Figure 1 is an example of a graph where  $V = \{1, 2, \dots, 5\}$  and  $E = \{(1, 2), (1, 3), (1, 5), (2, 3), (3, 4), (4, 5)\}$ . Without loss of generality, it is imposed that each edge in  $E$  is represented by a pair  $(i, j), i < j$ .



**Fig. 1.** An example of a graph

A vertex cover  $VC$  is a set of vertices in  $V$  where for all edge  $(i, j) \in E$ , either  $i \in VC$  or  $j \in VC$ . We denote  $VC_k$  ( $1 \leq k \leq |V|$ ) as a vertex cover with size less than equal to  $k$ . It can be observed that in the graph given in Figure 1, there exists a vertex cover  $VC_3 = \{1, 3, 4\}$ .

**Definition 1. Vertex Cover Problem (VCP)** Given a graph  $G = (V, E)$  and a positive integer  $k$  ( $1 \leq k \leq |V|$ ), is there a vertex cover  $VC_k$ ?

An independent set  $IS$  is a set of vertices where for all pair  $i, j \in IS$ , there is no edge in  $E$  connecting  $i$  and  $j$ . We let  $IS_k$  be an independent set of size at least  $k$ . In Figure 1,  $IS_2 = \{2, 5\}$  is an independent set of size 2. It can be observed that  $IS_2 = V - VC_3$ . This is a consequence of the lemma given in [3] stating that given a graph  $G = (V, E)$  and subset  $V' \subseteq V$ , then  $V'$  is a vertex cover for  $G$  if and only if  $V - V'$  is an independent set of  $G$ .

**Definition 2. Independent Set Problem (ISP)** Given a graph  $G = (V, E)$  and a positive integer  $k$  ( $1 \leq k \leq |V|$ ), is there an independent set  $IS_k$ ?

In boolean logic, a boolean formula in conjunctive normal form (CNF) involving a set of variables  $X$  is a conjunction of a set of propositional clauses where a propositional clause is defined as a disjunction of a set of variables in  $X$  that may take on values 1 (true) or 0 (false). Disjunction in a clause involves performing OR-operations on the variables involved while conjunction involves performing AND-operations on the result of the clause evaluations.

Formally, a formula  $\phi_X$  in CNF over a set of variables  $X = \{x_1, x_2, \dots, x_p\}$  is a conjunction of a set of propositional clauses represented as:

$$\phi_X = C_1 \wedge C_2 \wedge \dots \wedge C_m$$

where  $m \in \mathbb{Z}^+$  and  $C_i$ 's are propositional clauses such that

$$C_i = (y_{i1} \vee y_{i2} \vee \dots \vee y_{in})$$

where  $n \in \mathbb{Z}^+$  and  $y_{ij} \in X \cup \{\bar{x} \mid x \in X\}$ ,  $1 \leq j \leq n$ . The notation  $\bar{x}$  implies a negation so that  $\bar{\bar{x}} = x$ .

We define a  $k$ -CNF boolean formula as a boolean formula in CNF where each clause is a disjunction of exactly  $k$  variables. We say that a boolean formula is satisfiable if there exists an assignment for all variables such that the formula evaluates to true.

**Definition 3. 3-SAT Problem (3SP)** *Given a 3-CNF boolean formula  $\phi$  over a set of variables  $X$ , is  $\phi$  satisfiable?*

Let a 3-CNF boolean formula  $\phi_x = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$  where  $X = \{x_1, x_2, x_3, x_4\}$ . We say that  $\phi$  is satisfiable since the formula evaluates to true when  $x_1 = 0, x_2 = 0, x_3 = r, x_4 = r, r \in \{0, 1\}$ .

### 3 ECPe systems

Before we proceed, the readers are assumed to be familiar with the fundamentals of formal language theory and membrane computing [6].

A new variant of Evolution-Communication P systems [2] has been introduced in [1] to evaluate communication that is dependent on some energy produced from evolution rules. A special object  $e$  is introduced to the system to represent a quantum of energy. We use the definition for EC P system with energy (ECPe system) from [1].

**Definition 4.** *An EC P system with energy is a construct of the form*

$$\Pi = (O, e, \mu, w_1, \dots, w_m, R_1, R'_1, \dots, R_m, R'_m, i_{out})$$

where:

- (i)  $m$  pertains to the total number of membranes;
- (ii)  $O$  is the alphabet of objects;
- (iii)  $\mu$  is the membrane structure which can be denoted by a set of paired square brackets with labels. We say that membrane  $i$  is the *parent membrane* of a membrane  $j$ , denoted  $parent(j)$ , if the paired square brackets representing membrane  $j$  is located inside the paired square brackets representing membrane  $i$ , i.e.  $[i \dots [j \dots ]j]i$ . Reversely, we say that membrane  $j$  is a *child membrane* of membrane  $i$ , denoted  $j \in children(i)$  where  $children(i)$  refers to the set of membranes contained in membrane  $i$ . The relation of parent

and child membrane becomes more apparent when we represent the membrane structure as a tree. Since order does not matter in our model, there can be multiple trees (isomorphic with respect to children of a node), each corresponding to the same membrane structure representation.

- (iv)  $w_1, \dots, w_m$  are strings over  $O^*$  where  $w_i$  denotes the multiset of object present in the region bounded by membrane  $i$ .
- (v)  $R_1, \dots, R_m$  are sets of evolution rules, each associated with a region delimited by a membrane in  $\mu$ ;
  - An evolution rule is of the form  $a \rightarrow v$  where  $a \in O$ ,  $v \in (O \cup \{e\})^*$ . In the event that this type of rule is applied, the object  $a$  transforms into a multiset of objects  $v$  in the next time step. Through evolution rules, object  $e$  can be produced, but  $e$  should never be in the initial configuration and object  $e$  is not allowed to evolve.
- (vi)  $R'_1, \dots, R'_m$  are sets of communication rules, each associated with a membrane in  $\mu$ ; A communication rule can either be a symport or an antiport rule:
  - A symport rule can be of the form  $(ae^i, in)$  or  $(ae^i, out)$ , where  $a \in O$ ,  $i \geq 1$ . By using this rule,  $i$  copies of object  $e$  are consumed to transport object  $a$  inside (denoted by *in*) or outside (denoted by *out*) the membrane where the rule is defined. To consume copies of object  $e$  means that upon completion of the transportation of object  $a$ , the occurrences of  $e$  are lost, they do not pass from a region to another one.
  - An antiport rule is of the form  $(ae^i, out; be^j, in)$  where  $a, b \in O$  and  $i, j \geq 1$ . By using this rule, we know that there exists an object  $a$  in the region immediately outside the membrane where the rule is declared, and an object  $b$  inside the region bounded by the membrane. In the application of this rule, object  $a$  and object  $b$  are swapped using  $i$  and  $j$  copies of object  $e$  in the different regions, respectively. As in symport rules, the copies of object  $e$  are lost after the application.

We say that a communication rule has a *sending* and *receiving* region. For a rule  $r \in R'_i$  associated with an *in* label, its receiving region is region  $i$  and its sending region is the *parent*( $i$ ). The sending and receiving regions are reversed for a rule  $r \in R'_i$  associated with an *out* label. For an antiport rule  $r \in R'_i$ , region  $i$  and *parent*( $i$ ) are both sending and receiving region. Also, note that no communication can be applied without the utilization of object  $e$ .

- (vii)  $i_{out} \in \{0, 1, \dots, m\}$  is the output membrane. If  $i_{out} = 0$ , this means that the environment shall be the placeholder of the output.

Rules are applied in a nondeterministic, maximally parallel manner. Nondeterminism, in this case, has the following meaning: when there are more than two evolution rules that can be applied to an object, the system will randomly choose the rule to be applied for each copy of the object. The system assumes a universal clock for simultaneous processing of membranes; all applicable rules have to be applied to all possible objects at the same time. The behavior of maximally parallel application of rule requires that all object that can evolve (or be transferred) should evolve (or be transferred).



Note that there is a one-to-one mapping between region and membrane, however, strictly, region refers to the area delimited by a membrane. A configuration at any time  $i$ , denoted by  $C_i$ , is the state of the system; it consists of the membrane structure and the multiset of objects within each membrane. A transition from  $C_i$  to  $C_{i+1}$  through nondeterministic and maximally parallel manner of rule application can be denoted as  $C_i \Rightarrow C_{i+1}$ . A series of transition is said to be a computation and can be denoted as  $C_i \Rightarrow^* C_j$  where  $i < j$ . Computation succeeds when the system halts; this occurs when the system reaches a configuration wherein none of the rules can be applied. This configuration is called a halting configuration. If there is no halting configuration—that is, if the system does not halt—computation fails, because the system did not produce any output. Output can either be in the form of objects sent outside the skin, the outermost membrane, or objects sent into the output membrane.

### 3.1 Dynamical communication complexity measures for ECPe systems

Based on [1], the dynamical communication complexity parameters associated with a given computation for ECPe systems are:

$$ComN(C_i \Rightarrow C_{i+1}) = \begin{cases} 1 & \text{if at least a communication} \\ & \text{rule is used in this} \\ & \text{transition,} \\ 0 & \text{otherwise} \end{cases}$$

$$ComR(C_i \Rightarrow C_{i+1}) = \text{the number of communication rules used in this transition,}$$

$$ComW(C_i \Rightarrow C_{i+1}) = \text{the total energy of the communication rules used in this transition.}$$

These parameters are related in that  $ComN \leq ComR \leq ComW$ . They can be extended in a natural way to results of computations, systems, and sets of numbers. Again, we adapt the next definition from [1].

**Definition 5.** We let  $N(\Pi)$  be the set of numbers computed by the system. For  $ComX \in \{ComN, ComR, ComW\}$ , the following is defined:

$$\begin{aligned}
 ComX(\delta) &= \sum_{i=0}^{h-1} ComX(C_i \Rightarrow C_{i+1}), \\
 &\text{for } \delta : C_0 \Rightarrow C_1 \Rightarrow \dots \Rightarrow C_h \\
 &\text{is a halting computation,} \\
 ComX(n, \Pi) &= \min\{ComX(\delta) \mid \\
 &\delta : C_0 \Rightarrow C_1 \Rightarrow \dots \Rightarrow C_h \\
 &\text{in } \Pi \text{ with the result } n\}, \\
 ComX(\Pi) &= \max\{ComX(n, \Pi) \mid n \in N(\Pi)\}, \\
 ComX(Q) &= \min\{ComX(\Pi) \mid Q = N(\Pi)\}.
 \end{aligned}$$

### 3.2 Solving problems in ECPe systems

When solving problems in P systems, [7] uses the notion of a recognizer P system. For our definition of recognizer ECPe systems, we use the definition from [8].

**Definition 6.** Let  $\Pi$  be an ECPe system whose alphabet contains two distinct objects **yes** and **no**, such that every computation of  $\Pi$  is halting and during each computation, exactly one of the objects **yes**, **no** is sent out from the skin to signal acceptance or rejection. If all the computations of  $\Pi$  agree on the result, then  $\Pi$  is said to be confluent; if this is not necessarily the case, then it is said to be non-confluent and the global result is acceptance if and only if there exists an accepting computation.

From [7], we can formally represent a decision problem as a pair  $Y = (I_Y, \theta_Y)$  where  $I_Y$  is a language over a finite alphabet and  $\theta_Y$  is a total boolean function over  $I_Y$ . A representation of an instance of a decision problem in P systems is given by a pair  $(cod, s)$  where  $s \in \mathbb{N}$  and  $cod$  refers to an encoding of the instance which will be placed in an input membrane in the initial configuration.

Our notion of a P system solving a problem is adapted from definitions in both [7] and [8] where a problem is solved using a family of P systems. A family  $\Pi(n)$ ,  $n \in \mathbb{Z}^+$ , of P systems (specifically ECPe systems in our context) is a set of P systems that takes a parameter  $n$  to construct each system.

**Definition 7.** A family  $\Pi(n)$ ,  $n \in \mathbb{Z}^+$ , of ECPe systems, solves a problem  $(I_Y, \theta_Y)$  if there exists a pair  $(cod, s)$  over  $I_Y$  such that for each instance  $u \in I_Y$ :

- (i)  $n = s(u) \in \mathbb{N}$  and  $cod(u)$  is an input multiset of the system  $\Pi(n)$ ;
- (ii) there exists an accepting computation of  $\Pi(n)$  with input  $cod(u)$  if and only if  $\theta_Y(u) = 1$ .

The following definitions use dynamical communication measures given in Section 3.1 to analyze communication over ECPe systems solving problems.

**Definition 8.** Let  $Y = (I_Y, \theta_Y)$  be a decision problem,  $\Pi(n)$ ,  $n \in \mathbb{Z}^+$ , be a family of recognizer ECPe systems solving  $Y$  with a pair  $(\text{cod}, s)$  over  $I_X$ . For each instance  $u \in I_Y$ ,

$$\text{Com}X(u, \Pi(n)) = \min\{\text{Com}X(\delta) \mid \delta : C_0 \implies C_1 \implies \dots \implies C_h \text{ in } \Pi(n) \\ \text{with } n = s(u) \text{ and } \text{cod}(u) \text{ is an input multiset in } \Pi(n)\},$$

where  $\text{Com}X \in \{\text{Com}N, \text{Com}R, \text{Com}W\}$ . To analyze the communication resources used by  $\Pi(n)$  in solving problem  $Y$ ,  $\text{Com}X(Y, \Pi(n))$  is defined as:

$$\text{Com}X(Y, \Pi(n)) = \max\{\text{Com}X(u, \Pi(n)) \mid u \in I_Y\}.$$

**Definition 9.** Let  $F\text{Com}X \in \{F\text{Com}N, F\text{Com}R, F\text{Com}W\}$ . A decision problem  $Y = (I_Y, \theta_Y) \in F\text{Com}X(k)$  if and only if:

- (i) There exists a family  $\Pi(n)$ ,  $n \in \mathbb{Z}^+$ , of confluent recognizer ECPe systems that decides  $Y$ .
- (ii)  $\text{Com}X(Y, \Pi(n)) = k$ .

The analogous complexity classes for non-confluent recognizer ECPe systems are  $NF\text{Com}N$ ,  $NF\text{Com}R$ , and  $NF\text{Com}W$ .

We say that  $Y \in F\text{Com}NRW(p, q, r)$  if and only if  $Y \in F\text{Com}N(p)$ ,  $Y \in F\text{Com}R(q)$  and  $Y \in F\text{Com}W(r)$ . We use  $NF\text{Com}NRW$  for non-confluent recognizer ECPe systems.

We note here that the definition of  $F\text{Com}X$  in the previous definition is slightly modified from its definition in [1].

## 4 ECPe system solutions to NP-hard Problems

In this section, we shall present solutions to three NP-hard problems, namely, the vertex cover problem, the independent set problem and the 3-SAT problem.

Let the Vertex Cover Problem ( $VCP$ ) be represented by a pair  $VCP = (I_{VCP}, \theta_{VCP})$  where  $I_{VCP} = \{w_{(G,k)} \mid w_{(G,k)} \text{ is a string representing a graph } G \text{ and a positive integer } k\}$ . If the graph  $G$  contains a vertex cover of size at most  $k$ ,  $\theta_{VCP}(w_{(G,k)}) = 1$ ; otherwise,  $\theta_{VCP}(w_{(G,k)}) = 0$ .

**Theorem 1.**  $VCP \in NF\text{Com}NRW(6, |V_G| + 3k + 6, 3|E_G| + |V_G| + k + 5)$  where  $E_G$  is the edge set and  $V_G$  is the vertex set of the input graph  $G$ .

*Proof.* To prove our claim, we need to satisfy the requirements in Definition 9. To do this, we first introduce a family (denoted by  $\Pi_{VCP}(n)$ ) of ECPe systems for  $VCP = (I_{VCP}, \theta_{VCP})$ . We then show that  $\text{Com}N(VCP, \Pi_{VCP}(n)) = 6$ ,  $\text{Com}R(VCP, \Pi_{VCP}(n)) = |V_G| + 3k + 6$ , and  $\text{Com}W(VCP, \Pi_{VCP}(n)) = 3|E_G| + |V_G| + k + 5$ .

The first part of our proof provides a formal definition of  $\Pi_{VCP}(n)$ . We also define a pair  $(\text{cod}, s)$  over  $I_{VCP}$  and show that for each instance of  $I_{VCP}$ , the

two conditions given in Definition 7 are satisfied. Our family of ECPe systems for VCP is defined as a tuple  $\Pi_{VCP}(n)$ :

$$\Pi_{VCP}(n) = (O, [0[1]_1[2]_2[3]_3]_0, w_0, \emptyset, \emptyset, R_0, R'_0, R_1, R'_1, R_2, R'_2, R_3, R'_3)$$

where:

- $O = \{A_{ij}, v_i, \widehat{v}_i, \widehat{i}, \widehat{i}, \widehat{i} \mid 1 \leq i < j \leq n\} \cup \{c, c', d, d', \#_0, \#_1, \#_2, \#_3, \#_4, \#_5\} \cup \{\alpha_0, \alpha_1, \beta_0, \beta_1, \beta_2\}$
- $w_0 = v_1 v_2 \dots v_n \text{ cod}(w_{(G,k)}) \#_0$
- $R_0 = \{A_{ij} \rightarrow ie, A_{ij} \rightarrow je \mid 1 \leq i < j \leq n\} \cup \{v_i \rightarrow \widehat{v}_i e \mid 1 \leq i \leq n\} \cup \{\#_0 \rightarrow \#_1, \#_1 \rightarrow \#_2, \#_2 \rightarrow \#_3, \#_3 \rightarrow \#_4, \#_4 \rightarrow \#_5 \alpha_0 \beta_0 e^3\} \cup \{c \rightarrow c' e^2, d \rightarrow d' e\} \cup \{\beta_2 \rightarrow \text{yes} e, \alpha_1 \rightarrow \text{no} e\}$
- $R'_0 = \{(\text{no } e, \text{out}), (\text{yes } e, \text{out})\}$
- $R_1 = \{\widehat{v}_i \rightarrow \widehat{i} \mid 1 \leq i \leq n\} \cup \{c' \rightarrow e\}$
- $R'_1 = \{(\widehat{v}_i e, \text{in}), (ie, \text{in}; ie, \text{out}) \mid 1 \leq i \leq n\} \cup \{(c' e, \text{in})\}$
- $R_2 = \{d' \rightarrow e\} \cup \{\widehat{i} \rightarrow \widehat{i}^{n-2} \mid 1 \leq i < j \leq n\} \cup \{\alpha_0 \rightarrow \alpha_1\}$
- $R'_2 = \{(\widehat{ie}, \text{in}), (ie, \text{in}; ie, \text{out}) \mid 1 \leq i \leq n\} \cup \{(d' e, \text{in}), (\alpha_0 e, \text{in})\} \cup \{(\#_5 e, \text{in}; \alpha_1 e, \text{out})\}$
- $R_3 = \{\beta_0 \rightarrow \beta_1, \beta_1 \rightarrow \beta_2 e\}$
- $R'_3 = \{\beta_0 e, \text{in}), (\#_5 e, \text{in}; \beta_2 e, \text{out})\}$

We associate a pair  $(\text{cod}, s)$  over  $I_{VCP}$  such that for a given instance  $w_{(G,k)} \in I_{VCP}$  we have  $n = s(w_{(G,k)}) = |V_G|$  and the encoding  $\text{cod}(w_{(G,k)})$  is a multiset containing  $A_{ij}$  for every  $(i, j) \in E_G$ ,  $k$  copies of object  $c$  and  $|E_G| - k$  copies of object  $d$ . As shown in the construct  $\Pi_{VCP}(n)$ , the encoding is placed as part of the input in membrane 0. This guarantees that  $s(u)$  is a natural number and  $\text{cod}(u)$  is an input multiset for  $\Pi_{VCP}(n)$ , thus, satisfying condition (i) of Definition 7.

In order to show that condition (ii) of Definition 7 is satisfied, we discuss the system's computation:

*Setup phase* In this phase, for each edge in input region 0, an endpoint is nondeterministically chosen to cover that edge and represent that edge in the region. Also, representations of all the vertices are produced in region 1, along with an amount of energy equal to the maximum size of the vertex cover.

Initially, objects  $v_i$  ( $1 \leq i \leq n$ ),  $c$  and  $d$  evolves to  $\widehat{v}_i$ ,  $c'$ , and  $d'$ , respectively (through rules  $v_i \rightarrow \widehat{v}_i e$ ,  $c \rightarrow c' e^2$ ,  $d \rightarrow d' e$ ). At the same time, objects  $A_{ij}$  nondeterministically evolves to one of  $i$  and  $j$  through any of rules  $A_{ij} \rightarrow ie$  and  $A_{ij} \rightarrow je$ . The value  $i$  or  $j$  represents the vertex that is chosen to cover the edge  $(i, j) \in E$ .

In the next step, the single quanta of energy produced in the production of objects  $\widehat{v}_i$  ( $1 \leq i \leq n$ ),  $c'$ , and  $d'$  will be used to communicate the  $\widehat{v}_i$  and  $c'$  in region 1, and the  $d'$  in region 2. The third step involves evolution of communicated objects in region 1 and 2. Specifically,  $c'$  will evolve to object  $e$  and  $\widehat{v}_i$  becomes  $\widehat{i}$  (through rules  $c' \rightarrow e$  and  $\widehat{v}_i \rightarrow \widehat{i}$  ( $1 \leq i \leq n$ )) in region 1 while  $d'$  changes to  $e$  in region 2. Also, in region 0,  $\#_{m-1}$  evolves to  $\#_m$  in step  $m$  for  $m = 1, 2, 3$ .

*Finding a candidate solution.* In region 1, vertices to form a candidate vertex cover are selected and communicated to region 0.

The next step involves swapping the object  $\hat{i}$  in region 1 with its counterpart  $i$  in region 0 through rule  $(ie, in; \hat{ie}, out)$  in membrane 1. In this case, at most one copy of an object  $i$  will be placed in region 1. The set of all objects  $i$  that is transported in region 1 represents the *candidate vertex cover* chosen by a computation. Note that the size of the vertex cover is at most  $k$ . This size is assured by the limited number of  $e$ 's in region 1 that will be used for the transportation. Also note that for each  $i$  representing a vertex in the candidate vertex cover, there is now a corresponding  $\hat{i}$  in region 0. Moreover, the second quanta of energy produced in region 0 in the production of  $c'$  is now utilized in the selection of a candidate vertex cover. During this selection, at most  $k$  of the edges are already verified to be covered by the vertices in the chosen set. Also, in region 0,  $\#_3$  evolves to  $\#_4$  in this step.

*Validating candidate solution.* Representation of the vertices in the candidate solution are produced in region 2. These objects are used to validate that all edges are covered by the selected vertex cover. This is true if no representation of the edges is retained in region 0.

In the next step, the  $\hat{i}$  in region 0 is transported to region 2 (through rule  $(\hat{ie}, in)$ ) to signal that the vertex represented by  $i$  is in the candidate vertex cover. At the same time,  $\#_4$  evolves to  $\#_5\alpha_0\beta_0e^3$  in region 0. In the succeeding step, the objects  $\alpha_0$  and  $\beta_0$  are communicated to regions 2 and 3, respectively, using up two of the quanta of energy, while the  $\hat{i}$  in region 2 will produce  $|V|-2$  copies of  $\hat{i}$ . The  $\hat{i}$  will then be used to determine if the vertices chosen to cover the remaining  $(|E|-k)$  unverified edges are present in the candidate. This ascertaining is done by applying the rule  $(ie, in; ie, out)$ . Note that the maximum degree of simple graph is  $|V|-1$ . Since one incident edge for each vertex in the candidate vertex cover has been verified in the previous phase, then a maximum of  $(|V|-1)-1 = |V|-2$  edges that may be incident to a vertex still remain to be verified. Also, during this step,  $\alpha_0$  and  $\beta_0$  evolve to  $\alpha_1$  and  $\beta_1$  respectively.

*Output phase.* In this phase, object *yes* is released to the environment if a valid vertex cover is selected. Otherwise, object *no* will be sent out.

Note that region 2 started with  $|E|-k$  quanta of energy which equals the number of unverified edges at the start of the previous phase. Hence, in the case where not all of the vertices chosen (nondeterministically) to cover the edges in the setup phase belong to the candidate vertex cover, at least one  $e$  will be left in region 2. This case will allow the object  $\alpha_1$  to be sent out to region 0 and the object  $\#_5$  to enter region 2 through the rule  $(\#_5e, in; \alpha_1e, out)$ . If the candidate solution is indeed a vertex cover, then no  $e$  is left in region 2 not allowing the rule  $(\#_5e, in; \alpha_1e, out)$  to be used. Now, at this same step, object  $\beta_1$  evolves to  $\beta_2e$ . Hence, at the next step, the presence of object  $\#_5$  in region 0 allows the rule  $(\#_5e, in; \beta_2e, out)$  to be used and  $\beta_2$  exits to region 0 while  $\#_5$  enters region 3.

Finally, note that only one of the objects  $\alpha_1$  and  $\beta_2$  will be communicated to region 0 by a computation. If  $\alpha_1$  is in region 0, it evolves to **no**  $e$  and **no** is then released to the environment. This case signals that the computation failed to produce the desired vertex cover. Whereas, if  $\beta_2$  is in region 0, it evolves to **yes**  $e$  and **yes** is subsequently released to the environment. This case signals that the computation succeeded in finding a vertex cover with size at most  $k$  of the input graph  $G$ .

To complete our proof, we analyze the communication resources at each stage of the computation.

- The setup phase discussed previously takes three transitions. In this phase, the system communicates  $c'$ ,  $d'$  and  $\widehat{v}_i$  ( $1 \leq i \leq |V_G|$ ) from membrane 0 in exactly one communication step. Thus:
  - The number of communication steps to accomplish this phase is one.
  - The number of communication rules applied is  $2 + |V_G|$
  - The number of communicated objects is  $|E_G| + |V_G|$ , i.e.  $(|E_G| - k)$  number of  $d'$ ,  $k$  number of  $c'$  and  $|V_G|$  number of  $\widehat{v}_i$ .
- Finding a candidate solution requires at least one communication step. In this phase, there will be one antiport rule for every member of the candidate solution. Thus:
  - The number of communication steps to accomplish this phase is one.
  - The maximum number of communication rules occurs when the size of the candidate solution is  $k$ . In this case, the number of communication rules applied is  $k$ .
  - Following the previous item, the maximum number of communicated objects is  $2k$  since for every antiport rule, two objects are being communicated at the same time.
- For the validation and output phase:
  - The first communication step is used to initially place the  $\widehat{i}$  representing the vertices of the candidate solution in membrane 2. This communication step is also necessary for the initial steps of the output phase. For validating whether all remaining edges are covered, another communication step is needed. Finally, two additional communication steps will be used to produce a **yes** or a **no** and send to the environment.
  - In the first communication step, a maximum of  $k$  rules of the form  $(\widehat{ie}, in)$  ( $1 \leq i \leq |V|$ ) will be used. Simultaneously, rule  $(\alpha_0 e, in) \in R'_2$  and  $(\beta_0 e, in) \in R'_3$  will be used. In the succeeding communication step, a maximum of  $k$  rules of the form  $(ie, in; ie, out)$  will be used to validate the  $|E_G| - k$  remaining edges. This case occurs when the size of the candidate solution is exactly  $k$ . The next communication steps involves the used of either (a)  $(\#_5 e, in; \alpha_1 e, out)$  and  $(\text{no } e; out)$ , or (b)  $(\#_5 e, in; \beta_2 e, out)$  and  $(\text{yes } e; out)$ . Thus, the maximum number of communication rules applied will be  $k + k + 2 + 2$ .
  - Following the previous item, the maximum number of communicated objects will be  $k + 2 + 2(|E_G| - k) + 3$ .

From our discussion above, it can be observed that the path with the most expensive communication resource (steps, rules and objects) is achieved when the candidate solution examined is of size equal to  $k$  and when this candidate is evaluated to be true. Summing the communication resources at each phase, we get  $ComN(VCP, \Pi_{VCP}(n)) = 6$ ,  $ComR(VCP, \Pi_{VCP}(n)) = |V_G| + 3k + 6$ , and  $ComW(VCP, \Pi_{VCP}(n)) = 3|E_G| + |V_G| + k + 5$ .

*An Example for VCP* Given an instance represented in Figure 1 with  $k = 3$ , an ECPe system solving VCP is a construct:

$$\Pi_{VCP}(5) = (O, [0]_1[1]_2[2]_3[3]_0, w_0, \emptyset, \emptyset, R_0, R'_0, R_1, R'_1, R_2, R'_2, R_3, R'_3)$$

where:

- $O = \{A_{ij}, v_i, \hat{v}_i, \hat{i}, \hat{j} \mid 1 \leq i < j \leq 5\} \cup \{c, c', d, d', \#_0, \#_1, \#_2, \#_3, \#_4, \#_5\} \cup \{\alpha_0, \alpha_1, \beta_0, \beta_1, \beta_2\}$
- $w_0 = v_1 v_2 v_3 v_4 v_5 A_{12} A_{13} A_{15} A_{23} A_{34} A_{45} c^3 d^3 \#_0$
- $R_0 = \{A_{ij} \rightarrow ie, A_{ij} \rightarrow je \mid 1 \leq i < j \leq 5\} \cup \{v_i \rightarrow \hat{v}_i e \mid 1 \leq i \leq 5\} \cup \{\#_0 \rightarrow \#_1, \#_1 \rightarrow \#_2, \#_2 \rightarrow \#_3, \#_3 \rightarrow \#_4, \#_4 \rightarrow \#_5 \alpha_0 \beta_0 e^3\} \cup \{c \rightarrow c' e^2, d \rightarrow d' e\} \cup \beta_2 \rightarrow \text{yes } e, \alpha_1 \rightarrow \text{no } e\}$
- $R'_0 = \{(\text{no } e, \text{out}), (\text{yes } e, \text{out})\}$
- $R_1 = \{\hat{v}_i \rightarrow \hat{i} \mid 1 \leq i \leq 5\} \cup \{c' \rightarrow e\}$
- $R'_1 = \{(\hat{v}_i e, in), (ie, in; \hat{i} e, out) \mid 1 \leq i \leq 5\} \cup \{(c' e, in)\}$
- $R_2 = \{d' \rightarrow e\} \cup \{\hat{i} \rightarrow \hat{i}^3 \mid 1 \leq i < j \leq 5\} \cup \{\alpha_0 \rightarrow \alpha_1\}$
- $R'_2 = \{(\hat{i} e, in), (ie, in; \hat{i} e, out) \mid 1 \leq i \leq 5\} \cup \{(d' e, in), (\alpha_0 e, in)\} \cup \{(\#_5 e, in; \alpha_1 e, out)\}$
- $R_3 = \{\beta_0 \rightarrow \beta_1, \beta_1 \rightarrow \beta_2 e\}$
- $R'_3 = \{\beta_0 e, in, (\#_5 e, in; \beta_2 e, out)\}$

Below is an example of a computation for  $\Pi_{VCP}(5)$ , represented as a series of configurations ( $C_i$ ) ( $0 \leq i \leq 11$ ):

$C_0$ :  $[0 \ v_1 v_2 v_3 v_4 v_5 \ A_{12} A_{13} A_{15} A_{23} A_{34} A_{45} \ c^3 \ d^3 \ \#_0 \ [1]_1 [1]_2 [2]_3 [3]_0]$   
 $C_1$ :  $[0 \ \hat{v}_1 e \ \hat{v}_2 e \ \hat{v}_3 e \ \hat{v}_4 e \ \hat{v}_5 e \ 1e \ 3e \ 1e \ 3e \ 3e \ 4e \ c'^3 e^6 \ d'^3 e^3 \ \#_1 \ [1]_1 [1]_2 [2]_3 [3]_0]$   
 $C_2$ :  $[0 \ 1 \ 3 \ 1 \ 3 \ 3 \ 4 \ e^9 \ \#_2 \ [1 \ \hat{v}_1 \hat{v}_2 \hat{v}_3 \hat{v}_4 \hat{v}_5 \ c'^3]_1 [2 \ d'^3]_2 [3]_3]_0]$   
 $C_3$ :  $[0 \ 1 \ 3 \ 1 \ 3 \ 3 \ 4 \ e^9 \ \#_3 \ [1 \ \hat{1} \ \hat{2} \ \hat{3} \ \hat{4} \ \hat{5} \ e^3]_1 [2 \ e^3]_2 [3]_3]_0]$   
 $C_4$ :  $[0 \ \hat{1} \ \hat{3} \ 1 \ 3 \ 3 \ 4 \ e^6 \ \#_4 \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ e^3]_2 [3]_3]_0]$   
 $C_5$ :  $[0 \ 1 \ 3 \ 3 \ e^3 \ \#_5 \alpha_0 \beta_0 e^3 \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ \hat{1} \ \hat{3} \ \hat{4} \ e^3]_2 [3]_3]_0]$   
 $C_6$ :  $[0 \ 1 \ 3 \ 3 \ e^3 \ \#_5 e \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1^3 \ 3^3 \ 4^3 \ e^3 \ \alpha_0]_2 [3 \beta_0]_3]_0]$   
 $C_7$ :  $[0 \ 1 \ 3 \ 3 \ \#_5 e \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1 \ 3 \ 3 \ 1^2 \ 3 \ 4^3 \ \alpha_1]_2 [3 \ \beta_1]_3]_0]$   
 $C_8$ :  $[0 \ 1 \ 3 \ 3 \ \#_5 e \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1 \ 3 \ 3 \ 1^2 \ 3 \ 4^3 \ \alpha_1]_2 [3 \ \beta_2 e]_3]_0]$   
 $C_9$ :  $[0 \ 1 \ 3 \ 3 \ \beta_2 \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1 \ 3 \ 3 \ 1^2 \ 3 \ 4^3 \ \alpha_1]_2 [3 \ \#_5]_3]_0]$   
 $C_{10}$ :  $[0 \ 1 \ 3 \ 3 \ \text{yes} \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1 \ 3 \ 3 \ 1^2 \ 3 \ 4^3 \ \alpha_1]_2 [3 \ \#_5]_3]_0]$   
 $C_{11}$ :  $\text{yes} \ [0 \ 1 \ 3 \ 3 \ [1 \ 1 \ \hat{2} \ 3 \ 4 \ \hat{5}]_1 [2 \ 1 \ 3 \ 3 \ 1^2 \ 3 \ 4^3 \ \alpha_1]_2 [3 \ \#_5]_3]_0]$

Configurations  $C_0$  to  $C_3$  represents the set-up phase where necessary objects are placed in their respective regions for the succeeding phases. At the same time,

transition  $C_0 \Rightarrow C_1$  makes use of rules in  $\{A_{ij} \rightarrow ie, A_{ij} \rightarrow je \mid 1 \leq i, j \leq 5\}$  to choose the vertex that covers the edge represented by object  $A_{i,j}$ . Transition  $C_3 \Rightarrow C_4$  represents the phase where a candidate vertex cover is chosen; in the example computation, the candidate vertex cover is  $VC_3 = \{1, 3, 4\}$  as represented by objects 1, 3 and 4 in region 2. Computation  $C_4 \Rightarrow^* C_7$  represents the verification phase to assure that selected vertex used to cover an edge belongs to the candidate vertex cover. Lastly, the computation  $C_8 \Rightarrow^* C_{11}$  represents the output phase. Since all  $es$  in region 2 where used up, then  $\alpha_1$  will not exit the region and  $\#5$  stays in region 0. This allows  $\beta_2$  to exit region 3, resulting to the object **yes** to be communicated to the environment. This means that the computation succeeded in finding a vertex cover with size at most  $k$  of the input graph  $G$ .

If a different transition  $C_0 \Rightarrow C'_1$  is introduced where

$$C'_1: [{}_0 \hat{v}_1 e \hat{v}_2 e \hat{v}_3 e \hat{v}_4 e \hat{v}_5 e \text{ 1e 3e 1e 2e 4e 5e } c'^3 e^6 \text{ } d'^3 e^3 \text{ } \#_1 [{}_1]_1 [{}_2]_2 [{}_3]_3]_0$$

then the set of vertices chosen to cover the edges of the input graph is  $\{1, 2, 3, 4, 5\}$ . Since only 3 quanta of energy is present in region 2, the use of rule  $(ie, in; i'e, out)$  ( $i \in \{1, 2, 3, 4, 5\}$ ) in region 1 is limited to 3 applications. This case implies that two of the objects 1, 2, 3, 4 and 5 will remain in region 0. Hence, the corresponding  $\hat{i}$  for these remaining object will never exit region 1 and thus cannot be communicated to region 2 through rule  $(ie, in)$ . This scenario implies that there will be two  $e$ 's left in region 2. This case will allow  $\alpha_1$  to be communicated to region 0 in step 8 and subsequently, **no** is released to the environment in step 10. This means that not all of the vertices chosen to cover the edges in the setup phase belong to the candidate vertex cover.

The constructed family of ECPe systems used for VCP can also be used to solve ISP. This becomes apparent due to the lemma given in [3]. Note that the only difference will be the encoding of the instance for ISP where the initial copies of object  $c$  will be  $|V_G| - k$  and  $d$  has  $|E_G| - (|V_G| - k)$ . Also, at the end of a successful computation, the elements of  $IS_k$ ,  $1 \leq k \leq |V|$  is represented by object  $\hat{i}$  in region 1 (as illustrated by objects  $\hat{2}, \hat{5}$  representing  $IS_2 = \{2, 5\}$  in the previous example).

Formally, let the Independent Set Problem (ISP) be represented by a pair  $ISP = (I_{ISP}, \theta_{ISP})$  where  $I_{ISP} = \{w_{(G,k)} \mid w_{(G,k)} \text{ is a string representing a graph } G \text{ and a positive integer } k\}$ . If the graph  $G$  contains an independent set of size at least  $k$ ,  $\theta_{ISP}(w_{(G,k)}) = 1$ ; otherwise,  $\theta_{ISP}(w_{(G,k)}) = 0$ .

**Corollary 1.**  $ISP \in NFC\text{omNRW}(6, 4|V_G| - 3k + 6, 3|E_G| + 2|V_G| - k + 5)$  where  $E_G$  is the edge set and  $V_G$  is the vertex set of the input graph  $G$ .

We now present a solution to the 3-SAT problem in ECPe systems. Let the 3-SAT problem (3SP) be represented by a pair  $(I_{3SP}, \theta_{3SP})$  where  $I_{3SP} = \{w_{\phi_X} \mid w_{\phi_X} \text{ is a string representing a 3-CNF boolean formula } \phi_X\}$ . Boolean function  $\theta_{3SP}(w_{\phi_X})$  evaluates to 1 if  $\phi$  is satisfiable, otherwise,  $\theta_{3SP}(w_{\phi_X}) = 0$ .

**Theorem 2.**  $3SP \in NFC\text{omNRW}(5, 2n + 3, 4n + 3)$  where  $n$  is the number of clauses for the input 3-CNF boolean formula  $\phi_X$ .



*Proof.* A family of ECPe systems that solves the 3-SAT problem is represented as a construct  $\Pi_{3SP}$ :

$$\Pi_{3SP}(n) = (O, [0[1]_1 \dots [n]_n [n+1]_{n+1}]_0, w_0, \emptyset, \dots, \emptyset, R_0, R'_0, R_1, R'_1, \dots, R_n, R'_n, R_{n+1}, R'_{n+1})$$

where:

- $O = \{x_d, d, \hat{d} \mid 1 \leq d \leq 3n\} \cup \{0_{dq}, 1_{dq} \mid 1 \leq d \leq 3n, 1 \leq q \leq n\}$   
 $\cup \{A_{i_1 i_2 i_3, q}, B_{i_1 i_2 i_3, q} \mid 1 \leq q \leq n \text{ and } i_r \in \bigcup_{d=1}^{3n} \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\}$   
 $\cup \{c, \#_0, \#_1, \#_2, \#_3, \#_4, \Omega, \beta_0, \beta_1, \text{no}, \text{yes}, e\}$
- $w_0 = x_1 x_2 \dots x_{3n} \#_0 \text{cod}(w_{\phi_X})$
- $R_0 = \{x_d \rightarrow 0_{d1} 0_{d2} 0_{d3}, x_d \rightarrow 1_{d1} 1_{d2} 1_{d3} \mid 1 \leq d \leq 3n\}$   
 $\cup \{A_{i_1 i_2 i_3, q} \rightarrow B_{i_1 i_2 i_3, q} c e^2 \mid 1 \leq q \leq n \text{ and } i_r \in \bigcup_{d=1}^{3n} \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\}$   
 $\cup \{\#_0 \rightarrow \#_1 \Omega e, \#_1 \rightarrow \#_2, \#_2 \rightarrow \#_3, \#_3 \rightarrow \#_4, \#_4 \rightarrow \text{yes } e^2\}$   
 $\cup \{d \rightarrow e, \hat{d} \rightarrow e \mid 1 \leq d \leq 3n\}$
- $R'_0 = \{(\text{no } e, \text{out}), (\text{yes } e^{n+2}, \text{out})\}$
- For  $1 \leq q \leq n$ :
  - $R_q = \{B_{i_1 i_2 i_3, q} \rightarrow i_1 i_2 i_3 \beta_0 e \mid i_r \in \bigcup_{d=1}^{3n} \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\} \cup \{\beta_0 \rightarrow \beta_1\}$
  - $R'_q = \{(B_{i_1 i_2 i_3, q} e, \text{in}) \mid i_r \in \bigcup_{d=1}^{3n} \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\} \cup \{(ce, \text{in}; \beta_1 e, \text{out})\}$   
 $\cup \{(0_{dq} e, \text{in}; \hat{d} e, \text{out}), (1_{dq} e, \text{in}; d e, \text{out}) \mid 1 \leq d \leq 3n\}$
- $R_{n+1} = \{\Omega \rightarrow \text{no } e\}$
- $R'_{n+1} = \{(\Omega e, \text{in}), (\beta_1 e, \text{in}; \text{no } e, \text{out})\}$

We associate a pair  $(\text{cod}(w_{\phi_X}), s(w_{\phi_X}))$  over  $I_{\phi_X}$  where for each instance  $w_{\phi_X} \in I_{\phi_X}$  we have  $s(w_{\phi_X})$  being the number of clauses for the boolean formula  $\phi_X$ , i.e.  $s(w_{\phi_X}) = n$ . The encoding  $\text{cod}(w_{\phi_X})$  is a multiset containing  $A_{i_1 i_2 i_3, q}$  for  $1 \leq q \leq n$  where if  $C_q = y_{i_1, q} \vee y_{i_2, q} \vee y_{i_3, q}$ , then

$$i_l = \begin{cases} d & \text{if } y_{i_l, q} = x_d \\ \hat{d} & \text{if } y_{i_l, q} = \bar{x}_d \end{cases}$$

for  $l = 1, 2, 3$ , where  $x_d \in \{x_1, x_2, \dots, x_{3n}\}$ . It can be noticed that we limit the cardinality of  $X$  to be at most  $3n$  since the maximum number of variables that can simultaneously exist on a 3-CNF boolean formula is  $3n$  (that is, when all variables in all clauses are distinct). If cardinality of the set  $X$  is more than the number of variables present in the boolean formula, then the extra variables can take any boolean value without affecting the satisfiability of the formula being

evaluated. Our choice of  $s(w_{\phi_X})$  for an instance  $w_{\phi_X} \in I_{\phi_X}$  assures that  $s(w_{\phi_X})$  is a natural number. Furthermore, our constructed  $\Pi_{3SP}$  includes the encoding  $cod(w_{\phi_X})$  in region 1. This restriction guarantees that condition (i) of Definition 7 is satisfied.

To show that condition (ii) holds, we discuss how the computation proceeds as follows:

*Setup and finding a candidate solution phase.* In these steps, each variable is assigned a truth value. The input representation of each clause is also distributed to different regions.

The initial configuration requires object  $A_{i_1 i_2 i_3, q}$  in region 0 as input to represent each clause in  $\phi_X$  where  $q$  ( $1 \leq q \leq n$ ) symbolizes clause  $C_q$  and  $i_1, i_2$  and  $i_3$  corresponds to the variables contained in the clause  $C_q$ . In the next step, objects  $A_{i_1 i_2 i_3, q}$  will evolve to object  $B_{i_1 i_2 i_3, q}$  and  $c$ , producing two quanta of energy through rules  $A_{i_1 i_2 i_3, q} \rightarrow B_{i_1 i_2 i_3, q} c e^2$  ( $1 \leq q \leq n, i_l \in \bigcup_{d=1}^{3n} \{d, \hat{d}\}$ ). Objects  $x_d$  will be consumed through one of rules  $x_d \rightarrow 0_{d1} 0_{d2} 0_{d3}$  and  $x_d \rightarrow 1_{d1} 1_{d2} 1_{d3}$  ( $1 \leq d \leq 3n$ ) simultaneously. This choice represents the possible truth assignment for all variable  $x_d$  such that if the latter rule is used, this means  $x_d = 1$ , otherwise,  $x_d = 0$ . Also, during this step,  $\#_0$  evolves to  $\#_1 \Omega e$ . Upon completion of this step, the system determines a *candidate assignment* for variables in  $X$ .

*Validating candidate solution and output phase.* Simultaneously, each clause is checked whether it evaluates to true. If all clauses evaluate to true, the object *yes* is sent to the environment, otherwise, object *no* is sent out.

The next step involves validating the current candidate assignment. In this time step, object  $B_{i_1 i_2 i_3, q}$  is communicated to region  $q$  through rule  $(B_{i_1 i_2 i_3, q} e, in)$ . At the same time,  $\#_1$  in region 0 evolves to  $\#_2$  while  $\Omega$  enters region  $n+1$ . While  $\#_2$  in region 0 evolves to  $\#_3$  and  $\Omega$  in region  $n+1$  evolves to *no*  $e$ , the  $B_{i_1 i_2 i_3, q}$ 's evolve through rule  $B_{i_1 i_2 i_3, q} \rightarrow i_1 i_2 i_3 \beta_0 e$ . The objects  $i_1, i_2$  and  $i_3$  produced by this rule may take on values  $d$  (interpreted as  $x_d$  is contained in clause  $C_q$ ) or  $\hat{d}$  (interpreted as clause  $C_q$  contains  $\bar{x}_d$ ),  $x_d \in X$ . Also,  $\#_2$  in region 0 evolves to  $\#_3$  while  $\Omega$  in region  $n+1$  evolves to *no*  $e$ .

The quanta of energy left in region 0, as well as the object  $e$  produced in the aforementioned rule can be utilized to apply the antiport rules  $(0_{dq} e, in; \hat{d} e, out)$  and  $(1_{dq} e, in; d e, out)$  in any one of the objects  $i_1, i_2$  and  $i_3$  present in a region  $q$  ( $1 \leq q \leq n$ ). Meanwhile,  $\#_3$  evolves to  $\#_4$  in region 0 and  $\beta_0$  evolves to  $\beta_1$  in regions 1 to  $n$ . Note that only a single application of any one of the antiport rules can be applied per region, which represents that for a clause  $C_q = (y_{i_1, q} \vee y_{i_2, q} \vee y_{i_3, q})$ , at least one of the  $y_{i_l, q}$ 's ( $1 \leq q \leq 3$ ) evaluates to 1. If all  $C_q$ 's are satisfied, all the quanta of energy in regions 0 to  $n$  will be consumed. Henceforth, communication rules will no longer be applicable in membranes 1 to  $n+1$ . Afterwhich,  $\#_4$  evolves to *yes*  $e$  then *yes* is communicated to the environment. If at least one of the  $C_q$ 's is not satisfied, then at least one of regions 1 to  $n$  will have an  $e$  left, enabling at least one  $\beta_1$  to be communicated to region 0. The presence of a  $\beta_1$  in region 0 will allow the rule  $(\beta_1 e, in; no e, out)$  to

be used. This scenario results to the presence of **no** in region 0 and subsequently, the release of **no** to the environment.

We now evaluate the communication resources used as final requirement to satisfy Definition 9.

- In the *setup and finding a candidate phase*, only one communication step is needed to communicate each  $B$  object (representing a particular clause) from membrane 0 using one symport rule for each clause. This communication step also involves transporting an object  $\Omega$  in membrane  $n + 1$ .
  - number of communication step is one.
  - number of communication rules is  $n+1$ .
  - number of communication steps is  $n+1$ .
- In the *validation and output phase*,
  - the maximum number of communication step occurs when some clauses are satisfied and other clauses are not satisfied. This scenario will require two communication steps for the validation phase. Another two communication steps will be dedicated to communication a **no** to the skin, and outside the system. Thus, maximum number of communication step is four.
  - the maximum number of communication rules occurs when the chosen variable assignment of the system evaluates to false. In such a case, each of the  $n$  clauses will use any one of antiport rules  $(0_{dq}e, in; \widehat{de}, out)$  or  $(1_{dq}e, in; de, out)$  where  $d$  refers to a variable and  $q$  refers to a clause (for satisfied clauses) and  $(ce, in; \beta_1e, out)$  (for unsatisfied clauses). Note that since our rules are local to membranes/regions, the  $(ce, in; \beta_1e, out)$  in a membrane  $i$  is different from  $(ce, in; \beta_1e, out)$  in a membrane  $j$ . The output phase will be using two communication rules,  $(\beta_1e, in; no\ e, out)$  and  $(no\ e, out)$ . Therefore, the maximum number of communication rules will be  $n + 2$ .
  - the maximum number of communication rules occurs when the chosen variable assignment of the system evaluates to true. In such a case, there will be  $2n$  quanta of energy needed for executing antiport rules for satisfied clause (as shown in previous item) and another  $n$  quanta of energy for executing  $(yes\ e^{n+2}, out)$ . This occurrence means the maximum number of quanta of energy used is  $3n + 2$ .

It can be observed that the computations to compute the maximum number of communication steps, rules and objects are not necessarily the same. In fact, while the maximum number of communication rules occurs when the chosen variable assignments evaluates to false, the maximum number of energy for communication occurs when the chosen variable assignment evaluates to true. In summary,  $ComN(3SP, \Pi_{3SP}(n)) = 5$ ,  $ComR(3SP, \Pi_{3SP}(n)) = 2n + 3$ , and  $ComW(3SP, \Pi_{3SP}(n)) = 4n + 3$ .

*An Example for 3SP.* Given an instance of a 3-CNF boolean formula  $\phi_x = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$  where  $X = \{x_1, x_2, x_3, x_4\}$ ,

an ECPe system solving 3SP is a construct:

$$\Pi_{3SP}(3) = (O, [0[1]1[2]2[3]3[4]4]_0, w_0, \emptyset, \dots, \emptyset, R_0, R'_0, R_1, R'_1, R_2, R'_2, R_3, R'_3, R_4, R'_4)$$

where:

- $O = \{x_d, d, \hat{d} \mid 1 \leq d \leq 9\} \cup \{0_{dq}, 1_{dq} \mid 1 \leq d \leq 9, 1 \leq q \leq 3\}$   
 $\cup \{A_{i_1 i_2 i_3, q}, B_{i_1 i_2 i_3, q} \mid 1 \leq q \leq 3 \text{ and } i_r \in \bigcup_{d=1}^9 \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\}$   
 $\cup \{c, \#_0, \#_1, \#_2, \#_3, \#_4, \Omega, \beta_0, \beta_1, \text{no}, \text{yes}, e\}$
- $w_0 = x_1 x_2 \dots x_9 \#_0 \text{cod}(w_{\phi_X})$
- $R_0 = \{x_d \rightarrow 0_{d1} 0_{d2} 0_{d3}, x_d \rightarrow 1_{d1} 1_{d2} 1_{d3} \mid 1 \leq d \leq 9\}$   
 $\cup \{A_{i_1 i_2 i_3, q} \rightarrow B_{i_1 i_2 i_3, q} c e^2 \mid 1 \leq q \leq 3 \text{ and } i_r \in \bigcup_{d=1}^9 \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\}$   
 $\cup \{\#_0 \rightarrow \#_1 \Omega e, \#_1 \rightarrow \#_2, \#_2 \rightarrow \#_3, \#_3 \rightarrow \#_4, \#_4 \rightarrow \text{yes} e^2\}$   
 $\cup \{d \rightarrow e, \hat{d} \rightarrow e \mid 1 \leq d \leq 9\}$
- $R'_0 = \{(\text{no } e, \text{out}), (\text{yes } e^5, \text{out})\}$
- For  $1 \leq q \leq 3$ :
  - $R_q = \{B_{i_1 i_2 i_3, q} \rightarrow i_1 i_2 i_3 \beta_0 e \mid i_r \in \bigcup_{d=1}^9 \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\} \cup \{\beta_0 \rightarrow \beta_1\}$
  - $R'_q = \{(B_{i_1 i_2 i_3, q} e, \text{in}) \mid i_r \in \bigcup_{d=1}^9 \{d, \hat{d}\}, \forall r \in \{1, 2, 3\}\} \cup \{(ce, \text{in}; \beta_1 e, \text{out})\}$   
 $\cup \{(0_{dq} e, \text{in}; \hat{d} e, \text{out}), (1_{dq} e, \text{in}; d e, \text{out}) \mid 1 \leq d \leq 9\}$
- $R_4 = \{\Omega \rightarrow \text{no } e\}$
- $R'_4 = \{(\Omega e, \text{in}), (\beta_1 e, \text{in}; \text{no } e, \text{out})\}$

Below is an example of a computation for  $\Pi_{3SP}(3)$ , represented as a series of configurations ( $C_i$ ) ( $0 \leq i \leq 6$ ):

$C_0$ :  $[0 \ x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 \ \#_0 \ A_{\hat{1}23,1} A_{\hat{1}23,2} A_{\hat{1}24,3} \ [1]_1 \ [2]_2 \ [3]_3 \ [4]_4]_0$   
 $C_1$ :  $[0 \ 0_{11} 0_{12} 0_{13} \ 0_{21} 0_{22} 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} 1_{43} \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}$   
 $0_{81} 0_{82} 0_{83} 0_{91} 0_{92} 0_{93} \ \#_1 \Omega e \ B_{\hat{1}23,1} c e^2 \ B_{\hat{1}23,2} c e^2 \ B_{\hat{1}24,3} c e^2 \ [1]_1 \ [2]_2 \ [3]_3 \ [4]_4]_0$   
 $C_2$ :  $[0 \ 0_{11} 0_{12} 0_{13} \ 0_{21} 0_{22} 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} 1_{43} \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}$   
 $0_{81} 0_{82} 0_{83} \ 0_{91} 0_{92} 0_{93} \ \#_2 \ c^3 e^3 [1 \ B_{\hat{1}23,1}]_1 [2 \ B_{\hat{1}23,2}]_2 [3 \ B_{\hat{1}24,3}]_3 [4 \ \Omega]_4]_0$   
 $C_3$ :  $[0 \ 0_{11} 0_{12} 0_{13} \ 0_{21} 0_{22} 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} 1_{43} \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}$   
 $0_{81} 0_{82} 0_{83} \ 0_{91} 0_{92} 0_{93} \ \#_3 \ c^3 e^3 [1 \ \hat{1}23 \beta_0 e]_1 [2 \ \hat{1}23 \beta_0 e]_2 [3 \ \hat{1}24 \beta_0 e]_3 [4 \ \text{no } e]_4]_0$   
 $C_4$ :  $[0 \ \hat{1} \ 0_{12} 0_{13} \ 0_{21} \ \hat{2} \ 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} \ 4 \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}$   
 $0_{81} 0_{82} 0_{83} \ 0_{91} 0_{92} 0_{93} \ \#_4 \ c^3 [1 \ 0_{11} \ 23 \ \beta_1]_1 [2 \ 1 \ 0_{22} \ 3 \ \beta_1]_2 [3 \ \hat{1}2 \ 1_{43} \ \beta_1]_3 [4 \ \text{no } e]_4]_0$   
 $C_5$ :  $[0 \ e \ 0_{12} 0_{13} \ 0_{21} \ e \ 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} \ e \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}$   
 $0_{81} 0_{82} 0_{83} \ 0_{91} 0_{92} 0_{93} \ \text{yes } e^2 \ c^3 [1 \ 0_{11} \ 23 \ \beta_1]_1 [2 \ 1 \ 0_{22} \ 3 \ \beta_1]_2 [3 \ \hat{1}2 \ 1_{43} \ \beta_1]_3 [4 \ \text{no } e]_4]_0$   
 $C_6$ :  $\text{yes } [0 \ 0_{12} 0_{13} \ 0_{21} 0_{23} \ 0_{31} 0_{32} 0_{33} \ 1_{41} 1_{42} \ 0_{51} 0_{52} 0_{53} \ 0_{61} 0_{62} 0_{63} \ 0_{71} 0_{72} 0_{73}]_0$

$$0_{81}0_{82}0_{83} \ 0_{91}0_{92}0_{93} \ c^3 \ [1 \ 0_{11} \ 23 \ \beta_1]_1 \ [2 \ 1 \ 0_{22} \ 3 \ \beta_1]_2 \ [3 \ \hat{1}2 \ 1_{43} \ \beta_1]_3 \ [4 \ \text{no} \ e]_4 \ ]_0$$

Configurations  $C_0$  to  $C_3$  represent the set-up phase where necessary objects are placed in their respective regions for the succeeding phases. At the same time, transition  $C_0 \Rightarrow C_1$  makes use of rules in  $\{x_d \rightarrow 0_{d1}0_{d2}0_{d3}, x_d \rightarrow 1_{d1}1_{d2}1_{d3} \mid 1 \leq d \leq 9\}$  to choose assignment for each variable  $x_d \in X$ .

Transition  $C_3 \Rightarrow C_4$  represents the validation phase where the antiport rules  $\{(0_{dq}e, in; \hat{d}e, out), (1_{dq}e, in; de, out) \mid 1 \leq d \leq 9\}$  are used to check if the candidate assignments satisfies all clauses.

Note that in  $C_4$  all the quanta of energy in regions 0 to 3 were consumed. Henceforth, communication rules will no longer be applicable in membranes 1 to 4. Finally, transition  $C_4 \Rightarrow^* C_6$ , represents the output phase where the object **yes** is released to the environment to mean that a satisfying assignment was found for the given 3-CNF formula.

If we introduce a different transition  $C_0 \Rightarrow C'_1$  where configuration  $C'_1$  is represented as:

$$C'_1: [0 \ 1_{11}1_{12}1_{13} \ 0_{21}0_{22}0_{23} \ 1_{31}1_{32}1_{33} \ 0_{41}0_{42}0_{43} \ 0_{51}0_{52}0_{53} \ 0_{61}0_{62}0_{63} \ 0_{71}0_{72}0_{73}$$

$$0_{81}0_{82}0_{83}0_{91}0_{92}0_{93} \ \#_1 \Omega e \ B_{123,1}ce^2 \ B_{123,2}ce^2 \ B_{124,3}ce^2 \ [1]_1 \ [2]_2 \ [3]_3 \ [4]_4 \ ]_0$$

representing the assignment  $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$ . Note that in step 4, a quanta of energy is left in each of the regions 0 and 3. Hence, in the next step, we can apply the communication rule  $(ce, in; \beta_1 e, out)$  in membrane 3. The presence of  $\beta_1$  in region 0 allows the application of  $(\beta_1 e, in; \text{no} \ e, out)$  in membrane 4. Finally, in step 7, the object **no** is released to the environment to mean that the candidate solution does not satisfy the given 3-CNF formula.

## 5 Conclusion

In this paper, we studied the communication resources needed to non-confluently decide NP-complete problems namely, Vertex Cover Problem (VCP), consequently Independent Set Problem (ISP), and 3-SAT problem (3SP) using recognizer Evolution-Communication P systems with energy (ECPe systems).

In the solutions presented, it can be observed that while the number of membranes needed to solve VCP is constant (exactly four membranes), the number of membranes needed to solve 3SP is dependent on the number of clauses. However, in the results presented in both solutions, the number of communication steps are constant whereas the number of communication rules and energy for communication is dependent on the number of vertices and edges (for VCP), and clauses (for 3SP).

It remains an open problem whether we can reduce the number of communication steps, rules and energy; for example, can we construct recognizer ECPe systems using constant amount of rules or energy for communication? Also, from our results, it can be observed that the amount of communication steps needed to solve VCP is greater than the amount needed to solve 3SP, can we achieve a

better result? Otherwise, can we characterize the class of problems that can be decided using five communication steps? six communication steps? or lower number of communication steps? It is also worth mentioning that the constructed ECPe systems used in this paper decides non-confluently. Evaluating communication resources on ECPe systems that decide on problems confluently also remains to be explored.

As final remarks, part of our future work includes exploring the use of carpets in understanding communication over the recognizer ECPe systems defined for solving VCP and 3SP. It is worth noting that Sevilla carpets can be used to provide a visualization of communication on ECPe systems as explored in [5].

## Acknowledgments

N.H. S. Hernandez would like to thank the UP Diliman College of Engineering through the Jose P. Dans Jr. professorial chair for the financial support. R.A. B. Juayong is supported by the Engineering Research and Development for Technology (ERDT) Scholarship Program. H.N. Adorna is funded by a DOST-ERDT research grant and the Semirara Mining Corporation professorial chair of the UP Diliman, College of Engineering.

## References

1. H. Adorna, Gh. Păun, M. Pérez-Jiménez. On Communication Complexity in Evolution-Communication P systems, *Romanian Journal of Information Science and Technology*, Vol. 13, No. 2, pp. 113-130, 2010.
2. M. Cavaliere. Evolution-Communication P systems. *Membrane Computing. Proc. WMC 2002, Curtea de Argeş* (Gh. Păun et al., eds.), LNCS 2597, Springer, Berlin, pp. 134-145, 2003.
3. M.R. Garey, D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979, ISBN 0-7167-1044-7.
4. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez. Computing Backwards with P systems. *WMC10, Curtea de Argeş, Romania*, pp. 282-295, 2009.
5. R.A. B. Juayong, H. N. Adorna. Communication Complexity of Evolution-Communication P systems with Energy and Sevilla Carpet. *Philippine Computing Journal*, Vol. 6, No. 1, pp. 34-40, 2010.
6. Gh. Păun. Introduction to Membrane Computing. In: *Gabriel Ciobanu, Mario J. Pérez-Jiménez and Gheorghe Păun, eds: Applications of Membrane Computing, Natural Computing Series*. Springer, pp.1-42. (2006)
7. M. Pérez-Jiménez. A Computational Complexity Theory in Membrane Computing. *Workshop on Membrane Computing*, pp. 125-148, 2009.
8. A. E. Porreca, G. Mauri, C. Zandron. Non-confluence in divisionless P systems with active membranes. *Theoretical Computer Science* Vol. 411, No. 6, pp. 878-887, 2010.
9. X. Zeng, H. Adorna, M. A. Martínez-del-Amor, L. Pan, M. Pérez-Jiménez. Matrix Representation of Spiking Neural P Systems, *Membrane Computing: Lecture Notes in Computer Science*, Vol. 6501, pp. 377-391, 2011.

# About One-Sided One-Symbol Insertion-Deletion P Systems

Sergiu Ivanov<sup>1</sup> and Sergey Verlan<sup>1,2</sup>

<sup>1</sup> Laboratoire d'Algorithmique, Complexité et Logique,  
Université Paris Est – Créteil Val de Marne,  
61, av. gén. de Gaulle, 94010 Créteil, France  
email: {sergiu.ivanov,verlan}@u-pec.fr

<sup>2</sup> Institute of Mathematics and Computer Science,  
Academy of Sciences of Moldova,  
Academiei 5, Chisinau, MD-2028, Moldova

**Abstract.** In this article we consider insertion-deletion P systems inserting or deleting one symbol in one or two symbol(s) left context (more precisely of size  $(1, 2, 0; 1, 1, 0)$  and  $(1, 1, 0; 1, 2, 0)$ ). We show that computational completeness can be achieved by using only 3 membranes in a tree-like structure. Hence we obtain a trade-off between the sizes of contexts of insertion and deletion rules and the number of membranes sufficient for computational completeness.

## 1 Introduction

The operations of insertion and deletion were first considered with a linguistic motivation [19, 8, 22]. Another inspiration for these operations comes from the fact that the insertion operation and its iterated variants are generalized versions of Kleene's operations of concatenation and closure [14], while the deletion operation generalizes the quotient operation. A study of properties of the corresponding operations may be found in [10–12]. However, insertion and deletion also have interesting biological motivations, e.g., they correspond to a mismatched annealing of DNA sequences; these operations are also present in the evolution processes in the form of point mutations as well as in RNA editing, see the discussions in [3, 4, 26] and [24]. These biological motivations of insertion-deletion operations led to their study in the framework of molecular computing, see, for example, [6, 13, 24, 27].

In general, an insertion operation means adding a substring to a given string in a specified (left and right) context, while a deletion operation means removing a substring of a given string from a specified (left and right) context. A finite set of insertion-deletion rules, together with a set of axioms provide a language generating device: starting from the set of initial strings and iterating insertion-deletion operations as defined by the given rules, one gets a language.

Even in their basic variants, insertion-deletion systems are able to characterize the recursively enumerable languages. Moreover, as it was shown in [20], the context dependency may be replaced by insertion and deletion of strings of

sufficient length, in a context-free manner. If the length is not sufficient (less or equal to two) then such systems are decidable and a characterization of them was shown in [28].

Similar investigations were continued in [21, 16, 17] on insertion-deletion systems with one-sided contexts, i.e., where the context dependency is present only from the left or only from the right side of all insertion and deletion rules. The papers cited above give several computational completeness results depending on the size of parameters of insertion and deletion rules. We recall the interesting fact that some combinations are not leading to computational completeness, i.e., there are recursively enumerable languages that cannot be generated by such devices, in particular, by systems of size  $(1, 1, 0; 1, 1, 0)$ , where the first three numbers represent the maximal size of the inserted string and the maximal size of the left and right contexts, respectively, while the last three numbers provide the same information about deletion rules.

In order to increase the computational power of the corresponding variants they were considered in the framework of P systems [18] and it was shown that computational completeness can be achieved if 5 membranes are used. In [7] tissue P systems are considered and computational completeness is achieved with 4 membranes. In [2] computational completeness is achieved by simpler insertion-deletion rules, but instead using priorities. A summary of related results can be found in [1, 29].

In this article we would like to consider the trade-offs between the sizes of the contexts and the number of membranes. We consider insertion-deletion P systems of size  $(1, 2, 0; 1, 1, 0)$  and  $(1, 1, 0; 1, 2, 0)$ , and show that computational completeness can be achieved with only 3 membranes. We remind that previously it was shown that 4 membranes are enough to achieve computational completeness with insertion and deletion rules of size  $(1, 1, 0)$ .

## 2 Preliminaries

In this paper, the empty string is denoted by  $\lambda$ , the family of recursively enumerable, context-sensitive, and context-free languages by  $RE$ ,  $CS$  and  $CF$ , respectively. We will use the notation  $|w|$  for the length of a string  $w$ , while the number of occurrences of the symbol  $a$  in the string  $w$  will be referred to by the notation  $|w|_a$ . We do not define the standard concepts of the theory of formal languages in this section; the reader is invited to consider [25] for further details.

A type-0 grammar  $G = (N, T, S, P)$  is said to be in *Geffert normal form* [9] if the set of non-terminals  $N$  is defined as  $N = \{S, A, B, C, D\}$ ,  $T$  is an alphabet and  $P$  only contains context-free rules of the forms  $S \rightarrow uSv$  with  $u \in \{A, C\}^+$  and  $v \in (T \cup \{B, D\})^+$  as well as  $S \rightarrow \lambda$  and two (non-context-free) erasing rules  $AB \rightarrow \lambda$  and  $CD \rightarrow \lambda$ .

We remark that according to [9] the generation of a string using a grammar in this normal form is done in two stages. During the first stage only context-free rules  $S \rightarrow uSv$  can be applied (this follows from the fact that  $u \in \{A, C\}^+$  and  $v \in (\{B, D\} \cup T)^+$ ). During the second stage only non-context-free rules can



be applied (because there is no more symbol  $S$  in the string). The transition between the stages is done by the rule  $S \rightarrow \lambda$  (note that in [9] a set of rules of the form  $S \rightarrow uv$  is used instead leading to an equivalent result). Note that the symbols  $A, B, C, D$  are treated like terminals during the first stage and so, each rule  $S \rightarrow uSv$  is in some sense “linear”.

Throughout this paper we will use the special Geffert normal form. Let  $G = (N, T, S, P)$  be a grammar with  $N = N' \cup N''$ ,  $N' \cap N'' = \emptyset$ , where  $N'' = \{A, B, C, D\}$  and  $N'$  is a set of non-terminals containing  $S$ ,  $S'$  and some other auxiliary non-terminals (that are introduced by the translation from the Geffert normal form to the special variant). We say that  $G$  is in the *special Geffert normal form* if it only has two (non-context-free) erasing rules  $AB \rightarrow \lambda$  and  $CD \rightarrow \lambda$  and several context-free rules of one of the following forms:

$$\begin{aligned} X &\rightarrow bY, & \text{where } X, Y \in N', b \in N'', X \neq Y \\ X &\rightarrow Yb, & \text{where } X, Y \in N', b \in T \cup N'', X \neq Y \\ S' &\rightarrow \lambda. \end{aligned}$$

Moreover, it may be assumed without loss of generality that for any two rules  $X \rightarrow w$  and  $U \rightarrow w$  in  $P$  with the first symbol of  $w$  different from  $S, S'$ , we have  $U = X$ .

Any grammar  $G$  in the Geffert normal form can be transformed into a grammar  $G'$  in the special Geffert normal form generating the same language by replacing the “linear” rules by right- and left-linear ones. Let  $S'$  be a new non-terminal that will be used to mark the transition from the first stage to the second. The rule  $S \rightarrow uSv$  of  $G$ , where  $u = a_1 \dots a_n$  and  $v = b_1 \dots b_m$  is replaced in  $G'$  by the following rules:  $S \rightarrow a_1 X_1$ ,  $X_1 \rightarrow a_2 X_2$ ,  $\dots$ ,  $X_{n-1} \rightarrow a_n X_n$ ,  $X_n \rightarrow X_{n+1} b_m$ ,  $\dots$ ,  $X_{n+m} \rightarrow S b_1$ , where  $X_1, \dots, X_{n+m}$  are new non-terminals different from each other as well as from the corresponding non-terminals introduced by the translation of other rules. We also add rules  $X_{n+m} \rightarrow S' b_m$  and  $S' \rightarrow \lambda$  to  $G'$  in order to mark the transition to the second stage. Note that the rule  $S \rightarrow \lambda$  is not preserved in  $G'$ .

We also note that during the first stage of the derivation of a grammar in the special Geffert normal form there is exactly one non-terminal from  $N'$  present in the string and during the second stage the string does not contain any symbol from  $N'$ .

An *insertion-deletion system* is a construct  $\Gamma = (V, T, A, I, D)$ , where  $V$  is an alphabet,  $T \subseteq V$  is the *terminal* alphabet (the symbols from  $V \setminus T$  are called *non-terminal* symbols),  $A \subseteq V^*$  is the set of *axioms*, and  $I$  and  $D$  are finite sets of triples of the form  $(u, \alpha, v)$ , where  $u, \alpha$ , and  $v$  are strings over  $V$ , with  $\alpha \neq \lambda$ . The triples in  $I$  are called *insertion rules*, and those in  $D$  are called *deletion rules*.

An insertion rule  $(u, \alpha, v) \in I$  indicates that the string  $\alpha$  can be inserted between  $u$  and  $v$ , while a deletion rule  $(u, \alpha, v) \in D$  indicates that  $\alpha$  can be removed from between the contexts  $u$  and  $v$ . In other words,  $(u, \alpha, v) \in I$  corresponds to the rewriting rule  $uv \rightarrow u\alpha v$ , while  $(u, \alpha, v) \in D$  corresponds to the rewriting rule  $u\alpha v \rightarrow uv$ .

We denote the “derives by insertion” relation induced by insertion rules by  $\Rightarrow_{ins}$ . Formally,  $x \Rightarrow_{ins} y$  (“ $x$  derives  $y$  by insertion”) if and only if  $x = x_1uvx_2$  and  $y = x_1u\alpha vx_2$ ,  $x_1, x_2 \in V^*$ , and there exists  $(u, \alpha, v) \in I$ . By the notation  $\Rightarrow_{del}$  we refer to the “derives by deletion” relation defined by deletion rules. Formally,  $x \Rightarrow_{del} y$  (“ $x$  derives  $y$  by deletion”) if and only if  $x = x_1u\alpha vx_2$  and  $y = x_1uvx_2$ ,  $x_1, x_2 \in V^*$ , and there exists  $(u, \alpha, v) \in D$ . By  $\Rightarrow$  we refer to the union of the relations  $\Rightarrow_{ins}$  and  $\Rightarrow_{del}$ , and by  $\Rightarrow^*$  we denote the reflexive and transitive closure of  $\Rightarrow$ .

Instead of relying on separate sets  $I$  and  $D$ , we will often consider their union  $R = I \cup D$  and distinguish between insertion and deletion rules by the subscripts  $ins$  and  $del$ . Thus instead of  $(u, \alpha, v) \in I$ , we will write  $(u, \alpha, v)_{ins}$ , and instead of  $(u, \alpha, v) \in D$ , we will write  $(u, \alpha, v)_{del}$ .

The language generated by the insertion-deletion system  $\Gamma = (V, T, A, I, D)$  is defined as follows:

$$L(\Gamma) = \{w \in T^* \mid x \xRightarrow{*} w, x \in A\}.$$

The complexity of an insertion-deletion system  $\Gamma = (V, T, A, I, D)$  is described by the vector  $(n, m, m'; p, q, q')$  called *size*, where

$$\begin{aligned} n &= \max\{|\alpha| \mid (u, \alpha, v) \in I\}, & p &= \max\{|\alpha| \mid (u, \alpha, v) \in D\}, \\ m &= \max\{|u| \mid (u, \alpha, v) \in I\}, & q &= \max\{|u| \mid (u, \alpha, v) \in D\}, \\ m' &= \max\{|v| \mid (u, \alpha, v) \in I\}, & q' &= \max\{|v| \mid (u, \alpha, v) \in D\}. \end{aligned}$$

The *total size* of an insertion-deletion system  $\Gamma$  of size  $(n, m, m'; p, q, q')$  is defined as the sum of all the numbers from the vector:  $\Sigma(\Gamma) = n + m + m' + p + q + q'$ .

By  $INS_n^{m, m'} DEL_p^{q, q'}$  we denote the families of languages generated by insertion-deletion systems of size  $(n, m, m'; p, q, q')$ .

If  $*$  is specified instead of one of the parameters  $n, m, m', p, q$ , or  $q'$ , then there are no restrictions on the length of the corresponding component. In particular,  $INS_*^{0, 0} DEL_*^{0, 0}$  denotes the family of languages generated by context-free insertion-deletion systems.

If one of the numbers from the pairs  $m, m'$  or  $q, q'$  is equal to zero, while the other one is not, we say that the family is with one-sided context.

An *insertion-deletion P system* of degree  $n$  is the following construct:

$$\Pi = (V, T, \mu, M_1, \dots, M_n, R_1, \dots, R_n)$$

where

- $V$  is a finite alphabet,
- $T \subseteq V$  is the terminal alphabet,
- $\mu$  is the membrane (tree) structure of the system which has  $n$  membranes (nodes). This structure will be represented by a word containing correctly nested marked parentheses.
- $M_i$ , for each  $1 \leq i \leq n$  is a finite language associated with the membrane  $i$ .

- $R_i$ , for each  $1 \leq i \leq n$  is a set of insertion and deletion rules with target indicators associated with membrane  $i$  and having the following forms:  $(u, x, v; tar)_{ins}$ , where  $(u, x, v)$  is an insertion rule, and  $(u, x, v; tar)_{del}$ , where  $(u, x, v)$  is a deletion rule, and  $tar$ , called the *target indicator*, is from the set  $\{here, in, out\}$ .

Any  $n$ -tuple  $(N_1, \dots, N_n)$  of languages over  $V$  is called a configuration of  $\Pi$ . For two configurations  $(N_1, \dots, N_n)$  and  $(N'_1, \dots, N'_n)$  of  $\Pi$  we write  $(N_1, \dots, N_n) \Rightarrow (N'_1, \dots, N'_n)$  if one can pass from  $(N_1, \dots, N_n)$  to  $(N'_1, \dots, N'_n)$  by applying the insertion and deletion rules from each region of  $\mu$ , in the maximally parallel way, i.e., in parallel to all possible strings from the corresponding regions, and following the target indications associated with the rules. We assume that every string represented in a membrane has arbitrary many copies. Hence, by applying a rule to a string we get both arbitrary many copies of resulting string as well as old copies of the same string.

More specifically, if  $w \in N_i$  and  $r = (u, x, v; tar)_{ins} \in R_i$ , respectively  $r = (u, x, v; tar)_{del} \in R_i$ , such that  $w \xRightarrow{r}_{ins} w'$ , respectively  $w \xRightarrow{r}_{del} w'$ , then  $w'$  will go to the region indicated by  $tar$ . If  $tar = here$ , then the string remains in  $N_i$ , if  $tar = out$ , then the string is moved to the region immediately outside the membrane  $i$  (maybe, in this way the string leaves the system), if  $tar = in$ , then the string is moved to one of the regions immediately below region  $j$ .

A sequence of transitions between configurations of a given insertion-deletion P system  $\Pi$ , starting from the initial configuration  $(M_1, \dots, M_n)$ , is called a computation with respect to  $\Pi$ . The result of a computation consists of all strings over  $T$  which are sent out of the system at any time during the computation. We denote by  $L(\Pi)$  the language of all strings of this type. We say that  $L(\Pi)$  is generated by  $\Pi$ .

As in [23] we denote by  $ELSP_k(ins_n^{m,m'}, del_p^{q,q'})$  the family of languages generated by insertion-deletion P systems of degree at most  $k \geq 1$  having the size  $(n, m, m'; p, q, q')$ .

### 3 Computational power of one-sided insertion-deletion systems of small size

In this section we consider insertion-deletion P systems of size  $(1, 2, 0; 1, 1, 0)$  and  $(1, 1, 0; 1, 2, 0)$ . While the computational power of normal insertion-deletion systems with these parameters is not yet known, based on observations from [15] we conjecture that the corresponding models are not computationally complete. We also recall that most combinations of parameters involving left and right contexts as well as the insertion or deletion of more than one symbol are known to produce computationally complete insertion-deletion systems, see [29] for a complete list.

**Theorem 1**  $ELSP_3(ins_1^{2,0}, del_1^{1,0}) = RE$ .

*Proof.* Consider a type-0 grammar  $G = (N, T, P, S)$  in the special Geffert normal form and let  $N'' = \{A, B, C, D\} \subseteq N$ . We construct an insertion-deletion P system

$$\Pi = (V, T, [1[2[3]3]2]_1, \{\{\mathcal{X}S\}\}, \emptyset, \emptyset, R_1 \cup R'_1, R_2, R_3)$$

that simulates  $G$  as follows. The rules from  $P$  are supposed to be labeled in a one-to-one manner with labels from the set  $[1..|P|]$ . The alphabet of  $\Pi$  is  $V = N \cup T \cup \{M_i \mid i : X \rightarrow \alpha \in P\} \cup \{K, K', \mathcal{X}\}$ . The sets of rules  $R_1, R_2, R_3$  of  $\Pi$  are defined as follows.

For any rule  $i : X \rightarrow bY \in P$  we consider following sets of rules:

$$\begin{aligned} R_1^i &= \{i.1 : (X, M_i, \lambda; in)_{ins}\}, \\ R_2^i &= \{i.2 : (XM_i, Y, \lambda; in)_{ins}\} \cup \{i.3 : (\mathbf{a}, M_i, \lambda; out)_{del} \mid \mathbf{a} \in N''\}, \\ R_3^i &= \{i.4 : (\mathbf{a}, X, \lambda; here)_{del} \mid \mathbf{a} \in N''\} \cup \{i.5 : (\mathbf{a}M_i, b, \lambda; out)_{ins} \mid \mathbf{a} \in N''\}. \end{aligned}$$

For any rule  $i : X \rightarrow Yb$  we consider following sets of rules:

$$\begin{aligned} R_1^i &= \{i.1 : (X, M_i, \lambda; in)_{ins}\}, \\ R_2^i &= \{i.2 : (XM_i, b, \lambda; in)_{ins}\} \cup \{i.3 : (\mathbf{a}, M_i, \lambda; out)_{del} \mid \mathbf{a} \in N''\}, \\ R_3^i &= \{i.4 : (\mathbf{a}, X, \lambda; here)_{del} \mid \mathbf{a} \in N''\} \cup \{i.5 : (\mathbf{a}M_i, Y, \lambda; out)_{ins} \mid \mathbf{a} \in N''\}. \end{aligned}$$

For the rules  $i_1 : AB \rightarrow \lambda$  and  $i_2 : CD \rightarrow \lambda$  and  $i_3 : S' \rightarrow \lambda$  we consider following sets of rules:

$$\begin{aligned} R_1^{i_1} &= \{i_1.1 : (\lambda, K, \lambda; in)_{ins}\}, \\ R_2^{i_1} &= \{i_1.2 : (K, A, \lambda; in)_{del}\} \cup \{i_1.3 : (\lambda, K, \lambda; out)_{del}\}, \\ R_3^{i_1} &= \{i_1.4 : (K, B, \lambda; out)_{del}\}, \\ R_1^{i_2} &= \{i_2.1 : (\lambda, K', \lambda; in)_{ins}\}, \\ R_2^{i_2} &= \{i_2.2 : (K', C, \lambda; in)_{del}\} \cup \{i_2.3 : (\lambda, K, \lambda; out)_{del}\}, \\ R_3^{i_2} &= \{i_2.4 : (K', D, \lambda; out)_{del}\}, \\ R_1^{i_3} &= \{i_3.1 : (\lambda, S', \lambda; here)_{del}\}. \end{aligned}$$

Now for  $j = 1, 2, 3$  we define  $R_j = \cup_{1 \leq i \leq |P|} R_j^i$  and we define  $R'_1 = \{\mathcal{X} : (\lambda, \mathcal{X}, \lambda; out)_{del}\}$ .

We state that  $L(\Pi) = L(G)$ . For this we show how each rule of  $G$  can be simulated in  $\Pi$ . Consider a string  $wXw'$  in membrane 1 and suppose that there is a rule  $i : X \rightarrow bY$  in  $P$ . Then the following unique evolution can happen:

$$\begin{aligned} (wXw', 1) &\Longrightarrow_{i.1} (wXM_iw', 2) \Longrightarrow_{i.2} (wXM_iYw', 3) \Longrightarrow_{i.4} \\ &\Longrightarrow_{i.4} (wM_iYw', 3) \Longrightarrow_{i.5} (wM_ibYw', 2) \Longrightarrow_{i.3} (wbYw', 1). \end{aligned}$$

In the second step it was possible to apply the rule  $i.3$ , yielding string  $wXw'$  in membrane 1, but this just returns to the previous configuration.

The rule  $X \rightarrow Yb$  is simulated in a similar manner:

$$\begin{aligned} (wXw', 1) &\Rightarrow_{i.1} (wXM_iw', 2) \Rightarrow_{i.2} (wXM_ibw', 3) \Rightarrow_{i.4} \\ &\Rightarrow_{i.4} (wM_ibw', 3) \Rightarrow_{i.5} (wM_iYbw', 2) \Rightarrow_{i.3} (wYbw', 1). \end{aligned}$$

The rule  $i_1 : AB \rightarrow \lambda$  is simulated as follows (the case of rule  $i_2 : CD \rightarrow \lambda$  is treated in an analogous way). First a symbol  $K$  is inserted in a context-free manner into the string  $ww'$  by using the rule  $i_1.1$ , yielding  $wKw'$ . If the symbol to the right of  $K$  is not an  $A$ , then the only possibility is to apply rule  $i_1.3$  which deletes  $K$  and returns the string  $ww'$  to membrane 1. If  $K$  is inserted in front of a symbol  $A$  ( $w' = Aw''$ ) then rule  $i_1.2$  can be applied and string  $wKw''$  goes to membrane 3. Now if  $w''$  does not start with  $B$ , then the computation of this word is stopped and it does not yield a result. Otherwise ( $w'' = Bw'''$ ), rule  $i_1.4$  is applied yielding  $wKw'''$  in membrane 2. Now the computation may be continued in the same manner and  $K$  either eliminates another couple of symbols  $AB$  if this is possible, or the string appears in the skin membrane without  $K$  and then is ready for new evolutions.

When the system  $\Pi$  reaches the configuration  $\mathcal{X}w$  with  $w \in T^*$ , rule  $\mathcal{X}$  from  $R'_1$  can be applied yielding a terminal string  $w$  in the environment as a result of the computation.

Now in order to complete the proof, we observe that the only sequences of rules leading to a terminal derivation in  $\Pi$  correspond to the groups of rules as defined above. Hence, a derivation in  $G$  can be reconstructed from a derivation in  $\Pi$ .  $\square$

**Theorem 2**  $ELSP_3(ins_1^{1,0}del_1^{2,0}) = RE$ .

*Proof.* Consider the type-0 grammar  $G = (N, T, S, P)$  in the special Geffert normal form and denote  $N'' = \{A, B, C, D\} \subseteq N$ . Consider as well that the rules from  $P$  are bijectively labelled with the numbers from the set  $[1..|P|]$ . We will now construct the following insertion-deletion P system  $\Pi$  which simulates  $G$ :

$$\Pi = (V, T, [_1[_2[_3]_3]_2]_1, \{\{\mathcal{X}S\}\}, \emptyset, \emptyset, R_1 \cup R'_1, R_2 \cup R'_2, R_3 \cup R'_3).$$

The set of objects of  $\Pi$  will contain new special symbols per each rule of  $G$  and is constructed in the following way:

$$\begin{aligned} V = & \{M_i, \bar{Y}_i, M'_i \mid i : X \rightarrow bY \in P\} \\ & \cup \{M_i, N_i, \bar{Y}_i, M'_i \mid i : X \rightarrow Yb \in P\} \\ & \cup \{K, K', \mathcal{X}\} \cup N \cup T. \end{aligned}$$

For each  $i : X \rightarrow bY \in R$ , we construct the following three sets of rules:

$$\begin{aligned} R_1^i &= \{i.1 : (\lambda, M_i, \lambda; in)_{ins}\}, \\ R_2^i &= \{i.2 : (M_i, \bar{Y}_i, \lambda; here)_{ins}\} \cup \{i.3 : (M_i, b, \lambda; in)_{ins}\} \\ &\quad \cup \{i.4 : (b\bar{Y}_i, X, \lambda; out)_{del}\}, \\ R_3^i &= \{i.5 : (\lambda, M_i, \lambda; out)_{del}\}. \end{aligned}$$

For each  $i : X \rightarrow Yb \in R$ , we construct the following three sets of rules:

$$\begin{aligned} R_1^i &= \{i.1 : (\lambda, M_i, \lambda; in)_{ins}\}, \\ R_2^i &= \{i.2 : (M_i, N_i, \lambda; here)_{ins}\} \cup \{i.3 : (N_i, b, \lambda; in)_{ins}\} \\ &\quad \cup \{i.4 : (\bar{Y}_i b, X, \lambda; out)_{del}\}, \\ R_3^i &= \{i.5 : (M_i, \bar{Y}_i, \lambda; here)_{ins}\} \cup \{i.6 : (\lambda, M_i, \lambda; here)_{del}\} \\ &\quad \cup \{i.7 : (\mathbf{a}\bar{Y}_i, N_i, \lambda; out)_{del} \mid \mathbf{a} \in N''\}. \end{aligned}$$

Moreover, we also build the following three sets:

$$\begin{aligned} R_1' &= \{i_1' : (\lambda, M_i', \lambda; in)_{ins} \mid i : X \rightarrow bY \in P \text{ or } i : X \rightarrow Yb \in P\}, \\ R_2^i &= \{i_2' : (M_i', Y, \lambda; in)_{ins} \mid i : X \rightarrow bY \in P \text{ or } i : X \rightarrow Yb \in P\} \\ &\quad \cup \{i_3' : (\lambda, M_i', \lambda; in)_{del} \mid i : X \rightarrow bY \in P \text{ or } i : X \rightarrow Yb \in P\}, \\ R_3^i &= \{i_4' : (M_i'Y, \bar{Y}_i, \lambda; out)_{del} \mid i : X \rightarrow bY \in P \text{ or } i : X \rightarrow Yb \in P\}. \end{aligned}$$

Finally, for the rules  $i_1 : AB \rightarrow \lambda$ ,  $i_2 : CD \rightarrow \lambda$ , and  $i_3 : S' \rightarrow \lambda$  we consider the following sets of rules:

$$\begin{aligned} R_1^{i_1} &= \{i_1.1 : (\lambda, K, \lambda; in)_{ins}\}, \\ R_2^{i_1} &= \{i_1.2 : (K, A, \lambda; in)_{del}\} \cup \{i_1.3 : (\lambda, K, \lambda; out)_{del}\}, \\ R_3^{i_1} &= \{i_1.4 : (K, B, \lambda; out)_{del}\}, \\ R_1^{i_2} &= \{i_2.1 : (\lambda, K', \lambda; in)_{ins}\}, \\ R_2^{i_2} &= \{i_2.2 : (K', C, \lambda; in)_{del}\} \cup \{i_2.3 : (\lambda, K, \lambda; out)_{del}\}, \\ R_3^{i_2} &= \{i_2.4 : (K', D, \lambda; out)_{del}\}, \\ R_1^{i_3} &= \{i_3.1 : (\lambda, S', \lambda; here)_{del}\}. \end{aligned}$$

Now for  $j = 1, 2, 3$  we define the sets  $R_j = \cup_{1 \leq i \leq |P|} R_j^i$  and also  $R_1' = \{\mathcal{X} : (\lambda, \mathcal{X}, \lambda; out)_{del}\}$ .

We state that  $L(\Pi) = L(G)$ . For this we show how each rule of  $G$  can be simulated in  $\Pi$ . Consider a string  $wXw'$  in membrane 1 and suppose that there is a rule  $i : X \rightarrow bY$  in  $P$ . The simulation of this rule occurs in two phases: in the first phase we rewrite  $X$  to  $b\bar{Y}_i$ , while in the second one we substitute  $\bar{Y}_i$  with  $Y$ . The following is the valid first-phase simulation sequence in  $\Pi$ :

$$\begin{aligned} (wXw', 1) &\Rightarrow_{i.1} (wM_iXw', 2) \Rightarrow_{i.2} (wM_i\bar{Y}_iXw', 2) \Rightarrow_{i.3} (wM_i b\bar{Y}_iXw', 3) \\ &\Rightarrow_{i.5} (wb\bar{Y}_iXw, 2) \Rightarrow_{i.4} (wb\bar{Y}_i w', 1). \end{aligned}$$

The second phase happens due to the rules in the sets  $R_i'$ ,  $i = 1, 2, 3$ , and consists of the following steps:

$$\begin{aligned} (w\bar{Y}_i bw', 1) &\Rightarrow_{i'.1} (wM_i' \bar{Y}_i bw', 2) \Rightarrow_{i'.2} (wM_i' Y \bar{Y}_i bw', 3) \\ &\Rightarrow_{i'.4} (wM_i' Y bw', 2) \Rightarrow_{i'.3} (wY bw', 1). \end{aligned}$$

We claim the both the first phase and the second phase simulation sequences are the only ones which can happen in valid derivations of  $\Pi$ . Indeed, consider

the  $wXw'$  into which  $i.1$  has inserted an instance of  $M_i$ . By inspecting the symbol requirements of the rules associated with membrane 2, we conclude that only the rules  $i.2$  and  $i.3$  may become applicable. Suppose that rule  $i.3$  is applied directly. If, for example,  $M_i$  has been inserted to the right of  $X$ , this will produce the string  $\gamma M_i b \gamma'' X w'$ , which will be moved into membrane 3. The case when  $i.1$  inserts  $M_i$  to the left of  $X$  is treated in a similar way. Now, the only way to further move the computation out of membrane 3 is by applying the rule  $i.5$  which will remove the instance of  $M_i$  and move the string into the second membrane. However, no more rules will be applicable from now on, because the string contains no service symbols at all, but is in the second membrane.

Suppose now that, after the application of  $i.1$ , the rule  $i.2$  is applied  $k > 1$  times. The subsequent application of the rule  $i.3$  will insert an instance of  $b$  after  $M_i$ , thus yielding the substring  $M_i b (\bar{Y}_i)^k$ . Again, the only way to move the string out of membrane 3 is to erase the symbol  $M_i$  which produces a string with a substring of  $k$  instances of  $\bar{Y}_i$ . It is clear that, if  $X$  is situated to the left of this  $(\bar{Y}_i)^k$ , the string cannot contain  $\bar{Y}_i X$ , which is required by  $i.3$ . On the other hand, if  $X$  is to the right of  $(\bar{Y}_i)^k$ , it will not be possible to apply  $i.3$  again, because the string does not contain the substring  $\bar{Y}_i X$  preceded by a symbol from  $N''$ .

Finally, it is rather clear that, if  $i.1$  does not insert the  $M_i$  just to the left of  $X$ ,  $\Pi$  will not be able to move the string containing a  $\bar{Y}_i$  and an  $X$  out of membrane 2, thus blocking without producing any meaningful result.

We will focus on the second-phase simulation sequence now. The application of the rule  $i'.1$  inserts an instance of  $M'_i$  somewhere and moves the string into membrane 2. There are only two rules that may become applicable:  $i'.2$  and  $i'.3$ . Suppose that  $i'.3$  is applied directly after  $i'.1$ . In this case the system will come back into the configuration it has been in before the application of  $i'.1$  without doing any changes to the string whatsoever. Therefore, to actually modify the string, the rule  $i'.2$  must be applied.

An application of the rule  $i'.2$  inserts exactly one instance of  $Y$  after  $M'_i$  and puts the string into the innermost membrane 3. Now, the only way to exit this membrane is by applying the rule  $i'.4$ , which means that, if the application of the rule  $i'.1$  has not inserted  $M'_i$  to the left of  $\bar{Y}_i$ , the system  $\Pi$  will unproductively block in the third membrane. Consequently, after the application of  $i'.4$ , the string in the second membrane must be of the form  $wM'_i Y b w'$ . At this point, two rules are still applicable,  $i'.2$  and  $i'.3$ . Suppose indeed that the rule  $i'.2$  is applied a second time and inserts another instance of  $Y$  after  $M'_i$ , thus yielding the string  $wM'_i Y Y b w'$  and moving it into membrane 3. Now, however, the rule  $i'.4$  is not applicable because the string lacks  $\bar{Y}_i$  and  $\Pi$  will thus block. Therefore, the only productive way to move the string  $wM'_i Y b w'$  out of the second membrane is to apply  $i'.3$ .

Now consider a rule  $i : X \rightarrow Yb$ . Again, the simulation of  $i$  happens in two phases: in the first phase we rewrite  $X$  to  $\bar{Y}_i b$ , while in the second phase we substitute  $\bar{Y}_i$  with  $Y$ . Since the second phase of the simulation happens in exactly the same way as in the case of the rule  $X \rightarrow bY$ , we will only focus on

the first-stage simulation sequence:

$$\begin{aligned} (wXw', 1) &\Rightarrow_{i.1} (wM_iXw', 2) \Rightarrow_{i.2} (wM_iN_iXw', 2) \Rightarrow_{i.3} (wM_iN_ibXw', 3) \\ &\Rightarrow_{i.5} (wM_i\bar{Y}_iN_ibXw', 3) \Rightarrow_{i.6} (w\bar{Y}_iN_ibXw', 3) \\ &\Rightarrow_{i.7} (w\bar{Y}_ibXw', 2) \Rightarrow_{i.4} (w\bar{Y}_ibw', 1). \end{aligned}$$

We claim that the first-phase simulation sequence we have just shown is the only possible valid derivation of  $\Pi$ . We will now consider the variations that can interfere with this subderivation and show that none of them can influence the result of a computation of  $\Pi$ .

Consider the application of  $i.1$  which inserts  $M_i$  into the original string  $wXw'$  and moves the new string, say  $\gamma M_i \gamma' Xw'$ , into membrane 2. The case when  $M_i$  is inserted to the right of  $X$  is treated in a similar way. In the current situation, the only applicable rule is  $i.3$ , which may insert  $k$  instances of  $N_i$ , thus yielding the string  $\gamma M_i (N_i)^k \gamma' Xw'$ . If one discards the possibility to produce yet more instances of  $N_i$ , the only other way to evolve is the application of the rule  $i.3$  to insert a  $b$  after one of the  $N_i$ 's and thereby to move the string into membrane 3.

In the new configuration, membrane 3 will contain  $\gamma M_i (N_i)^{k_1} b (N_i)^{k_2} \gamma' Xw'$ , where  $k_1 \geq 1$  and  $k_1 + k_2 = k$ . We immediately remark that the only way for  $\Pi$  to move out of this membrane is to apply the rule  $i.7$ . This rule requires that there is a substring of  $\bar{Y}_i N_i$  preceded by a symbol from  $N''$ . The string  $\gamma M_i (N_i)^{k_1} b (N_i)^{k_2} \gamma' Xw'$ , with which the system  $\Pi$  has just arrived in membrane 3, does not contain any instances of  $\bar{Y}_i$ , but the rule  $i.5$  can introduce them. Suppose this latter rule is applied  $t$  times,  $t \geq 0$ , thus yielding the following result:

$$\gamma M_i (\bar{Y}_i)^t (N_i)^{k_1} b (N_i)^{k_2} \gamma' Xw'.$$

Clearly, the rule  $i.7$  is not yet applicable, because there are no instances of  $\bar{Y}_i$  preceded by symbols from  $N''$ . The only way to reach this situation is to apply the rule  $i.6$  to obtain the string

$$\gamma (\bar{Y}_i)^t (N_i)^{k_1} b (N_i)^{k_2} \gamma' Xw'.$$

The rule  $i.7$  imposes an even stronger requirement: the instance of  $\bar{Y}_i$  which is preceded by a symbol from  $N''$  must be immediately followed by  $N_i$ . Since instances of  $\bar{Y}_i$  can only be inserted to the right of  $M_i$ , and since the process of inserting  $N_i$ 's has already been completed in membrane 2, applying  $i.7$  actually requires that exactly one instance of  $\bar{Y}_i$  has been inserted by  $i.5$  (i.e., it requires that  $t = 1$ ), giving

$$\gamma \bar{Y}_i (N_i)^{k_1} b (N_i)^{k_2} \gamma' Xw'.$$

An application of the rule  $i.7$  will erase the leftmost instance of  $N_i$  and will put the following string into membrane 2:

$$\gamma \bar{Y}_i (N_i)^{k_1-1} b (N_i)^{k_2} \gamma' Xw'.$$

The rule  $i.3$  will still be applicable at this moment. Remark, however, that the string which will be moved into membrane 3 by this application will contain no



instances of  $M_i$ , so the rule which may be applicable is  $i.7$ , which will remove yet another instance of  $N_i$  following  $\bar{Y}_i$ . Applications of the rules  $i.3$  and  $i.7$  in a loop will only be possible as long as there are instances of  $N_i$  just to the right of  $\bar{Y}_i$  and then  $\Pi$  will either block in membrane 3 or move the string into membrane 1 with an application of  $i.4$ .

Based on the observations we have made in the previous paragraph, we can assert that the general form of the strings which may appear in membrane 2 after at least one traversal of membrane 3 is  $\gamma\bar{Y}_i(N_i^*(N_ib)^*)^*\gamma'Xw'$ . If we discard the possibility of yet again re-tracing the loop formed by the rules  $i.3$  and  $i.7$ , the only other way for  $\Pi$  to proceed is to apply  $i.4$  and move the string into membrane 1. However, the rule  $i.4$  imposes a strong condition on the form of the string it can be applied to: there has to exist a substring  $\bar{Y}_ibX$ . Clearly, the only way to have exactly one  $b$  between  $\bar{Y}_i$  and  $X$  is, firstly, to have  $i.1$  insert  $M_i$  exactly to the left of  $X$  (that is,  $\gamma'$  should be zero) and, secondly, to only apply  $i.3$  once during the whole simulation process, thus obtaining the string  $\gamma\bar{Y}_ibXw'$  in membrane 2. The application of  $i.4$  will thus erase the  $X$  and successfully finish the rewriting of  $X$  into  $\bar{Y}_ib$ .

We conclude the proof by stating the simulation of the rules  $AB \rightarrow \lambda$  and  $CD \rightarrow \lambda$  is done in exactly the same way as in the case of the systems from the class  $ELSP_3(ins_1^{2,0}del_1^{1,0})$ .  $\square$

## 4 Conclusion

In this article we considered insertion-deletion P systems of size  $(1, 2, 0; 1, 1, 0)$  and  $(1, 1, 0; 1, 2, 0)$  and showed that computational completeness can be achieved with 3 membranes. Compared to [7] this result shows an interesting trade-off between the size of contexts in insertion-deletion rules and the number of membranes: with 4 membranes, computational completeness is obtained already with insertion and deletion rules of size  $(1, 1, 0)$ . Now it remains an open question if the number of membranes can be further decreased for the investigated systems or for systems having bigger contexts for the insertion or deletion rules.

## References

1. A. Alhazov, A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Small size insertion and deletion systems. In C. Martin-Vide, ed., *Scientific Applications of Language Methods*, volume 2 of *Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory*, chap. 9, 459–524. World Sci., 2010.
2. A. Alhazov, A. Krassovitskiy, Y. Rogozhin, S. Verlan, P Systems with Minimal Insertion and Deletion, *Theoretical Computer Science*, **412**(1-2), 136–144, (2011).
3. R. Benne. *RNA Editing: The Alteration of Protein Coding Sequences of RNA*. Ellis Horwood, Chichester, West Sussex, 1993.
4. F. Biegler, M. J. Burrell, and M. Daley. Regulated RNA rewriting: Modelling RNA editing with guided insertion. *Theor. Comput. Sci.*, 387(2):103 – 112, 2007.
5. E. Csuhaj-Varjú, A. Salomaa, Networks of Parallel Language Processors, *Lecture Notes in Computer Science*, 1218, 299–318, 1997.

6. M. Daley, L. Kari, G. Gloor, and R. Siromoney. Circular contextual insertions/deletions with applications to biomolecular computation. In *SPIRE/CRIWG*, 47–54, 1999.
7. R. Freund, M. Kogler, Y. Rogozhin, and S. Verlan. Graph-controlled insertion-deletion systems. In I. McQuillan and G. Pighizzini, eds, *Proc. of 12<sup>th</sup> Workshop on Descriptive Complexity of Formal Systems*, vol. 31 of *EPTCS*, 88–98, 2010.
8. B. Galiukschov. Semicontextual grammars. *Matem. Logika i Matem. Lingvistika*, 38–50, 1981. Tallin University (in Russian).
9. V. Geffert. Normal forms for phrase-structure grammars. *ITA*, 25:473–498, 1991.
10. D. Haussler. *Insertion and Iterated Insertion as Operations on Formal Languages*. PhD thesis, Univ. of Colorado at Boulder, 1982.
11. D. Haussler. Insertion languages. *Information Sciences*, 31(1):77–89, 1983.
12. L. Kari. *On Insertion and Deletion in Formal Languages*. PhD thesis, University of Turku, 1991.
13. L. Kari, G. Păun, G. Thierrin, and S. Yu. At the crossroads of DNA computing and formal languages: Characterizing RE using insertion-deletion systems. In *Proc. of 3rd DIMACS Workshop on DNA Based Computing*, 318–333. Philadelphia, 1997.
14. S. C. Kleene. Representation of events in nerve nets and finite automata. In C. Shannon and J. McCarthy, eds, *Automata Studies*, 3–41. Princeton University Press, Princeton, NJ, 1956.
15. A. Krassovitskiy. Complexity and Modeling Power of Insertion-Deletion Systems. PhD thesis, Universitat Rovira i Virgili, Tarragona, Spain, 2011.
16. A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Further results on insertion-deletion systems with one-sided contexts. In C. Martín-Vide et al., eds, *Language and Automata Theory and Applications, Second International Conference, LATA 2008. Revised Papers*, vol. 5196 of *LNCS*, 333–344. Springer, 2008.
17. A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of P systems with small size insertion and deletion rules. In T. Neary, D. Woods, A. K. Seda, and N. Murphy, eds, *Proc. International Workshop on The Complexity of Simple Programs, Cork, Ireland, 6-7th December 2008*, vol. 1 of *EPTCS*, 108–117, 2009.
18. A. Krassovitskiy, Y. Rogozhin, and S. Verlan. Computational power of insertion-deletion (P) systems with rules of size two. *Natural Computing*, 10(2), 835–852, 2011.
19. S. Marcus. Contextual grammars. *Rev. Roum. Math. Pures Appl.*, 14:1525–1534, 1969.
20. M. Margenstern, G. Păun, Y. Rogozhin, and S. Verlan. Context-free insertion-deletion systems. *Theor. Comput. Sci.*, 330(2):339–348, 2005.
21. A. Matveevici, Y. Rogozhin, and S. Verlan. Insertion-deletion systems with one-sided contexts. In J. O. Durand-Lose and M. Margenstern, eds, *Machines, Computations, and Universality, 5th International Conference, MCU 2007, Orléans, France, Proceedings*, vol. 4664 of *LNCS*, 205–217. 2007.
22. G. Păun. *Marcus Contextual Grammars*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.
23. G. Păun. *Membrane Computing. An Introduction*. Springer-Verlag, 2002.
24. G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Springer, 1998.
25. G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages*. Springer-Verlag, Berlin, 1997.
26. W. D. Smith. DNA computers in vitro and in vivo. In R. Lipton and E. Baum, editors, *Proceedings of DIMACS Workshop on DNA Based Computers*, DIMACS

- Series in Discrete Math. and Theoretical Computer Science, 121–185. Amer. Math. Society, 1996.
27. A. Takahara and T. Yokomori. On the computational power of insertion-deletion systems. In M. Hagiya and A. Ohuchi, eds, *Proc. of 8th International Workshop on DNA Based Computers, Revised Papers*, vol. 2568 of *LNCS*, 269–280. 2002.
  28. S. Verlan. On minimal context-free insertion-deletion systems. *Journal of Automata, Languages and Combinatorics*, 12(1-2):317–328, 2007.
  29. S. Verlan. Study of language-theoretic computational paradigms inspired by biology. Habilitation thesis, University of Paris Est, 2010.



# Flattening and Simulation of Asynchronous Divisionless P Systems with Active Membranes

Alberto Leporati<sup>1</sup>, Luca Manzoni<sup>2</sup>, and Antonio E. Porreca<sup>1</sup>

<sup>1</sup> Dipartimento di Informatica, Sistemistica e Comunicazione  
Università degli Studi di Milano-Bicocca  
Viale Sarca 336/14, 20126 Milano, Italy  
{leporati,porreca}@disco.unimib.it

<sup>2</sup> Laboratoire i3S, Université Nice Sophia Antipolis,  
CS 40121 – 06903 Sophia Antipolis CEDEX, France  
luca.manzoni@i3s.unice.fr

**Abstract.** We prove that asynchronous P systems with active membranes without division rules can be simulated by single-membrane transition P systems using cooperative rules, even if the synchronisation mechanisms provided by electrical charges and membrane dissolution are exploited. In turn, the latter systems can be simulated by means of place/transition Petri nets, and hence are computationally weaker than Turing machines.

## 1 Introduction

P systems with active membranes [10] are parallel computation devices inspired by the structure and functioning of biological cells. A tree-like hierarchical structure of membranes divides the space into regions, where *multisets* of objects (representing chemical substances) are located. The systems evolve by means of rules rewriting or moving objects, and possibly changing the membrane structure itself (by dissolving or dividing membranes) or the state of the membranes (by changing their electrical charge).

Under the *maximally parallel* updating policy, whereby all components of the system that can evolve concurrently during a given computation step are required to do so, these devices are known to be computationally universal. Alternative updating policies have also been investigated. In particular, *asynchronous* P systems with active membranes [7], where any, not necessarily maximal, number of non-conflicting rules may be applied in each computation step, have been proved able to simulate partially blind register machines [8], computation devices equivalent under certain acceptance conditions to place/transition Petri nets and vector addition systems [11]. This simulation only requires object evolution (rewriting) rules and communication rules (moving objects between regions).

In an effort to further characterise the effect of asynchronicity on the computational power of P systems, we prove that asynchronous P systems with active membranes without dissolution can be flattened if we allow the use of cooperative

rules, obtaining a system that can be easily simulated by place/transition Petri nets, and as such they are not computationally equivalent to Turing machines: indeed, the reachability of configurations and the deadlock-freeness (i.e., the halting problem) of Petri nets are decidable [2]. This holds even when membrane dissolution, which provides an additional synchronisation mechanism (besides electrical charges) whereby all objects are released simultaneously from the dissolving membrane, is employed by the P system being simulated. Unfortunately, this result does not seem to immediately imply the equivalence with partially blind register machines, as the notion of acceptance for Petri nets employed here is by halting and not by placing a token into a “final” place [8].

The paper is organised as follows: in Section 2 we recall the relevant definitions of P systems with active membranes and place/transition Petri nets; in Section 3 we prove that asynchronous P systems with active membranes are computationally equivalent to their *sequential* version, where a single rule is applied during each computation step; in Section 4 we show that sequential P systems with dissolution rules can be simulated by sequential transition P systems with cooperative rules having only one membrane; finally, in Section 5 we show how sequential single-membrane transition P systems using cooperative rules can be simulated by Petri nets. Section 6 contains our conclusions and some open problems.

## 2 Definitions

We first recall the definition of P systems with active membranes and its various operating modes.

**Definition 1.** A P system with active membranes of initial degree  $d \geq 1$  is a tuple  $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \dots, w_{h_d}, R)$ , where:

- $\Gamma$  is an alphabet, i.e., a finite nonempty set of objects;
- $\Lambda$  is a finite set of labels for the membranes;
- $\mu$  is a membrane structure (i.e., a rooted unordered tree) consisting of  $d$  membranes injectively labelled by elements of  $\Lambda$ ;
- $w_{h_1}, \dots, w_{h_d}$ , with  $h_1, \dots, h_d \in \Lambda$ , are strings over  $\Gamma$ , describing the initial multisets of objects located in the  $d$  regions of  $\mu$ ;
- $R$  is a finite set of rules.

Each membrane possesses, besides its label and position in  $\mu$ , another attribute called *electrical charge*, which can be either neutral (0), positive (+) or negative (−) and is always neutral before the beginning of the computation.

The following four kinds of rules are employed in this paper.

- *Object evolution rules*, of the form  $[a \rightarrow w]_h^\alpha$   
They can be applied inside a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the object  $a$  is rewritten into the multiset  $w$  (i.e.,  $a$  is removed from the multiset in  $h$  and replaced by every object in  $w$ ).

- *Send-in communication rules*, of the form  $a [ ]_h^\alpha \rightarrow [b]_h^\beta$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and such that the external region contains an occurrence of the object  $a$ ; the object  $a$  is sent into  $h$  becoming  $b$  and, simultaneously, the charge of  $h$  is changed to  $\beta$ .
- *Send-out communication rules*, of the form  $[a]_h^\alpha \rightarrow [ ]_h^\beta b$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the object  $a$  is sent out from  $h$  to the outside region becoming  $b$  and, simultaneously, the charge of  $h$  is changed to  $\beta$ .
- *Dissolution rules*, of the form  $[a]_h^\alpha \rightarrow b$   
They can be applied to a membrane labeled by  $h$ , having charge  $\alpha$  and containing an occurrence of the object  $a$ ; the membrane  $h$  is dissolved and its contents are released in the surrounding region unaltered, except that an occurrence of  $a$  becomes  $b$ .

We recall that the most general form of P systems with active membranes [10] also includes *membrane division rules*, which duplicate a membrane and its contents; however, these rules are not used in this paper.

Each instantaneous configuration of a P system with active membranes is described by the current membrane structure, including the electrical charges, together with the multisets located in the corresponding regions. A computation step changes the current configuration according to the following set of principles:

- Each object and membrane can be subject to at most one rule per step, except for object evolution rules (inside each membrane several evolution rules having the same left-hand side, or the same evolution rule can be applied simultaneously; this includes the application of the same rule with multiplicity).
- When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step.
- In each computation step, all the chosen rules are applied simultaneously (in an atomic way). However, in order to clarify the operational semantics, each computation step is conventionally described as a sequence of micro-steps as follows. First, all evolution rules are applied inside the elementary membranes, followed by all communication and dissolution rules involving the membranes themselves; this process is then repeated to the membranes containing them, and so on towards the root (outermost membrane). In other words, the membranes evolve only after their internal configuration has been updated. For instance, before a membrane dissolution occurs, all chosen object evolution rules must be applied inside it; this way, the objects that are released outside during the dissolution are already the final ones.
- The outermost membrane cannot be dissolved, and any object sent out from it cannot re-enter the system again.

In the *maximally parallel* mode, the multiset of rules to be applied at each step must be maximal, in the sense that no further rule can be added without creating

conflicts. In the *asynchronous* mode, any nonempty multiset of applicable rules can be chosen. Finally, in the *sequential* mode, exactly one rule per computation step is applied. In the following, only the latter two modes will be considered.

A *halting computation* of the P system  $\Pi$  is a finite sequence of configurations  $\mathcal{C} = (\mathcal{C}_0, \dots, \mathcal{C}_n)$ , where  $\mathcal{C}_0$  is the initial configuration, every  $\mathcal{C}_{i+1}$  is reachable from  $\mathcal{C}_i$  via a single computation step, and no rule can be applied in  $\mathcal{C}_n$ . A *non-halting computation*  $\mathcal{C} = (\mathcal{C}_i : i \in \mathbb{N})$  consists of infinitely many configurations, again starting from the initial one and generated by successive computation steps, where the applicable rules are never exhausted.

The other model of computation we will employ is Petri nets. In particular, with this term we denote place/transition Petri nets with weighted arcs, self-loops and places of unbounded capacity [4]. A Petri net  $N$  is a triple  $(P, T, F)$  where  $P$  is the set of *places*,  $T$  the set of *transitions* (disjoint from  $P$ ) and  $F \subseteq (P \times T) \cup (T \times P)$  is the *flow relation*. The arcs are weighted by a function  $w : F \rightarrow (\mathbb{N} - \{0\})$ . A *marking* (i.e., a configuration) is a function  $M : P \rightarrow \mathbb{N}$ . Given two markings  $M, M'$  of  $N$  and a transition  $t \in T$  we say that  $M'$  is reachable from  $M$  via the firing of  $t$ , in symbols  $M \rightarrow_t M'$ , if and only if:

- for all places  $p \in P$ , if  $(p, t) \in F$  and  $(t, p) \notin F$  then  $M(p) \geq w(p, t)$  and  $M'(p) = M(p) - w(p, t)$ ;
- for all  $p \in P$ , if  $(t, p) \in F$  and  $(p, t) \notin F$  then  $M'(p) = M(p) + w(t, p)$ ;
- for all  $p \in P$ , if both  $(p, t) \in F$  and  $(t, p) \in F$  then  $M(p) \geq w(p, t)$  and  $M'(p) = M(p) - w(p, t) + w(t, p)$ .

Petri nets are nondeterministic devices, hence multiple markings may be reachable from a given configuration. We call *halting computation* a sequence of markings  $(M_0, \dots, M_n)$  where  $M_0 \rightarrow_{t_1} M_1 \rightarrow_{t_2} \dots \rightarrow_{t_n} M_n$  for some  $t_1, \dots, t_n$ , and no transition may fire in  $M_n$ . Several problems related to the reachability of markings and halting configurations (or *deadlocks*) are decidable [2].

### 3 Asynchronicity and Sequentiality

In this section we show how it is possible to construct, for every asynchronous P system with active membranes, a *sequential* version that is equivalent to the original one in the sense that they both halt on the same inputs and produce the same outputs.

The main idea is that each asynchronous step where more than one rule is applied can be substituted by a sequence of asynchronous steps where the rules are reordered and applied one at a time.

**Proposition 1.** *Let  $\Pi$  be a P system with active membranes using object evolution, communication, and dissolution rules. Then, the asynchronous and the sequential updating policies of  $\Pi$  are equivalent in the following sense: for each asynchronous (resp., sequential) computation step  $\mathcal{C} \rightarrow \mathcal{D}$  we have a series of sequential (resp., asynchronous) steps  $\mathcal{C} = \mathcal{C}_0 \rightarrow \dots \rightarrow \mathcal{C}_n = \mathcal{D}$  for some  $n \in \mathbb{N}$ .*



*Proof.* Every asynchronous computation step  $\mathcal{C} \rightarrow \mathcal{D}$  consists in the application of a finite multiset of rules  $\{e_1, \dots, e_p, c_1, \dots, c_q, d_1, \dots, d_r\}$ , where  $e_1, \dots, e_p$  are object evolution rules,  $c_1, \dots, c_q$  are communication rules (either send-in or send-out), and  $d_1, \dots, d_r$  are dissolution rules.

Since evolution rules do not change any charge nor the membrane structure itself, the computation step  $\mathcal{C} \rightarrow \mathcal{D}$  can be decomposed into two asynchronous computation steps  $\mathcal{C} \rightarrow \mathcal{E} \rightarrow \mathcal{D}$ , where the step  $\mathcal{C} \rightarrow \mathcal{E}$  consists in the application of the evolution rules  $\{e_1, \dots, e_p\}$ , and the step  $\mathcal{E} \rightarrow \mathcal{D}$  in the application of the remaining rules  $\{c_1, \dots, c_q, d_1, \dots, d_r\}$ . Notice that in  $\mathcal{E}$  there still exist enough objects to apply these communication and dissolution rules, since by hypothesis  $\mathcal{C} \rightarrow \mathcal{D}$  is a valid computation step.

Furthermore, notice how there is no conflict between object evolution rules (once they have been assigned to the objects they transform). Therefore, the application of the rules  $\{e_1, \dots, e_p\}$  can be implemented as a series of sequential steps  $\mathcal{C} = \mathcal{C}_0 \rightarrow \dots \rightarrow \mathcal{C}_p = \mathcal{E}$ .

Each membrane can be subject to at most a single rule of communication or dissolution type in the computation step  $\mathcal{C} \rightarrow \mathcal{D}$ ; hence, applying one of these rules does not interfere with any other. Thus, these rules can also be serialised into sequential computation steps  $\mathcal{E} \rightarrow \mathcal{C}_{p+1} \rightarrow \dots \rightarrow \mathcal{C}_{p+q+r} = \mathcal{D}$ . Once again, all rules remain applicable since they were in the original computation step.

By letting  $n = p + q + r$ , the first half of the proposition follows. The second part is due to the fact that every sequential computation step is already an asynchronous computation step.  $\square$

## 4 Single-Membrane Transition P Systems

In this section we recall the notion of *transition P system*, imposing as an additional constraint that the system has only one membrane. For a description of a general framework in which these systems can be described see [6]. As proved in [5], these systems are not universal; indeed, a simple simulation by means of Petri nets, inspired by [3], is provided in the next section. Our simulation involves a flattening of the membrane structure and the use of cooperative rules; the first simulation of this type was presented in [1] and, in fact, our construction is similar. Unlike that construction, however, the semantics that we use is sequential and we do not include promoters and inhibitors.

**Definition 2.** A *single-membrane transition P system* is a structure

$$\Pi = (\Gamma, w, R)$$

where  $\Gamma$  is a finite alphabet,  $w$  is a multiset of elements representing the initial state of the system, and  $R$  is a set of cooperative rules in the form  $v \rightarrow w$  where  $v$  and  $w$  are multisets of objects of  $\Gamma$ .

Notice that the definition is a simplified version of the original definition of transition P systems [9], since specifying the membrane structure is not needed.

We can now show that single-membrane transition P systems are equivalent to divisionless P systems with active membranes when operating under the sequential semantics.

Let  $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \dots, w_{h_d}, R)$  be a P system with active membranes and  $\mathcal{C}$  a configuration of  $\Pi$ . The *flattened encoding* of  $\mathcal{C}$  is the multiset  $E(\mathcal{C})$  over  $(\Gamma \cup \{-, 0, +\}) \times \Lambda$  defined as follows:

1. If there are  $n$  copies of the object  $a$  contained in a membrane  $h$  in  $\mathcal{C}$ , then  $E(\mathcal{C})$  contains  $n$  copies of the element  $(a, h)$ .
2. If a membrane  $h$  has charge  $c$ , then the object  $(c, h)$  is in  $E(\mathcal{C})$ .

It is easy to see that, for a fixed  $\Pi$ , the encoding function is a bijection between the configurations of  $\Pi$  and its image, that is, the function  $E$  is invertible. Hence, for any multiset  $A$  that is the encoding of some configuration, the decoding is uniquely identified, i.e., for any configuration  $\mathcal{C}$ ,  $E^{-1}(E(\mathcal{C})) = \mathcal{C}$ .

**Proposition 2.** *Let  $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \dots, w_{h_d}, R)$  be a P system with active membranes working in the sequential mode and using object evolution, communication, and dissolution rules, with initial configuration  $\mathcal{C}_0$ . Then, there exists a single-membrane transition P system  $\Pi' = ((\Gamma \cup \{-, 0, +\}) \cup \{\bullet\}) \times \Lambda, v, R'$ , for some initial multiset  $v$ , working in the sequential mode, such that:*

- (i) *If  $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_m)$  is a halting computation of  $\Pi$ , then there exists a halting computation  $\mathcal{D} = (E(\mathcal{C}_0), \mathcal{D}_1, \dots, \mathcal{D}_n)$  of  $\Pi'$  such that  $\mathcal{D}_n$  is the union of  $E(\mathcal{C}_m)$  and the set of all the elements in the form  $(\bullet, h)$  where  $h$  is a membrane that has been dissolved in  $\mathcal{C}$ .*
- (ii) *If  $\mathcal{D} = (E(\mathcal{C}_0), \mathcal{D}_1, \dots, \mathcal{D}_n)$  is a halting computation of  $\Pi'$ , then there exists a halting computation  $\mathcal{C} = (\mathcal{C}_0, \mathcal{C}_1, \dots, \mathcal{C}_m)$  of  $\Pi$  such that  $\mathcal{D}_n$  can be written as the union of the set of elements in the form  $(\bullet, h)$ , where  $h$  is a membrane that was dissolved in  $\mathcal{C}$ , and the set  $E(\mathcal{C}_m)$ .*
- (iii)  *$\Pi$  admits a non-halting computation  $(\mathcal{C}_0, \mathcal{C}_1, \dots)$  if and only if  $\Pi'$  admits a non-halting computation  $(E(\mathcal{C}_0), \mathcal{D}_1, \dots)$ .*

*Proof.* The main idea is to replace every dissolution rule of a membrane  $h$  in  $R$  with a cooperative rule such that an object in the form  $(\bullet, h)$  is generated and all the objects in the form  $(a, h)$  are rewritten to  $(a, h')$ , where  $h'$  is the lowest ancestor of  $h$  in  $\mu$  that has not been dissolved.

Let  $[a]_{h_1}^\alpha \rightarrow b$  be a dissolution rule in  $R$ . Then,  $R'$  contains the following cooperative rules:

$$(a, h_1)(\alpha, h_1) \rightarrow (b, h_1)(\bullet, h_1). \quad (1)$$

The objects that have  $h_1$  as the second component are then rewritten by means of the following rules:

$$(a, h_1)(\bullet, h_1) \rightarrow (a, h_2)(\bullet, h_1) \quad (2)$$

where  $h_2$  is the parent membrane of  $h_1$  in  $\mu$ . Notice that, if  $(\bullet, h_2)$  exists, then membrane  $h_2$  has been dissolved during a previous computation step; this means

that there exists another rule of type (2) rewriting all the objects having  $h_2$  as the second component. This process continues as long as there are objects with the label of a dissolved membrane as their second component (excluding the ones having  $\bullet$  as the first component).

An object evolution rule  $[a \rightarrow w]_h^\alpha$  is simulated by the following cooperative rule:

$$(a, h)(\alpha, h) \rightarrow (w_1, h) \dots (w_n, h)(\alpha, h). \quad (3)$$

A send-out communication rule  $[a]_{h_1}^\alpha \rightarrow []_{h_1}^\beta b$  is replaced by the following rules:

$$(a, h_1)(\alpha, h_1) \rightarrow (b, h_2)(\beta, h_1) \quad (4)$$

where  $h_2$  is the parent membrane of  $h_1$  in  $\mu$ . As mentioned before, if  $(\bullet, h_2)$  exists, then a rule of type (2) will subsequently rewrite  $(b, h_2)$ .

Finally, a send-in communication rule  $a []_{h_1}^\alpha \rightarrow [b]_{h_1}^\beta$  is simulated as follows. Let  $(h_n, h_{n-1}, \dots, h_2, h_1)$  be a sequence of nested membranes surrounding  $h_1$ , i.e., a descending path in the membrane tree  $\mu$ . For every such sequence, we add the following rules to  $R'$ :

$$(\bullet, h_{n-1}) \dots (\bullet, h_2)(\alpha, h_1)(a, h_n) \rightarrow (\bullet, h_{n-1}) \dots (\bullet, h_2)(\beta, h_1)(b, h_1). \quad (5)$$

These rules rewrite the object  $(a, h_n)$  into  $(b, h_1)$  if in  $\Pi$  all the membranes between  $h_n$  and  $h_1$  have been dissolved. Observe that the number of descending paths leading to  $h_1$  is bounded above by the depth of  $\mu$ .

Notice how every rule of  $R'$  is exactly of one type among (1)–(5); in particular, given a rule in  $R'$  of type (1), (3), (4), or (5), it is always possible to reconstruct the original rule in  $R$ .

Each computation step of  $\Pi$  consisting in the application of an evolution or send-in communication rule is simulated by a single computation step of  $\Pi'$  by means of a rule of type (3) or (5), respectively.

The dissolution of a membrane  $h_1$  in  $\Pi$  requires a variable number of steps of  $\Pi'$ : first, a rule of type (1) is applied, then each object in the form  $(a, h_1)$  must be rewritten, by using rules of type (2), in order to obtain an object in the form  $(a, h_n)$ , where  $h_n$  is the lowest ancestor membrane of  $h_1$  that has not been dissolved in the original system. The exact number of steps depends on the number of objects located inside  $h_1$  and the number of membranes that have been dissolved. The reasoning is analogous for send-out communication rules, simulated by means of rules of type (4) and (2).

Part (i) of the proposition directly follows from the semantics of the above cooperative rules.

Now let  $\mathcal{D} = (\mathcal{D}_0 = E(\mathcal{C}_0), \mathcal{D}_1, \dots, \mathcal{D}_n)$  be a halting computation of  $\Pi'$ . Then there exists a sequence of rules  $\mathbf{r} = (r_1, \dots, r_n)$  in  $R'$  such that

$$\mathcal{D}_0 \rightarrow_{r_1} \mathcal{D}_1 \rightarrow_{r_2} \dots \rightarrow_{r_{n-1}} \mathcal{D}_{n-1} \rightarrow_{r_n} \mathcal{D}_n$$

where the notation  $\mathcal{X} \rightarrow_r \mathcal{Y}$  indicates that configuration  $\mathcal{Y}$  is reached from  $\mathcal{X}$  by applying the rule  $r$ . Let  $f: \mathbb{N} \rightarrow \mathbb{N}$  be defined as

$$f(t) = |\{r_i : 1 \leq i \leq t \text{ and } r_i \text{ is not of type (2)}\}|.$$

We claim that there exists a sequence of rules  $\mathbf{s} = (s_1, \dots, s_m)$  such that the computation  $\mathcal{C} = (\mathcal{C}_0, \dots, \mathcal{C}_m)$  of  $\Pi$  generated by applying the rules of  $\mathbf{s}$ , i.e.,

$$\mathcal{C}_0 \rightarrow_{s_1} \mathcal{C}_1 \rightarrow_{s_2} \dots \rightarrow_{s_{m-1}} \mathcal{C}_{m-1} \rightarrow_{s_m} \mathcal{C}_m$$

has the following property  $P(t)$  for each  $t \in \{0, \dots, n\}$ :

For all  $h \in A$  and  $a \in \Gamma$ , if  $(\gamma, h)$  with  $\gamma \in \{+, 0, -\}$  is in configuration  $\mathcal{D}_t$  of  $\Pi'$ , then the number of copies of the objects of the form  $(a, h')$  with  $h'$  any descendant of  $h$  in  $\mu$ , or  $h$  itself, is equal to the number of copies of  $a$  contained in the membrane substructure rooted in  $h$  in  $\mathcal{C}_{f(t)}$ , and  $h$  has the charge  $\gamma$ . If  $(\bullet, h)$  is in  $\mathcal{D}_t$ , then  $h$  does not appear in  $\mathcal{C}_{f(t)}$  (having been dissolved before).

We prove this property by induction on  $t$ . The case  $t = 0$  clearly holds, by the definition of the encoding function:  $E(\mathcal{C}_{f(0)}) = E(\mathcal{C}_0) = \mathcal{D}_0$ , as  $f(0) = |\emptyset|$ .

Now suppose  $P(t)$  holds for some  $t < n$ . If  $r_{t+1}$  is a rule of type (2) then for each object  $a \in \Gamma$ , the only change in the objects with  $a$  as the first component is when the second component  $h$  is the label of a membrane that has been dissolved in  $\Pi$  and the objects retain  $a$  as the first component while the second one became the label of the parent membrane of  $h$  in  $\mu$ . Furthermore, no symbol in the form  $(\gamma, h)$ , where  $\gamma$  is a charge, is rewritten to a different symbol. Since  $r_{t+1}$  is of type (2), we have  $f(t+1) = f(t)$  hence  $\mathcal{C}_{f(t+1)} = \mathcal{C}_{f(t)}$ , and property  $P(t+1)$  holds.

On the other hand, if  $r_{t+1}$  is not of type (2), then  $f(t+1) = f(t) + 1$  by definition. Let  $s_{f(t)+1} = s_{f(t+1)}$  be the rule corresponding to the cooperative rule  $r_{t+1}$  as described above (an object evolution rule if  $r_{t+1}$  is of type (3), a dissolution rule if  $r_{t+1}$  is of type (1), and so on). Observe that if  $r_{t+1}$  is applicable in  $\mathcal{D}_t$ , then  $s_{f(t)+1}$  is applicable in  $\mathcal{C}_{f(t)}$  by induction hypothesis:

- if  $(\gamma, h)$  is in  $\mathcal{D}_t$  then the membrane  $h$  has charge  $\gamma$  in  $\mathcal{C}_{f(t)}$ ;
- if  $r_{t+1}$  is of type (1), (3), or (4) and uses an object  $(a, h)$  in  $\mathcal{D}_t$ , then a copy of  $a$  appears in membrane  $h$  in  $\mathcal{C}_{f(t)}$ ;
- if  $r_{t+1}$  is of type (5) and uses an object  $(a, h)$  and  $(\bullet, h)$  is in  $\mathcal{D}_t$ , then the object  $a$  appears in  $\mathcal{C}_{f(t)}$  inside the membrane having the same label as the lowest ancestor of  $h$  in the original membrane structure such that  $(\gamma, h)$  with  $\gamma \neq \bullet$  is in  $\mathcal{D}_t$ .

The configuration  $\mathcal{C}_{f(t)+1}$  such that  $\mathcal{C}_{f(t)} \rightarrow_{s_{f(t)+1}} \mathcal{C}_{f(t)+1}$ , due to the semantics of the corresponding rules applied by  $\Pi$  and  $\Pi'$ , is such that the property  $P(t+1)$  holds.

In particular,  $P(n)$  holds: configurations  $\mathcal{D}_n$  and  $\mathcal{C}_{f(n)}$  have the following properties: the encoding  $E(\mathcal{C}_{f(n)})$  is contained in  $\mathcal{D}_n$  and all other objects non

contained in  $E(\mathcal{C}_{f(n)})$  are in the form  $(\bullet, h)$ , where  $h$  is the label of a membrane that has been dissolved during the computation. Notice that  $\mathcal{C}_{f(n)}$  is a halting configuration, since otherwise any rule applicable from it could be simulated from  $\mathcal{D}_n$  as in statement (i). Furthermore, if an object  $(\bullet, h)$  is in  $\mathcal{D}_n$  then no object in form  $(a, h)$  with  $a \in \Gamma$  exists, otherwise further rules of type (2) could be applied, contradicting the hypothesis that  $\mathcal{D}_n$  is a halting configuration. For all membranes  $h$  in  $\mathcal{C}_{f(n)}$  and for all objects  $a \in \Gamma$ , the number of copies of  $a$  that are inside the membrane  $h$  in  $\mathcal{C}_{f(n)}$  is equal to the number of objects in the form  $(a, h)$  in  $\mathcal{D}_n$ , and statement (ii) follows.

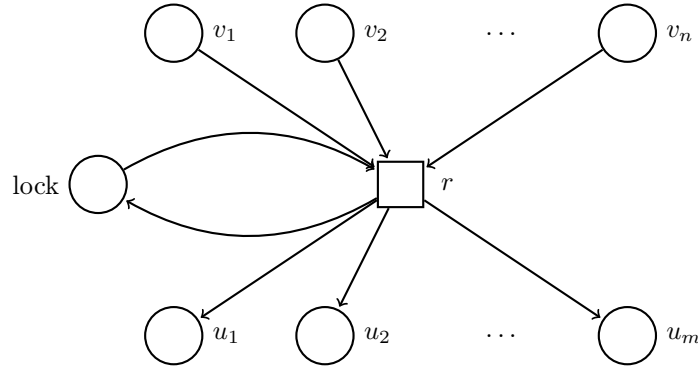
Finally, let us consider a non-halting computation of  $\Pi$ . Each time a computation of  $\Pi$  can be extended by one step by applying a rule, that rule can be simulated by  $\Pi'$  using the same argument employed to prove statement (i), thus yielding a non-halting computation of  $\Pi'$ . Vice versa, in a non-halting computation of  $\Pi'$  it is never the case that infinitely many rules of type (2) are applied sequentially, as only finitely many objects exist at any given time, and eventually they are rewritten to have the form  $(a, h)$  without also having the object  $(\bullet, h)$ . As soon as a rule of type (1), (3), (4), or (5) is applied, the corresponding rule can also be applied by  $\Pi$ , thus yielding a non-halting computation.  $\square$

## 5 Simulation with Petri Nets

The single-membrane transition P systems described in the last section can be simulated by Petri nets in a straightforward way. The idea of using Petri nets as a device for the simulation is originally due to [3].

**Proposition 3.** *Let  $\Pi = (\Gamma, w, R)$  be a single-membrane sequential transition P system. Then, there exists a Petri net  $N$ , having  $\Gamma$  among its places, such that  $\mathcal{C} \rightarrow \mathcal{C}'$  is a computation step of  $\Pi$  if and only if  $M \rightarrow M'$  is a computation step of  $N$ , where  $M(a)$  is the number of instances of  $a$  in  $\mathcal{C}$ .*

*Proof.* The set of places of  $N$  is defined as  $\Gamma \cup \{\text{lock}\}$ , where lock is a place always containing a single token that is employed in order to ensure the firing of at most one transition per step. For each cooperative rule  $v_1 \cdots v_n \rightarrow u_1 \cdots u_m$  the net has a transition defined as follows:



Notice that the output places need not be distinct, as the multiset in the left hand side may contain multiple occurrences of the same symbol; in that case, a weighted arc is used. The output places need not be distinct from the input places either; in that case, the net contains a corresponding loop.

The initial marking  $M_0$  of  $N$  is given by  $M_0(a) = |w|_a$ , for all  $a \in \Gamma$ , where  $|w|_a$  is the multiplicity of  $a$  in  $w$ .

Notice that a transition  $r$  in  $N$  is enabled exactly when the corresponding rule  $r \in R$  is applicable, producing a transition  $M \rightarrow_r M'$  corresponding to a computation step  $\mathcal{C} \rightarrow_r \mathcal{C}'$  of  $\Pi$  as required. In every moment the number of tokens in a place is equal to the multiplicity of the corresponding object in the configuration of  $\Pi$ .  $\square$

By combining Propositions 1, 2, and 3, we can finally prove the following theorem.

**Theorem 1.** *For every asynchronous P system with active membranes  $\Pi$  using evolution, communication, and dissolution rules, there exists a Petri net  $N$  such that (i) every halting configuration of  $\Pi$  corresponds to a halting configuration of  $N$  and vice versa (under the encoding of Propositions 2 and 3), and (ii) every non-halting computation of  $\Pi$  corresponds to a non-halting computation of  $N$  and vice versa.*  $\square$

Notice that, given the strict correspondence of computations and their halting configurations (if any) between the two devices, this result holds both for P systems computing functions over multisets/Parikh vectors and those recognising or generating families of multisets/Parikh vectors, since the only difference between these computing modes is the initial configuration and the acceptance condition; these are translated directly into the simulating Petri net.

## 6 Conclusions

We have proved that asynchronous P systems with active membranes (without division rules) can be flattened and simulated by single-membrane transition P systems using cooperative rules. These systems can, in turn, be easily simulated by place/transition Petri nets, and hence are not computationally universal. In order to achieve this result, we exploited the equivalence between the asynchronous and the sequential parallelism policies for divisionless P systems with active membranes.

The conjectured equivalence of asynchronous P systems with active membranes and Petri nets does not seem to follow immediately from our result and the previous simulation of partially blind register machines by means of asynchronous P systems with active membranes [7]. Indeed, an explicit signalling (putting a token into a specified place) instead of accepting by halting seems to be required in order to simulate Petri nets with partially blind register machines [8]. Directly simulating Petri nets with asynchronous P systems with active membranes is also nontrivial, since transitions provide a stronger synchronisation

mechanism than the limited context-sensitivity of the rules of a P system with active membranes. This equivalence is thus left as an open problem.

### Acknowledgements

We would like to thank Luca Bernardinello for his advice on the theory of Petri nets. We would also like to thank the anonymous reviewers for pointing out relevant literature that allowed a simplification of the original construction.

This research was partially funded by Lombardy Region under project NEDD and by the French National Research Agency project EMC (ANR-09-BLAN-0164).

### References

1. Agrigoroaiei, O., Ciobanu, G.: Flattening the transition P systems with dissolution. In: Conference on Membrane Computing, CMC 11. LNCS, vol. 6501, pp. 53–64. Springer (2011)
2. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. *Theoretical Computer Science* 147, 117–136 (1995)
3. Dal Zilio, S., Formenti, E.: On the dynamics of PB systems: A Petri net view. In: Workshop on Membrane Computing, WMC3. pp. 153–167. LNCS, Springer (2004)
4. Desel, J., Reisig, W.: Place/transition Petri nets. In: Reisig, W., Rozenberg, G. (eds.) *Lectures on Petri nets I: Basic models*, *Advances in Petri Nets*, vol. 1491, pp. 122–173. Springer (1998)
5. Freund, R.: Asynchronous P systems and P systems working in the sequential mode. In: Workshop on Membrane Computing, WMC4. LNCS, vol. 3365, pp. 36–62. Springer (2005)
6. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Workshop on Membrane Computing, WMC8. LNCS, vol. 4860, pp. 271–284. Springer (2007)
7. Frisco, P., Govan, G., Leporati, A.: Asynchronous P systems with active membranes. *Theoretical Computer Science* 429, 74–86 (2012)
8. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science* 7, 311–324 (1978)
9. Păun, Gh.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
10. Păun, Gh.: P systems with active membranes: Attacking NP-complete problems. *Journal of Automata, Languages and Combinatorics* 6(1), 75–90 (2001)
11. Peterson, J.L.: *Petri net theory and the modeling of systems*. Prentice-Hall (1981)





# Enzymatic Numerical P Systems Using Elementary Arithmetic Operations

Alberto Leporati, Giancarlo Mauri, Antonio E. Porreca, and Claudio Zandron

Dipartimento di Informatica, Sistemistica e Comunicazione  
Università degli Studi di Milano-Bicocca  
Viale Sarca 336/14, 20126 Milano, Italy  
{leporati,mauri,porreca,zandron}@disco.unimib.it

**Abstract.** We prove that all-parallel enzymatic numerical P systems whose production functions can be expressed as a combination of sums, differences, products and integer divisions characterise **PSPACE** when working in polynomial time. We also show that, when only sums and differences are available, exactly the problems in **P** can be solved in polynomial time. These results are proved by showing how EN P systems and random access machines, running in polynomial time and using the same basic operations, can simulate each other efficiently.

## 1 Introduction

Numerical P systems have been introduced in [8] as a model of membrane systems inspired both from the structure of living cells and from economics. Each region of a numerical P system contains some numerical variables, that evolve from initial values by means of *programs*. Each program consists of a *production function* and a *repartition protocol*; the production function computes an output value from the values of some variables occurring in the same region in which the function is located, while the repartition protocol distributes this output value among the variables in the same region as well as in the neighbouring (parent and children) ones.

In [8], and also in Chapter 23.6 of [9], some results concerning the computational power of numerical P systems are reported. In particular, it is proved that nondeterministic numerical P systems with polynomial production functions characterize the recursively enumerable sets of natural numbers, while deterministic numerical P systems, with polynomial production functions having non-negative coefficients, compute strictly more than semilinear sets of natural numbers.

Enzymatic Numerical P systems (EN P systems, for short) have been introduced in [10] as an extension of numerical P systems in which some variables, named the *enzymes*, control the application of the rules, similarly to what happens in P systems with promoters and inhibitors [2]. As shown in [11, 3] and references therein, the most promising application of EN P systems seems to be the simulation of control mechanisms of mobile and autonomous robots.

The computational power of EN P systems has also been thoroughly investigated. In [6] a short review of previously known universality results is presented, together with an improvement on some of them: linear production functions involving only one variable suffice to obtain universality in the one-parallel and all-parallel modes.

In this paper we deal with computational complexity issues, and show how the choice of arithmetical operations allowed in the production functions influences the efficiency of computation of all-parallel EN P systems, exactly as it happens for random access machines [5]. Indeed, we prove that these two computation devices can simulate each other efficiently in some relevant cases. As a consequence, we show the limitations of linear production functions, and how these are overcome by allowing multiplication and integer division, leading to polynomial time solutions to **PSPACE**-complete problems.

The paper is organised as follows. In Section 2 we recall the definitions of EN P systems and random access machines, together with the relevant results from the literature. In Section 3 we show, as a technical result, how indirect addressing can be eliminated when RAMs operate in polynomial time, thus simplifying the simulation by means of all-parallel EN P systems that is presented in Section 4. The converse simulation is illustrated in Section 5, leading to our main result about the computational complexity of all-parallel EN P systems. Finally, conclusions and open problems are described in Section 6.

## 2 Definitions and Previous Results

An *enzymatic numerical P system* (EN P system, for short) is a construct of the form:

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

where  $m \geq 1$  is the degree of the system (the number of membranes),  $H$  is an alphabet of labels,  $\mu$  is a tree-like membrane structure with  $m$  membranes injectively labeled with elements of  $H$ ,  $Var_i$  and  $Pr_i$  are respectively the set of variables and the set of programs that reside in region  $i$ , and  $Var_i(0)$  is the vector of initial values for the variables of  $Var_i$ . All sets  $Var_i$  and  $Pr_i$  are finite. In the original definition of EN P systems [10] the values assumed by the variables may be real, rational or integer numbers; in what follows we will allow instead only integer numbers. The variables from  $Var_i$  are written in the form  $x_{j,i}$ , for  $j$  running from 1 to  $|Var_i|$ , the cardinality of  $Var_i$ ; the value assumed by  $x_{j,i}$  at time  $t \in \mathbb{N}$  is denoted by  $x_{j,i}(t)$ . Similarly, the programs from  $Pr_i$  are written in the form  $P_{l,i}$ , for  $l$  running from 1 to  $|Pr_i|$ .

The *programs* allow the system to evolve the values of variables during computations. Each program is composed of two parts: a *production function* and a *repartition protocol*. The former can be any function using variables from the region that contains the program. Using the production function, the system computes a *production value*, from the values of its variables at that time. This value is distributed to variables from the region where the program resides, and to

variables in its upper (parent) and lower (children) compartments, as specified by the repartition protocol. Formally, for a given region  $i$ , let  $v_1, \dots, v_{n_i}$  be all these variables; let  $x_{1,i}, \dots, x_{k_i,i}$  be some variables from  $Var_i$ , let  $F_{l,i}(x_{1,i}, \dots, x_{k_i,i})$  be the production function of a given program  $P_{l,i} \in Pr_i$ , and let  $c_{l,1}, \dots, c_{l,n_i}$  be natural numbers. The program  $P_{l,i}$  is written in the following form:

$$F_{l,i}(x_{1,i}, \dots, x_{k_i,i}) \rightarrow c_{l,1}|v_1 + c_{l,2}|v_2 + \dots + c_{l,n_i}|v_{n_i} \quad (1)$$

where the arrow separates the production function from the repartition protocol. Let  $C_{l,i} = \sum_{s=1}^{n_i} c_{l,s}$  be the sum of all the coefficients that occur in the repartition protocol. If the system applies program  $P_{l,i}$  at time  $t \geq 0$ , it computes the value

$$q = \frac{F_{l,i}(x_{1,i}(t), \dots, x_{k_i,i}(t))}{C_{l,i}}$$

that represents the “unitary portion” to be distributed to variables  $v_1, \dots, v_{n_i}$  proportionally with coefficients  $c_{l,1}, \dots, c_{l,n_i}$ . So each of the variables  $v_s$ , for  $1 \leq s \leq n_i$ , will receive the amount  $q \cdot c_{l,s}$ . An important observation is that variables  $x_{1,i}, \dots, x_{k_i,i}$  involved in the production function are reset to zero after computing the production value, while the other variables from  $Var_i$  retain their value. The quantities assigned to each variable from the repartition protocol are added to the current value of these variables, starting with 0 for the variables which were reset by a production function. As pointed out in [12], a delicate problem concerns the issue whether the production value is divisible by the total sum of coefficients  $C_{l,i}$ . As it is done in [12], in this paper we assume that this is the case, and we deal only with such systems; see [8] for other possible approaches.

Besides programs (1), EN P systems may also have programs of the form

$$F_{l,i}(x_{1,i}, \dots, x_{k_i,i})|_{e_{j,i}} \rightarrow c_{l,1}|v_1 + c_{l,2}|v_2 + \dots + c_{l,n_i}|v_{n_i}$$

where  $e_{j,i}$  is a variable from  $Var_i$  different from  $x_{1,i}, \dots, x_{k_i,i}$  and  $v_1, \dots, v_{n_i}$ . Such a program can be applied at time  $t$  only if  $e_{j,i}(t) > \min(x_{1,i}(t), \dots, x_{k_i,i}(t))$ . Stated otherwise, variable  $e_{j,i}$  operates like an *enzyme*, that enables the execution of the program but, as happens with catalysts, it is neither consumed nor modified by the execution of the program. However, in EN P systems enzymes can evolve by means of other programs, that is, enzymes can receive “contributions” from other programs and regions.

A *configuration* of  $\Pi$  at time  $t \in \mathbb{N}$  is given by the values of all the variables of  $\Pi$  at that time; in a compact notation, we can write it as the sequence  $(Var_1(t), \dots, Var_m(t))$ , where  $m$  is the degree of  $\Pi$ . The *initial configuration* can thus be described as the sequence  $(Var_1(0), \dots, Var_m(0))$ . The system  $\Pi$  evolves from an initial configuration to other configurations by means of *computation steps*, in which one or more programs of  $\Pi$  (depending upon the *mode* of computation) are executed. In [12], at each computation step the programs to be executed are chosen in the so called *sequential* mode: one program is nondeterministically chosen in each region, among the programs that can be executed at

that time. Another possibility is to select the programs in the so called *all-parallel* mode: in each region, all the programs that can be executed are selected, with each variable participating in all programs where it appears. Note that in this case EN P systems become *deterministic*, since nondeterministic choices between programs never occur. A variant of parallelism, analogous to the maximal one which is often used in membrane computing, is the so called *one-parallel* mode: in each region, all the programs which can be executed can be selected, but the actual selection is made in such a way that each variable participates in only one of the chosen programs. We say that the system reaches a *final configuration* if and when it happens that no applicable set of programs produces a change in the current configuration.

EN P systems may be used as (polynomial) time-bounded recognising devices as follows. Notice that we use two variables (instead of just one of them), named *accept* and *reject*, to signal the end of computations. This is done because some programs of the system may be applied forever, causing the system to never halt even if the configuration does not change any more. By using two variables, the event of reaching a final configuration is made visible and distinguishable from the outside.

**Definition 1.** Let  $L \subseteq \{0,1\}^*$  be a language, and let  $\Pi$  be a deterministic EN P system with two distinguished variables *accept* and *reject*. We say that  $\Pi$  decides  $L$  in polynomial time iff, for all  $x \in \{0,1\}^*$ , when the integer having binary representation  $1x$  is initially given to a specified input variable<sup>1</sup> the P system  $\Pi$  reaches a final configuration such that

- if  $x \in L$ , then *accept* = 1 and *reject* = 0
- if  $x \notin L$ , then *accept* = 0 and *reject* = 1

within a number of steps bounded by  $O(|x|^k)$  for some  $k \in \mathbb{N}$ .

As proved in [6], every all-parallel and one-parallel EN P system can be “flattened” into an equivalent (both in terms of output and number of computation steps) system having only one membrane. For simplicity, in the following sections we shall always deal with flattened EN P systems.

The proofs in this paper will be based on random access machines [7, 5]. We define the specific variant we will employ:

**Definition 2 (RAM).** A random access machine consists of an infinite number of registers ( $r_i : i \in \mathbb{N}$ ) having values in  $\mathbb{N}$ , initially set to zero, and a finite sequence of instructions injectively labelled by elements  $\ell \in \mathbb{N}$ . The instructions are of the following types:

- assignment of a constant  $k \in \mathbb{N}$ : “ $\ell: r_i := k$ ” ( $r_i$  is assigned a constant value)
- copying a register: “ $\ell: r_i := r_j$ ” ( $r_i$  is assigned the content of a fixed register)
- indirect addressing: “ $\ell: r_i := r_{r_j}$ ” ( $r_i$  is assigned the content of a register whose number is given by a fixed register)

<sup>1</sup> The “1” is prefixed to the input string  $x$  in order to keep the leading zeroes.

- arithmetic operations, with  $\bullet \in \{+, -, \times, \div\}$ : “ $\ell: r_i := r_j \bullet r_k$ ”
- conditional jump, with  $\ell_1, \ell_2 \in \mathbb{N}$ : “ $\ell: \text{if } r_i \neq 0 \text{ then } \ell_1 \text{ else } \ell_2$ ”
- halt and accept: “ $\ell: \text{accept}$ ”
- halt and reject: “ $\ell: \text{reject}$ ”.

The labels of the instructions will sometimes be left implicit.

We assume, without loss of generality, that it is never the case that a register or a label are mentioned multiple times in the same instruction (e.g., in “ $\ell: r_i := r_j \bullet r_k$ ” we assume  $i \neq j$ ,  $j \neq k$ , and  $i \neq k$ ).

Since RAMs operate on natural numbers, we only allow *non-negative* subtraction, i.e.,  $x - y = 0$  when  $y > x$ .

**Definition 3.** Let  $L \subseteq \{0, 1\}^*$  be a language, and let  $M$  be a RAM. We say that  $M$  decides  $L$  in polynomial time iff, for all  $x \in \{0, 1\}^*$ , when the integer having binary representation  $1x$  is loaded into a specified input register, the machine  $M$  behaves as follows:

- if  $x \in L$ , then  $M$  reaches an “accept” instruction
- if  $x \notin L$ , then  $M$  reaches a “reject” instruction

within a number of steps bounded by  $O(|x|^k)$  for some  $k \in \mathbb{N}$ .

In the rest of this paper we will denote the class of random access machines using the set of basic operations  $X \subseteq \{+, -, \times, \div\}$  by  $\text{RAM}(X)$ , and the class of all-parallel EN P systems whose production functions can be expressed in terms of  $X$  by  $\text{ENP}(X)$ . In particular, we are interested in all-parallel EN P systems having linear production functions,  $\text{ENP}(+, -)$ , and those with production functions consisting of polynomials augmented by integer division,  $\text{ENP}(+, -, \times, \div)$ .

We shall also employ the following notation for complexity classes:

**Definition 4.** Let  $D$  be one of the classes of computing devices described above. Then, by **P-D** we denote the class of decision problems solvable in polynomial time by devices of type  $D$ .

The computational power of polynomial-time RAMs is strictly dependent on the set of basic operations that can be computed in a single time step. When only addition and subtraction are available, then polynomial-time RAMs are equivalent to polynomial-time Turing machines [4].

**Proposition 1.**  $\text{P-RAM}(+, -) = \text{P}$ . □

On the other hand, multiplication and division considerably increase the efficiency of polynomial-time RAMs [1]:

**Proposition 2.**  $\text{P-RAM}(+, -, \times, \div) = \text{PSPACE}$ . □

```

1  $e := y$ 
2  $z := 1$ 
3 while  $e > 0$  do
4    $\{x^e \times z = x^y\}$ 
5    $p := 1$ 
6    $p' := 2$ 
7    $a := x$ 
8    $a' := x \times x$ 
9   while  $p' \leq e$  do
10     $p := p'$ 
11     $p' := p' + p'$ 
12     $a := a'$ 
13     $a' := a' \times a'$ 
14  end
15   $\{e - p \leq e/2\}$ 
16   $e := e - p$ 
17   $z := z \times a$ 
18 end

```

$\left. \begin{array}{l} \text{lines 4-13} \\ \text{lines 9-13} \end{array} \right\} O(\log y) \text{ iterations}$

**Fig. 1.** Polynomial-time exponentiation algorithm by repeated squaring.

### 3 Avoiding Indirect Addressing

In this section we recall how indirect addressing may be eliminated from random access machines by encoding any number of registers as a single large integer. The resulting machine only needs a constant number of registers and, when the original machine runs in polynomial time, the slowdown is only polynomial.

In order to eliminate indirect addressing we employ multiplication, integer division and exponentiation. The first two operations, which are built-in on a  $\text{RAM}(+, -, \times, \div)$ , can be computed in quadratic time by a  $\text{RAM}(+, -)$  using repeated doubling.

**Proposition 3.** *The product  $x \times y$  and the quotient  $x \div y$  can be computed in  $O((\log y)^2)$  time and  $O((\log x)^2)$  time respectively by a  $\text{RAM}(+, -)$  using a constant number of auxiliary registers.  $\square$*

Exponentiation can be also computed in polynomial time, using a repeated squaring algorithm, both by a  $\text{RAM}(+, -)$  and a  $\text{RAM}(+, -, \times, \div)$ .

**Proposition 4.** *The exponential  $x^y$  can be computed in  $O((\log y)^2)$  time by a  $\text{RAM}(+, -, \times)$  and in  $O((y \log y \log x)^2)$  time by a  $\text{RAM}(+, -)$  using a constant number of auxiliary registers.*

*Proof.* The algorithm of Fig. 1 computes  $z := x^y$  by repeated squaring.

The outermost loop maintains the invariant  $x^e \times z = x^y$ , and the innermost loop computes the largest power  $2^i$  less than or equal to  $e$ , which is then subtracted from  $e$ , thus reducing the value of this register by half or more (hence,

eventually, to 0); the product of the values  $x^{2^i}$  is accumulated into  $z$ . In other words, the algorithm computes the value  $x^y$  as

$$\begin{aligned} x^y &= x^{y_m 2^m} \times x^{y_{m-1} 2^{m-1}} \times \dots \times x^{y_1 2^1} \times x^{y_0 2^0} \\ &= x^{y_m 2^m + y_{m-1} 2^{m-1} + \dots + y_1 2^1 + y_0 2^0} \end{aligned}$$

where  $y_m y_{m-1} \dots y_1 y_0$  is the binary expansion of  $y$ .

Each line of the algorithm is performed by a  $\text{RAM}(+, -, \times)$  in constant time, for a total of  $O((\log y)^2)$  time. On a  $\text{RAM}(+, -)$ , the product of line 8 is computed in  $O((\log x)^2)$  time, and the products of lines 13 and 17 in  $O((\log x^y)^2) = O((y \log x)^2)$  time, since  $a$  reaches the value  $x^y$  in the worst case (i.e., when  $y$  is a power of 2). The total time is thus  $O((y \log y \log x)^2)$ .  $\square$

An arbitrary random access machine never uses more registers than time steps; however, in principle, the largest register index employed can be exponential on a  $\text{RAM}(+, -)$ , or even doubly exponential on a  $\text{RAM}(+, -, \times, \div)$ . The following proposition [5] obviates the problem.

**Proposition 5.** *Let  $M$  be a RAM with addition, subtraction and possibly multiplication and division, working in time  $t(n)$ . Then there exists a RAM with the same basic operations working in time  $O(t(n)^2)$ , having the same output as  $M$ , and using only its first  $O(t(n))$  registers.*  $\square$

The three Propositions 3, 4, and 5 allow us to simulate indirect addressing from polynomial-time RAMs with a polynomial slowdown.

**Proposition 6.** *Let  $M_1$  be a  $\text{RAM}(+, -)$  (respectively, a  $\text{RAM}(+, -, \times, \div)$ ) working in polynomial time  $O(n^k)$ . Then, there exists a  $\text{RAM}(+, -)$  (resp., a  $\text{RAM}(+, -, \times, \div)$ )  $M_2$  working  $O(n^{8k}(\log n)^2)$  time (resp.,  $O(n^{2k}(\log n)^2)$ ) and computing the same result as  $M_1$  without using indirect addressing.*

*Proof.* Since  $M_1$  works in polynomial time, by Proposition 5 there exists another RAM  $M'_1$  with the same output as  $M_1$ , working in polynomial time  $t = c_1 n^{2k} + c_0$  and using at most the first  $m = d_1 n^k + d_0$  registers (for some  $c_0, c_1, d_0, d_1 \in \mathbb{N}$ ).

The machine  $M_2$  simulates  $M'_1$  as follows. All the registers  $(r_0, \dots, r_{m-1})$  of  $M'_1$  are stored in a single register  $r$  of  $M_2$  as a base- $b$  number:

$$r = b^{m-1} r_{m-1} + b^{m-2} r_{m-2} + \dots + b^1 r_1 + b^0 r_0.$$

The base  $b$  is one more than the largest number that can ever be stored in a register by  $M'_1$ , which can be computed as follows:

- If  $M'_1$  is a  $\text{RAM}(+, -)$ , the most expensive instruction (in terms of magnitude of the values of the registers) is “ $x := x + x$ ”, where  $x$  is the input register. After  $t$  steps, the value of any register is thus bounded by  $2^t x$ , and we choose  $b = 2^t x + 1$ .

- If  $M'_1$  is a  $\text{RAM}(+, -, \times, \div)$ , then the most expensive instruction is squaring, i.e., “ $x := x \times x$ ”, leading to an upper bound of  $x^{2^t}$  after  $t$  steps. In this case, we choose  $b = x^{2^t} + 1$ .

Notice that  $r$  has an upper bound of  $b^{m+1}$ .

The machine  $M_2$  first computes the length  $n = O(\log x)$  of the input (contained in the register  $x$ ) as follows:

```

1  $y := x$ 
2  $n := 0$ 
3 while  $y \neq 0$  do
4    $y := y \div 2$ 
5    $n := n + 1$ 
6 end
```

This requires  $O(\log x)$  steps on a  $\text{RAM}(+, -, \times, \div)$ , and  $O((\log x)^3)$  steps on a  $\text{RAM}(+, -)$ , due to the cost of the division of line 4.

$M_2$  then computes the number of steps  $t$  of  $M'_1$  to be simulated:

```

7  $t := c_1 n^{2k} + c_0$ 
```

Line 7 can be executed in  $O(1)$  time by a  $\text{RAM}(+, -, \times, \div)$ , since  $k$ ,  $c_0$ , and  $c_1$  are constants; on a  $\text{RAM}(+, -)$  the time is  $O((\log n)^2) = O((\log \log x)^2)$ . Notice that evaluating such complex expressions only requires a constant number of auxiliary registers.

The base  $b$  described above is then computed. For a  $\text{RAM}(+, -)$  the calculation is

```

8  $b := 2^t x + 1$ 
```

which executes in  $O((t \log t)^2 + (\log x)^2) = O((n^{2k} \log n)^2)$  time.

For a  $\text{RAM}(+, -, \times, \div)$  the calculation is

```

8  $b := x^{2^t} + 1$ 
```

which executes in  $O(t^2) = O(n^{4k})$  time.

The last phase of the initialisation of  $M_2$  sets up register  $r$ , which initially contains only  $x$  in its 0-th position:

```

9  $r := x$ 
```

Every time a register of  $M'_1$ , say  $r_i$  (with  $i$  a constant), has to be read, its value can be extracted from the register of  $r$  of  $M_2$  and stored in an auxiliary register, say  $y$ , as follows:

```

 $y := (r \div b^i) \bmod b$ 
```



where  $a \bmod b = a - (a \div b \times b)$ . This requires

$$\begin{aligned} O((\log b)^2 + (\log r)^2) &= O((\log b)^2 + (\log b^{m+1})^2) = O((\log b^{m+1})^2) \\ &= O((m \log b)^2) = O(((d_1 n^k + d_0) \log(2^t x + 1))^2) \\ &= O((n^k(t + \log x))^2) = O(n^{6k}) \end{aligned}$$

time on a  $\text{RAM}(+, -)$ , and  $O(1)$  time on a  $\text{RAM}(+, -, \times, \div)$ .

If indirect access is needed, that is, we read  $r_i$  where  $i < m$  is *not* a constant, then the computation time becomes

$$\begin{aligned} O((i \log i \log b)^2 + (\log r)^2) &= O((m \log m \log b)^2 + (m \log b)^2) \\ &= O((m \log m \log b)^2) = O((n^k \log n \log b)^2) \\ &= O((n^k \log n \cdot (t + \log x))^2) \\ &= O((n^k \log n \cdot n^{2k})^2) = O((n^{3k} \log n)^2) \\ &= O(n^{6k}(\log n)^2) \end{aligned}$$

on a  $\text{RAM}(+, -)$ , and

$$O((\log i)^2) = O((\log m)^2) = O((\log n)^2)$$

on a  $\text{RAM}(+, -, \times, \div)$ . Hence, reading a register of  $M'_1$  (and, in particular, indirect addressing) can be simulated in polynomial time both on a  $\text{RAM}(+, -)$  and on a  $\text{RAM}(+, -, \times, \div)$ .

The operation of writing the value of a register  $y$  of  $M_2$  into a simulated register  $r_i$  of  $M'_1$  is similar:

$$\begin{aligned} 1 \quad z &:= (r \div b^i) \bmod b \\ 2 \quad r &:= r - (z \times b^i) + (y \times b^i) \end{aligned}$$

and has the same asymptotical time complexity as above (keeping in mind that  $i$  is a constant in this case).

We can now finally describe how the instructions of  $M'_1$  are simulated by  $M_2$ .

– Assignment of a constant “ $r_i := c$ ”

$$\begin{aligned} z &:= (r \div b^i) \bmod b \\ r &:= r - (z \times b^i) + (c \times b^i) \end{aligned}$$

– Copying the value of a register “ $r_i := r_j$ ”

$$\begin{aligned} y &:= (r \div b^j) \bmod b \\ z &:= (r \div b^i) \bmod b \\ r &:= r - (z \times b^i) + (y \times b^i) \end{aligned}$$

– Copying the value of a register through indirect addressing “ $r_i := r_{r_j}$ ”

$$\begin{aligned} y &:= (r \div b^j) \bmod b \\ y' &:= (r \div b^y) \bmod b \\ z &:= (r \div b^i) \bmod b \\ r &:= r - (z \times b^i) + (y' \times b^i) \end{aligned}$$

- Arithmetical operations “ $r_i := r_j \bullet r_k$ ” with  $\bullet \in \{+, -\}$  (for a RAM(+, -)) or  $\bullet \in \{+, -, \times, \div\}$  (for a RAM(+, -,  $\times$ ,  $\div$ ))
 
$$y_1 := (r \div b^j) \bmod b$$

$$y_2 := (r \div b^k) \bmod b$$

$$y := y_1 \bullet y_2$$

$$z := (r \div b^i) \bmod b$$

$$r := r - (z \times b^i) + (y \times b^i)$$
- Conditional jump “if  $r_i \neq 0$  then  $\ell_1$  else  $\ell_2$ ”
 
$$y := (r \div b^i) \bmod b$$

$$\text{if } y \neq 0 \text{ then } \ell'_1 \text{ else } \ell'_2$$

where  $\ell'_1$  (resp.,  $\ell'_2$ ) is the label of the first of the instructions of  $M_2$  simulating the instruction  $\ell_1$  (resp.,  $\ell_2$ ) of  $M'_1$ .

The discussion above implies that simulating each instruction of  $M'_1$  requires at most  $O(n^{6k}(\log n)^2)$  for a RAM(+, -), and  $O((\log n)^2)$  for a RAM(+, -,  $\times$ ,  $\div$ ). Hence, the total number of steps to complete the simulation is  $O(n^{8k}(\log n)^2)$  and  $O(n^{2k}(\log n)^2)$  respectively.  $\square$

## 4 Simulating RAMs without Indirect Addressing

We now prove that each RAM, whose instructions satisfy the mild constraints we have imposed in the definition, and do not use indirect addressing, can be simulated by an appropriate EN P system working in the all-parallel mode. The simulation is efficient, in the sense that each RAM instruction is simulated in just one step.

**Theorem 1.** *Let  $M$  be a RAM that does not use indirect addressing. Then, for each instruction of  $M$  there exists a set of programs for an all-parallel EN P system  $\Pi$  that simulates it in one computation step.*

*Proof.* We proceed by examining all possible cases. In what follows,  $z$  is a variable whose value is always zero, variables  $r_i, r_j, r_k$  represent registers of  $M$  (containing non negative integer values), and variables  $p_\ell$  assume values in  $\{0, 1\}$  to indicate the next instruction of  $M$  to be simulated.

RAM instructions of type “ $\ell: r_i := k$ ” can be simulated by the following set of all-parallel programs:

$$0r_i + k + z|_{p_\ell} \rightarrow 1|r_i$$

$$p_\ell \rightarrow 1|p_{\ell+1}$$

When  $p_\ell = 0$  the first program is not executed, while the second program zeroes  $p_\ell$  (thus leaving its value unaltered) and gives a contribution of zero to variable  $p_{\ell+1}$ , thus behaving as a NOP (No Operation). Hence no interference is produced in the variables involved in the RAM instruction currently simulated. On the other hand, if  $p_\ell = 1$  then the first program first zeroes  $r_i$  and then

assigns the value  $k$  to it, while the second program zeroes  $p_\ell$  and sets  $p_{\ell+1}$  to 1, thus pointing to the next instruction of  $M$  to be simulated.

Assignment instructions of type “ $\ell: r_i := r_j$ ”, with  $j \neq i$ , can be simulated using the following programs:

$$\begin{aligned} 0r_i + 2r_j + z|_{p_\ell} &\rightarrow 1|r_i + 1|r_j \\ p_\ell &\rightarrow 1|p_{\ell+1} \end{aligned}$$

As in the previous case, when  $p_\ell = 0$  the first program is not active while the second one operates like a NOP. When  $p_\ell = 1$ , instead, the first program first zeroes both  $r_i$  and  $r_j$  and then assigns to them the old value of  $r_j$ ; the second program, as before, passes the control to instruction  $\ell+1$ . Albeit in our definition of RAMs we have avoided the case when  $j = i$ , here we just observe that we can also easily deal with it: we simply remove the first program, since in this case it always operates like a NOP.

Arithmetic instructions of type “ $\ell: r_i := r_j \bullet r_k$ ”, with  $\bullet \in \{+, -, \times, \div\}$  and  $i \neq j$ ,  $j \neq k$ , and  $i \neq k$ , can be simulated as follows:

$$\begin{aligned} 0r_i + r_j \bullet r_k + z|_{p_\ell} &\rightarrow 1|r_i \\ r_j + z|_{p_\ell} &\rightarrow 1|r_j \end{aligned} \tag{2}$$

$$\begin{aligned} r_k + z|_{p_\ell} &\rightarrow 1|r_k \\ p_\ell &\rightarrow 1|p_{\ell+1} \end{aligned} \tag{3}$$

When  $p_\ell = 0$  the first three programs are not executed, while the last program behaves as a NOP. On the other hand, if  $p_\ell = 1$  then the first program first zeroes variables  $r_i$ ,  $r_j$  and  $r_k$ , and then it assigns to  $r_i$  the result of the operation  $r_j \bullet r_k$ , using the old values of  $r_j$  and  $r_k$ . Programs (2) and (3) are used to preserve the old values of variables  $r_j$  and  $r_k$ , whereas the last program passes the control to instruction  $\ell+1$ .

Finally, instructions of type “ $\ell: \text{if } r_i \neq 0 \text{ then } \ell_1 \text{ else } \ell_2$ ”, with  $\ell \neq \ell_1$ ,  $\ell \neq \ell_2$ , and  $\ell_1 \neq \ell_2$ , can be simulated by the following programs:

$$\begin{aligned} p_\ell &\rightarrow 1|p_{\ell_1} \\ r_i - 1|_{p_\ell} &\rightarrow 1|p_{\ell_1} \end{aligned} \tag{4}$$

$$r_i + 1|_{p_\ell} \rightarrow 1|p_{\ell_2} \tag{5}$$

in which we assume  $r_i \neq 0$  and correct if this is not the case. Note, in particular, that programs (4) and (5) are active if and only if  $p_\ell = 1$  and  $r_i = 0$ . So, when  $p_\ell = 0$  only the first program is executed, behaving as a NOP. When  $p_\ell = 1$  and  $r_i > 0$ , the first program passes the control to instruction  $\ell_1$  whereas the other two programs are not executed. Finally, when  $p_\ell = 1$  and  $r_i = 0$  the first program zeroes  $p_\ell$  and (incorrectly) sets  $p_{\ell_1}$  to 1. This time, however, also the other two programs are executed: after resetting once again the value of  $r_i$  to 0, program (4) gives a contribution of  $-1$  to  $p_{\ell_1}$ , so that its final value will be zero, whereas program (5) sets  $p_{\ell_2}$  to 1, indicating the next instruction of  $M$  to be simulated.  $\square$

## 5 Simulating all-parallel EN P Systems with RAMs

Having proved that all-parallel EN P systems are able to simulate efficiently random access machines using the same arithmetic operations, we now turn our attention to the converse simulation. Without loss of generality, we assume that the all-parallel EN P systems being simulated have a single membrane [6].

Since the production functions of EN P systems may evaluate to negative numbers, even if the variables themselves are always non-negative, it is convenient to employ RAMs with registers holding values in  $\mathbb{Z}$ . This poses no restriction, since signed integers may be simulated with a constant-time slowdown by RAMs using non-negative numbers, for instance by storing them with a sign-and-modulus representation.

**Proposition 7.** *Let  $\Pi$  be an  $\text{ENP}(+, -)$  (respectively, an  $\text{ENP}(+, -, \times, \div)$ ) working in all-parallel mode and polynomial time  $t(n) \leq c_1 n^k + c_0$ . Then, there exists a  $\text{RAM}(+, -)$  (respectively, a  $\text{RAM}(+, -, \times, \div)$ )  $M$  computing the same output as  $\Pi$  in time  $O(t(n)^3)$  (respectively,  $O(t(n))$ ).*

*Proof.* Let  $x_1, \dots, x_m$  be the variables of  $\Pi$ . The machine  $M$  stores the values of these variables in registers that we will denote with the same names, and will have the same value in the initial configuration, including the input variable of  $\Pi$ . Let  $p_1, \dots, p_h$  be the programs of  $\Pi$ .

Before describing the simulation proper, let us compute the maximum value of a variable of  $\Pi$ . If  $\Pi$  is an  $\text{ENP}(+, -)$ , then the rules have one of the following forms:

$$\begin{aligned} a_{i_1}x_{i_1} \pm \dots \pm a_{i_k}x_{i_k} \pm a &\rightarrow b_1|x_1 + \dots + b_m|x_m \\ a_{i_1}x_{i_1} \pm \dots \pm a_{i_k}x_{i_k} \pm a|_e &\rightarrow b_1|x_1 + \dots + b_m|x_m \end{aligned}$$

for some constants  $a, a_{i_1}, \dots, a_{i_k}, b_1, \dots, b_m \in \mathbb{N}$ . The following program, with some constant  $a \in \mathbb{N}$ , produces the maximum increase in the variable  $x$ , which we assume to be the input variable:

$$ax \rightarrow 1|x \tag{6}$$

After  $t = c_1 n^k + c_0$  computation steps, the value of  $x$  reaches its maximum  $a^t x$ . (Naturally, a program such as (6) is not admissible in a halting EN P system; that program is considered here only in order to provide an upper bound to the value of the variables of  $\Pi$ .)

On the other hand, if  $\Pi$  is an  $\text{ENP}(+, -, \times, \div)$ , the program that maximises the value of  $x$  is

$$x^a \rightarrow 1|x$$

for some  $a \in \mathbb{N}$ . In this case, after  $t$  steps the value of  $x$  reaches  $x^{a^t}$ . These upper bounds to the values of the variables of  $\Pi$  will be used later in order to determine the time required by  $M$  in order to simulate the EN P system.

The following is an overview of the simulation of  $\Pi$ :

```

repeat
  save the current values of the variables
  compute the variations due to  $p_1$  (if applicable)
   $\vdots$ 
  compute the variations due to  $p_h$  (if applicable)
  compute the new values of the variables
until a final configuration is reached
if  $\Pi$  accepted then
  accept
else
  reject
end

```

At the beginning of each simulated step, the current values of the variables are copied:

$$\begin{aligned}
 x'_1 &:= x_1 \\
 &\vdots \\
 x'_m &:= x_m
 \end{aligned}$$

In the variables  $\Delta_1, \dots, \Delta_m$ , initially zero, we accumulate the contributions to  $x_1, \dots, x_m$  given by the programs of  $\Pi$  during the current step:

$$\begin{aligned}
 \Delta_1 &:= 0 \\
 &\vdots \\
 \Delta_m &:= 0
 \end{aligned}$$

Each program  $p_i$  of the form  $f(x_{i_1}, \dots, x_{i_k}) \rightarrow a_1|x_1 + \dots + a_m|x_m$  is simulated as follows:

$$\begin{aligned}
 f &:= f(x_{i_1}, \dots, x_{i_k}) \\
 x'_{i_1} &:= 0 \\
 &\vdots \\
 x'_{i_k} &:= 0 \\
 u &:= f \div (a_1 + \dots + a_m) \\
 \Delta_1 &:= \Delta_1 + a_1 u \\
 &\vdots \\
 \Delta_m &:= \Delta_m + a_m u
 \end{aligned}$$

First, the value of the production function is computed. This requires  $O(1)$  time, since by construction  $\Pi$  and  $M$  admit the same basic arithmetic operations. Then, the copies of the variables occurring on the left-hand side of the program are zeroed.

The unit  $u$  to be distributed according to the repartition protocol is then computed. Here the division is performed in  $O(1)$  time if  $M$  is a  $\text{RAM}(+, -, \times, \div)$ , but  $O((\log f)^2) = O((\log(a^t x))^2) = O(t^2) = O(n^{2k})$  if it is a  $\text{RAM}(+, -)$ .

Finally, the contributions to the variables of  $\Pi$  are updated according to the repartition protocol. This only requires  $O(1)$  times, as  $a_1, \dots, a_m$  are constants.

Programs  $p_i$  of the form  $f(x_{i_1}, \dots, x_{i_k})|_e \rightarrow a_1|x_1 + \dots + a_m|x_m$  are simulated analogously, only with an extra test in order to ensure that the value of the enzyme is larger than the minimum of the variables.

```

if  $e > x_{i_1}$  or  $e > x_{i_1}$  or  $\dots$  or  $e > x_{i_k}$  then
   $f := f(x_{i_1}, \dots, x_{i_k})$ 
   $x'_{i_1} := 0$ 
   $\vdots$ 
   $x'_{i_k} := 0$ 
   $u := f \div (a_1 + \dots + a_m)$ 
   $\Delta_1 := \Delta_1 + a_1 u$ 
   $\vdots$ 
   $\Delta_m := \Delta_m + a_m u$ 
end

```

The time required is again  $O(1)$  if  $\Pi$  is an  $\text{ENP}(+, -, \times, \div)$  and  $O(n^{2k})$  if it is an  $\text{ENP}(+, -)$ .

After all programs have been examined (and applied, when possible), we can check whether a final configuration is reached: this occurs when, for each variable  $x_i$ , we have  $x_i = x'_i + \Delta_i$ , i.e., when the old value  $x_i$  equals the (possibly zeroed) value increased by the sum of the contributions it received in the current simulated step. If this is *not* the case, then the values of the variables are updated:

```

 $x_1 := x'_1 + \Delta_1$ 
 $\vdots$ 
 $x_m := x'_m + \Delta_m$ 

```

and the next step of  $\Pi$  is simulated.

When a final configuration is actually reached, the machine  $M$  checks the value of the *accept* variable of  $\Pi$  and provides the same result:

```

if  $\text{accept} = 1$  then
  accept
else
  reject
end

```

The total time required in order to perform the simulation of  $\Pi$  is  $O(n^{3k})$  for an  $\text{ENP}(+, -)$ , and  $O(n^k)$  for an  $\text{ENP}(+, -, \times, \div)$ .  $\square$

We can now state our main result, summarising the computational efficiency of EN P systems using arithmetic operations.

**Theorem 2.** *The following complexity classes coincide:*

$$\begin{aligned}\mathbf{P}\text{-ENP}(+, -) &= \mathbf{P}\text{-RAM}(+, -) = \mathbf{P} \\ \mathbf{P}\text{-ENP}(+, -, \times, \div) &= \mathbf{P}\text{-RAM}(+, -, \times, \div) = \mathbf{PSPACE}\end{aligned}$$

Furthermore, the inclusion  $\mathbf{P}\text{-ENP}(+, -, \times) \subseteq \mathbf{P}\text{-RAM}(+, -, \times)$  holds.  $\square$

## 6 Conclusions

We have analysed the computational efficiency of all-parallel EN P systems and their relationships with more traditional computing devices such as RAMs and Turing machines. We have showed some efficient simulations of all-parallel EN P systems by RAMs and vice versa, when the same basic arithmetic operations are used.

Hence we found that, by using only addition and subtraction, EN P systems working in polynomial time and all-parallel mode characterise the complexity class  $\mathbf{P}$ , whereas by also allowing multiplication and integer division we obtain a characterisation of  $\mathbf{PSPACE}$ .

Establishing the precise efficiency of all-parallel EN P systems (as well as random access machines) with addition, subtraction and multiplication is still an open problem. The possibility to extend the results exposed in this paper to EN P systems working in the sequential or in the one-parallel mode, as well as to numerical P systems not using the enzyme control, is also open.

## References

1. Bertoni, A., Mauri, G., Sabadini, N.: A characterization of the class of functions computable in polynomial time on random access machines. In: STOC '81 Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing. pp. 168–176 (1981), <http://dx.doi.org/10.1145/800076.802470>
2. Bottoni, P., Martin-Vide, C., Păun, Gh., Rozenberg, G.: Membrane systems with promoters/inhibitors. *Acta Informatica* 38(10), 695–720 (2002)
3. Buiu, C., Vasile, C., Arsene, O.: Development of membrane controllers for mobile robots. *Information Sciences* 187, 33–51 (2012), <http://dx.doi.org/10.1016/j.ins.2011.10.007>
4. Cook, S.A., Reckhow, R.A.: Time bounded random access machines. *Journal of Computer and System Sciences* 7, 354–375 (1973), [http://dx.doi.org/10.1016/S0022-0000\(73\)80029-7](http://dx.doi.org/10.1016/S0022-0000(73)80029-7)
5. Hartmanis, J., Simon, J.: On the power of multiplication in random access machines. In: IEEE Conference Record of 15th Annual Symposium on Switching and Automata Theory. pp. 13–23 (1974), <http://dx.doi.org/10.1109/SWAT.1974.20>
6. Leporati, A., Porreca, A.E., Zandron, C., Mauri, G.: Improving universality results on parallel enzymatic numerical P systems. In: Proceedings of the Eleventh Brainstorming Week on Membrane Computing (2013), to appear

7. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
8. Păun, Gh., Păun, R.: Membrane computing and economics: Numerical P systems. *Fundamenta Informaticae* 73(1–2), 213–227 (2006), <http://iospress.metapress.com/content/7xyefwrwy7mkg46a/>
9. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
10. Pavel, A., Arsene, O., Buiu, C.: Enzymatic numerical P systems – A new class of membrane computing systems. In: Li, K., Tang, Z., Li, R., Nagar, A.K., Thamburaj, R. (eds.) *Proceedings 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA 2010)*. pp. 1331–1336 (2010), <http://dx.doi.org/10.1109/BICTA.2010.5645071>
11. Pavel, A.B., Buiu, C.: Using enzymatic numerical P systems for modeling mobile robot controllers. *Natural Computing* 11(3), 387–393 (2012), <http://dx.doi.org/10.1007/s11047-011-9286-5>
12. Vasile, C.I., Pavel, A.B., Dumitrache, I., Păun, Gh.: On the power of enzymatic numerical P systems. *Acta Informatica* 49, 395–412 (2012), <http://dx.doi.org/10.1007/s00236-012-0166-y>



# Communication Rules Working in Generated Membrane Boundaries

Tamás Mihálydeák<sup>1</sup>, Zoltán Ernő Csajbók<sup>2</sup>, and Péter Takács<sup>2</sup>

<sup>1</sup> Department of Computer Science, Faculty of Informatics, University of Debrecen  
Kassai út 26, H-4028 Debrecen, Hungary

`mihalydeak.tamas@inf.unideb.hu`

<sup>2</sup> Department of Health Informatics, Faculty of Health, University of Debrecen,  
Sóstói út 2-4, H-4400 Nyíregyháza, Hungary  
`{csajbok.zoltan,takacs.peter}@foh.unideb.hu`

**Abstract.** In natural processes, the events represented by communication rules in membrane computing are taken place in the vicinity of membranes. Looking at regions as multisets, partial approximation spaces generalized for multisets give a plausible opportunity to model membrane boundaries in an abstract way. Thus, motivated by natural phenomena, the abstract notion of “to be close enough to a membrane” can be built in membrane computing.

**Keywords:** Membrane computing, multiset theory, partial approximation of multisets.

## 1 Introduction

Membrane computing invented by Păun [15–17] is motivated by biological and chemical processes. Membranes delimit regions for which a set of rules is given. Evolution rules model reactions inside regions (e.g., like chemical processes work), whereas communication rules model movements of objects through membranes (e.g., like biological processes work).

In natural processes, however, the events represented by communication rules are taken place in the vicinity of a membrane. In membrane computing, there are some attempts to interpret membrane boundary based on space perception [1, 3]. In regions, however, there is no precise information about the nature of the space of objects or their positions in general [4].

In [11], an abstract, not necessarily space-like, membrane boundary was proposed. Accordingly, looking at regions as multisets, partial approximation spaces generalized for multisets give a plausible opportunity to model the abstract concept of “to be close enough to a membrane”.

The paper, with the help of examples, undertakes to show what happens when the executions of communication rules are constrained to membrane boundaries defined in an abstract way. Having outlined the fundamental notions of partial multiset approximation spaces in Section 2, Section 3 and 4 present its application to membrane computing and the examples indicated before.

## 2 General Multiset Approximation Spaces

Set approximations were invented by Pawlak [13, 14]. There are many different generalizations of Pawlakian rough set theory, among others, for multisets relying on equivalence or general multirelations [6, 8]. Partial nature of real-life problems, however, requires working out partial approximation schemes. Such a scheme for multiset first was proposed in [10, 11] in connection with membrane computing introduced by Păun [15–17]. In this section, the most important features of partial multiset approximation spaces are summarized (based on [11]).

### 2.1 Set-Theoretical Relations and Operations for Multisets

Let  $U$  be a finite nonempty set. A *multiset*  $M$ , or *mset*  $M$  for short, over  $U$  is a mapping  $M : U \rightarrow \mathbb{N} \cup \{\infty\}$ , where  $\mathbb{N}$  is the set of natural numbers. The set  $M^* = \{a \in U \mid M(a) \neq 0\}$  is called the *support* of  $M$ .  $M$  is *finite* if  $M(a) < \infty$  for all  $a \in M^*$ . The mset  $M$  over  $U$  is the *empty mset*, denoted by  $\emptyset$  if  $M^* = \emptyset$ .

Let  $\mathcal{MS}(U)$  denote the set of all multisets over  $U$ .

A set  $\mathcal{M}$  of finite multisets over  $U$  is called a *macroset*  $\mathcal{M}$  over  $U$  [9]. We define the following two fundamental macrosets:  $\mathcal{MS}^n(U)$  ( $n \in \mathbb{N}$ ), the set of all multisets  $M$  over  $U$  such that  $M(a) \leq n$  for all  $a \in U$ , and  $\mathcal{MS}^{<\infty}(U) = \bigcup_{n=0}^{\infty} \mathcal{MS}^n(U)$ .

The basic set-theoretical relations can be generalized for multisets as follows.

**Definition 1 ([11]).** Let  $M, M_1, M_2$  be multisets over  $U$ .

1. Multiplicity relation for an mset  $M$  over  $U$  is:  
 $a \in M$  ( $a \in U$ ) if  $M(a) \geq 1$ .
2. Let  $n \in \mathbb{N}^{>0}$  be a positive integer.  $n$ -times multiplicity relation for an mset  $M$  over  $U$  is:  $a \in^n M$  ( $a \in U$ ) if  $M(a) = n$ .
3.  $M_1 = M_2$  if  $M_1(a) = M_2(a)$  for all  $a \in U$  (mset equality relation).
4.  $M_1 \sqsubseteq M_2$  if  $M_1(a) \leq M_2(a)$  for all  $a \in U$  (mset inclusion relation).

The next definitions give the generalizations for multisets of the basic set-theoretical operations.

**Definition 2 ([11]).** Let  $M, M_1, M_2 \in \mathcal{MS}(U)$  be multisets over  $U$  and  $\mathcal{M} \subseteq \mathcal{MS}(U)$  be a set of multisets over  $U$ .

1.  $(M_1 \sqcap M_2)(a) = \min\{M_1(a), M_2(a)\}$  for all  $a \in U$  (intersection).
2.  $(\bigcap \mathcal{M})(a) = \min\{M(a) \mid M \in \mathcal{M}\}$  for all  $a \in U$ .
3.  $(M_1 \sqcup M_2)(a) = \max\{M_1(a), M_2(a)\}$  for all  $a \in U$  (set-type union).
4.  $(\bigcup \mathcal{M})(a) = \sup\{M(a) \mid M \in \mathcal{M}\}$  for all  $a \in U$ . By definition,  $\bigcup \emptyset = \emptyset$ .
5.  $(M_1 \oplus M_2)(a) = M_1(a) + M_2(a)$  for all  $a \in U$  (mset addition).
6. For any  $n \in \mathbb{N}$ ,  $n$ -times addition of  $M$ , denoted by  $\oplus_n M$ , is given by the following inductive definition:
  - (a)  $\oplus_0 M = \emptyset$ ;
  - (b)  $\oplus_1 M = M$ ;
  - (c)  $\oplus_{n+1} M = \oplus_n M \oplus M$ .
7.  $(M_1 \ominus M_2)(a) = \max\{M_1(a) - M_2(a), 0\}$  for all  $a \in U$  (mset subtraction).

By the  $n$ -times addition, the  $n$ -times inclusion relation ( $\sqsubseteq^n$ ) can be defined.

**Definition 3.** Let  $M_1 \neq \emptyset, M_2$  be two msets over  $U$ .

For any  $n \in \mathbb{N}$ ,  $M_1 \sqsubseteq^n M_2$  if  $\oplus_n M_1 \sqsubseteq M_2$  but  $\oplus_{n+1} M_1 \not\sqsubseteq M_2$ .

**Corollary 1.** Let  $M_1 \neq \emptyset, M_2$  be two msets over  $U$ .

Then for all  $n \in \mathbb{N}$ ,  $M_1 \sqsubseteq^n M_2$  if and only if  $nM_1(a) \leq M_2(a)$  for all  $a \in U$  and there is an  $a' \in U$  such that  $(n+1)M_1(a') > M_2(a')$ .

Note that  $U \langle \mathcal{MS}(U), \sqcup, \sqcap \rangle$  is a complete lattice [2, 5, 7], and  $\langle \mathcal{MS}(U), \sqsubseteq \rangle$  is a partially ordered set in which  $M_1 \sqsubseteq M_2$  if and only if  $M_1 \sqcup M_2 = M_2$ , or equivalently,  $M_1 \sqcap M_2 = M_1$  ( $M_1, M_2 \in \mathcal{MS}(U)$ ).

In addition,  $\langle \mathcal{MS}^{<\infty}(U), \sqcup, \sqcap \rangle$  is the sublattice of  $\langle \mathcal{MS}(U), \sqcup, \sqcap \rangle$ . However,  $\langle \mathcal{MS}^{<\infty}(U), \sqcup, \sqcap \rangle$  is not a complete lattice because of it lacks a top element. For more details, see [12].

## 2.2 General Multiset Approximation Spaces

A general mset approximation space has four basic components:

- a set of msets as the *domain* of the space whose members are approximated;
- some distinguished msets of the domain as the *basis* of approximations;
- *definable msets* which are derived from base msets in some way and candidates for possible approximations of members of the domain;
- an *approximation pair* determines the lower and upper approximations of members of the domain relying on definable msets.

**Definition 4** ([11]). The ordered 5-tuple  $\text{MAS}(U) = \langle \mathcal{MS}^{<\infty}(U), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, \mathfrak{l}, \mathfrak{u} \rangle$  is a (general) mset approximation space over  $U$  with the domain  $\mathcal{MS}^{<\infty}(U)$  if

1.  $\mathfrak{B} \subseteq \mathcal{MS}^{<\infty}(U)$  and if  $B \in \mathfrak{B}$ , then  $B \neq \emptyset$  (in notation  $\mathfrak{B} = \{B_\gamma \mid \gamma \in \Gamma\}$  where  $\Gamma$  is an arbitrary non-empty set of indexes);  
 $\mathfrak{B}$  is called the base system, its members are called the base msets;
2.  $\mathfrak{D}_{\mathfrak{B}} \subseteq \mathcal{MS}^{<\infty}(U)$  is an extension of  $\mathfrak{B}$  satisfying the following minimal requirement: if  $B \in \mathfrak{B}$ , then  $\oplus_n B \in \mathfrak{D}_{\mathfrak{B}}$  for all  $n \in \mathbb{N}$ ; members in  $\mathfrak{D}_{\mathfrak{B}}$  are called definable msets;
3. the functions  $\mathfrak{l}, \mathfrak{u} : \mathcal{MS}^{<\infty}(U) \rightarrow \mathcal{MS}^{<\infty}(U)$  (called lower and upper approximations) form a weak approximation pair  $\langle \mathfrak{l}, \mathfrak{u} \rangle$  if
  - (C0)  $\mathfrak{l}(\mathcal{MS}^{<\infty}(U)), \mathfrak{u}(\mathcal{MS}^{<\infty}(U)) \subseteq \mathfrak{D}_{\mathfrak{B}}$  (definability of  $\mathfrak{l}, \mathfrak{u}$ );<sup>3</sup>
  - (C1) the functions  $\mathfrak{l}$  and  $\mathfrak{u}$  are monotone, i.e., for all  $M_1, M_2 \in \mathcal{MS}^{<\infty}(U)$  if  $M_1 \sqsubseteq M_2$ , then  $\mathfrak{l}(M_1) \sqsubseteq \mathfrak{l}(M_2)$ ,  $\mathfrak{u}(M_1) \sqsubseteq \mathfrak{u}(M_2)$  (monotonicity of  $\mathfrak{l}, \mathfrak{u}$ );
  - (C2)  $\mathfrak{u}(\emptyset) = \emptyset$  (normality of  $\mathfrak{u}$ );
  - (C3) if  $M \in \mathcal{MS}^{<\infty}(U)$ , then  $\mathfrak{l}(M) \sqsubseteq \mathfrak{u}(M)$  (weak approximation property).

**Corollary 2.**  $\mathfrak{l}(\emptyset) = \emptyset$  (normality of  $\mathfrak{l}$ ).

<sup>3</sup>  $\mathfrak{l}(\mathcal{MS}^{<\infty}(U))$ , and  $\mathfrak{u}(\mathcal{MS}^{<\infty}(U))$  denote the range of the functions  $\mathfrak{l}$  and  $\mathfrak{u}$ .

$\text{MAS}(U)$  is *total*, if for any  $M \in \mathcal{MS}^{<\infty}(U)$  there is a definable mset  $D \in \mathfrak{D}_{\mathfrak{B}}$  such that  $M \sqsubseteq D$ , it is *partial* otherwise.

It is reasonable to assume that the base msets and their  $n$ -times additions are exactly approximated from “lower side”. In certain cases, it is also required of definable msets.

**Definition 5.** A weak approximation pair  $\langle l, u \rangle$  is

- (C4) granular if  $B \in \mathfrak{B}$ , then  $l(\oplus_n B) = \oplus_n B$  ( $n \in \mathbb{N}$ ) (in other words,  $l$  is granular);
- (C5) standard if  $D \in \mathfrak{D}_{\mathfrak{B}}$ , then  $l(D) = D$  (in other words,  $l$  is standard).

An important question is how lower and upper approximations relate to the approximated mset.

**Definition 6.** A weak approximation pair  $\langle l, u \rangle$  is

- (C6) lower semi-strong if  $l(M) \sqsubseteq M$  ( $M \in \mathcal{MS}^{<\infty}(U)$ ) ( $l$  is contractive);
- (C7) upper semi-strong if  $M \sqsubseteq u(M)$  ( $M \in \mathcal{MS}^{<\infty}(U)$ ) ( $u$  is extensive);
- (C8) strong if it is lower and upper semi-strong simultaneously, i.e., each subset  $M \in \mathcal{MS}^{<\infty}(U)$  is bounded by  $l(M)$  and  $u(M)$ :  $l(M) \sqsubseteq M \sqsubseteq u(M)$ .

**Definition 7.** The general mset approximation space  $\text{MAS}(U)$  is a weak/granular/standard/lower semi-strong/upper semi-strong/strong mset approximation space, if the approximation pair  $\langle l, u \rangle$  is weak/granular/standard/lower semi-strong/upper semi-strong/strong, respectively.

### 2.3 Generalized Pawlakian Multiset Approximation Spaces

It is a natural assumption that  $\mathfrak{D}_{\mathfrak{B}}$  is obtained (derived) from  $\mathfrak{B}$  by some sorts of transformations, for the most important cases, see [11]. In order to build a generalized Pawlakian mset approximation space, first, we define  $\mathfrak{D}_{\mathfrak{B}}$  as follows.

**Definition 8 ([11]).**  $\text{MAS}(U)$  is a strictly set-union type mset approximation space if  $\mathfrak{D}_{\mathfrak{B}}$  is given by the following inductive definition:

1.  $\emptyset \in \mathfrak{D}_{\mathfrak{B}}$ ;
2.  $\mathfrak{B} \subseteq \mathfrak{D}_{\mathfrak{B}}$ ;
3. if  $\mathfrak{B}^{\oplus} = \{\oplus_n B \mid B \in \mathfrak{B}, n = 1, 2, \dots\}$  and  $\mathfrak{B}' \subseteq \mathfrak{B}^{\oplus}$ , then  $\bigsqcup \mathfrak{B}' \in \mathfrak{D}_{\mathfrak{B}}$ .

The next proposition summarizes the most important features of strictly set-union type mset approximation spaces.

**Proposition 1 ([12]).** Let  $\text{MAS}(U) = \langle \mathcal{MS}^{<\infty}(U), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, l, u \rangle$  be a strictly set-union type mset approximation space over  $U$ .

1. For any definable set  $D \in \mathfrak{D}_{\mathfrak{B}}$ ,

$$D = \bigsqcup \{\oplus_n B \in \mathfrak{B}^{\oplus} \mid n \in \mathbb{N}^{>0}, B \in \mathfrak{B}, B \sqsubseteq^n D\}.$$

2. If  $\text{MAS}(U)$  is also granular and lower semi-strong, for any  $M \in \mathcal{MS}^{<\infty}(U)$ ,

$$l(M) = \bigsqcup \{\oplus_n B \in \mathfrak{B}^{\oplus} \mid n \in \mathbb{N}^{>0}, B \in \mathfrak{B}, B \sqsubseteq^n M\}.$$

Next, the Pawlakian approximation pair for msets is generalized in strictly set-union type mset approximation spaces.

**Definition 9 ([11]).** Let  $\text{MAS}(U) = \langle \mathcal{MS}^{<\infty}(U), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, \mathfrak{l}, \mathfrak{u} \rangle$  be a strictly set-union type mset approximation space.

The functions  $\mathfrak{l}, \mathfrak{u} : \mathcal{MS}^{<\infty}(U) \rightarrow \mathcal{MS}^{<\infty}(U)$  are a (generalized) Pawlakian mset approximation pair  $\langle \mathfrak{l}, \mathfrak{u} \rangle$  if for any mset  $M \in \mathcal{MS}^{<\infty}(U)$

1.  $\mathfrak{l}(M) = \bigsqcup \{ \oplus_n B \mid n \in \mathbb{N}^{>0}, B \in \mathfrak{B} \text{ and } B \sqsubseteq^n M \},$
2.  $\mathfrak{b}(M) = \bigsqcup \{ \oplus_n B \mid B \in \mathfrak{B}, B \not\sqsubseteq M, B \sqcap M \neq \emptyset \text{ and } B \sqcap M \sqsubseteq^n M \},$
3.  $\mathfrak{u}(M) = \mathfrak{l}(M) \sqcup \mathfrak{b}(M),$

where the function  $\mathfrak{b}$  gives the Pawlakian boundary of the mset  $M$ .

It is easy to check by Definition 9 that when  $\text{MAS}(U)$  is a strictly set-union type mset approximation space with a Pawlakian mset approximation pair,  $\text{MAS}(U)$  is a lower semi-strong mset approximation space, and  $\mathfrak{l}$  is granular. In other words,  $\text{MAS}(U)$  fulfills the conditions (C0)–(C3), (C4), (C6).

**Definition 10.** A strictly set-union type approximation space with a Pawlakian mset approximation pair is called a Pawlakian mset approximation space.

### 3 Applications in Membrane Computing

**Definition 11.** A membrane structure  $\mu$  of degree  $m$  ( $m \geq 1$ ) is a rooted tree with  $m$  nodes identified with the integers  $1, \dots, m$ .

A membrane structure  $\mu$  of degree  $m$  ( $m \geq 1$ ) can be represented by the set  $R_\mu \subseteq \{1, \dots, m\} \times \{1, \dots, m\}$ .  $\langle i, j \rangle \in R_\mu$  means that there is an edge from  $i$  (parent) to  $j$  (child) of the tree  $\mu$  which is formulated by  $\text{parent}(j) = i$ .

**Definition 12.** Let  $\mu$  be a membrane structure with  $m$  nodes and  $V$  be a finite alphabet. The tuple

$$\Pi = \langle V, \mu, w_1, w_2, \dots, w_m, R_1, R_2, \dots, R_m \rangle$$

is a  $P$  system if

1.  $w_i \in \mathcal{MS}^{<\infty}(V)$  for  $i = 1, 2, \dots, m$ ;
2.  $R_i$  is a finite set of rules for  $i = 1, 2, \dots, m$  such that if  $r \in R_i$ , its form is one of the following:
  - (a) symport rules:  $\langle u, \text{in} \rangle, \langle u, \text{out} \rangle$ , where  $u \neq \lambda$  and there is an mset  $M \in \mathcal{MS}^{<\infty}(V)$  such that  $u$  represents  $M$ ;
  - (b) antiport rule:  $\langle u, \text{in}; v, \text{out} \rangle$ , where  $u \neq \lambda, v \neq \lambda$  and there are msets  $M_1, M_2 \in \mathcal{MS}^{<\infty}(V)$  such that  $u, v$  represent  $M_1, M_2$ , respectively.

If the  $P$  system  $\Pi = \langle V, \mu, w_1, w_2, \dots, w_m, R_1, R_2, \dots, R_m \rangle$  is given, let  $\text{MAS}(\Pi) = \langle \mathcal{MS}^{<\infty}(V), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, \mathfrak{l}, \mathfrak{u} \rangle$  be a strictly set-union type mset approximation space with a generalized Pawlakian approximation pair  $\langle \mathfrak{l}, \mathfrak{u} \rangle$ .  $\text{MAS}(\Pi)$  is called a *joint membrane approximation space*.

Having given a membrane system  $\Pi$  and its joint membrane approximation space  $\text{MAS}(\Pi)$ , we can define the boundaries of the regions  $w_1, w_2, \dots, w_m$  as msets with the help of approximative functions  $\mathfrak{l}, \mathfrak{u}, \mathfrak{b}$  specified in Definition 9.<sup>4</sup>

**Definition 13 ([11]).** Let  $\Pi = \langle V, \mu, w_1, w_2, \dots, w_m, R_1, R_2, \dots, R_m \rangle$  be a  $P$  system and  $\text{MAS}(\Pi) = \langle \mathcal{MS}^{<\infty}(V), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, \mathfrak{l}, \mathfrak{u} \rangle$  be its joint membrane approximation space. If  $B \in \mathfrak{B}$  and  $i = 1, 2, \dots, m$ , let

$$N(B, i) = \begin{cases} 0, & \text{if } B \sqsubseteq w_i \text{ or } B \sqcap w_i = \emptyset; \\ n, & \text{if } i = 1 \text{ and } B \sqcap w_1 \sqsubseteq^n w_1; \\ \min\{k, n \mid B \sqcap w_i \sqsubseteq^k w_i, \text{ and } B \ominus w_i \sqsubseteq^n w_{\text{parent}(i)}\}, & \text{otherwise.} \end{cases}$$

Then, for  $i = 1, \dots, m$ ,

$$\begin{aligned} \text{bnd}(w_i) &= \bigsqcup \{ \oplus_{N(B, i)} B \mid B \in \mathfrak{B} \}; \text{bnd}^{\text{out}}(w_i) = \text{bnd}(w_i) \ominus w_i; \\ \text{bnd}^{\text{in}}(w_i) &= \text{bnd}(w_i) \ominus \text{bnd}^{\text{out}}(w_i). \end{aligned}$$

The functions  $\text{bnd}, \text{bnd}^{\text{out}}, \text{bnd}^{\text{in}}$  give *membrane boundaries, outside and inside membrane boundaries*, respectively.

The general notion of boundaries given in Definition 9 cannot be used here, because membrane boundaries have to follow the given membrane structure  $\mu$ . The lower approximations  $\mathfrak{l}(w_i)$  ( $i = 1, \dots, m$ ) obey the membrane structure. The upper approximation  $\mathfrak{u}(w_1)$  and the Pawlakian boundaries  $\mathfrak{b}(w_1)$  are wholly within the environment of the membrane structure. However, the upper approximation  $\mathfrak{u}(w_i)$ , therefore the Pawlakian boundary  $\mathfrak{b}(w_i)$  ( $i = 2, \dots, m$ ) do not obey the membrane structure in general. Thus, the Pawlakian boundaries have to be adjusted to the membrane structure by the function  $\text{bnd}$ . Of course,  $\mathfrak{b}(w_1) = \text{bnd}(w_1)$ , but  $\mathfrak{b}(w_i) \neq \text{bnd}(w_i)$  ( $i = 2, \dots, m$ ) in general. Moreover, membrane boundaries  $\text{bnd}(w_i)$  ( $i = 1, \dots, m$ ) are split into two parts, inside and outside membrane boundaries.

Using membrane boundaries, the following constraints for rule executions are prescribed: a rule  $r \in R_i$  of a membrane  $i$  ( $i = 1, \dots, m$ ) has to work only in the membrane boundary of its region. More precisely,

- a symport rule of the form  $\langle u, in \rangle$  is executed only in the case when  $u \sqsubseteq \text{bnd}^{\text{out}}(w_i)$ ;
- a symport rule of the form  $\langle u, out \rangle$  is executed only in the case when  $u \sqsubseteq \text{bnd}^{\text{in}}(w_i)$ ;
- an antiport rule of the form  $\langle u, in; v, out \rangle$  is executed only in the case when  $u \sqsubseteq \text{bnd}^{\text{out}}(w_i)$  and  $v \sqsubseteq \text{bnd}^{\text{in}}(w_i)$ .

It can be shown that the membrane computation actually works in the membrane boundaries, see [11], Theorem 1.

<sup>4</sup> We are speaking about the boundaries of regions but, to tell the truth, these boundaries are the boundaries of msets of different regions.

## 4 An Illustrative Example

In this section, we follow the customary representations of msets. Accordingly, if an mset  $M$  is finite, it is represented by all permutations of the string  $w$ :

$$w = \begin{cases} a_{k_1}^{M(a_{k_1})} a_{k_2}^{M(a_{k_2})} \dots a_{k_l}^{M(a_{k_l})}, & \text{if } M \text{ is nonempty;} \\ \lambda, & \text{otherwise;} \end{cases}$$

where  $M^* = \{a_{k_1}, a_{k_2}, \dots, a_{k_l}\} \subseteq U$  and  $\lambda$  is the empty string.

As usual, with a slight abuse of terminology, simply “the mset  $w$ ” is said instead of “the mset  $M$  represented by the string  $w$  and all of its permutations”. Moreover, any permutation of the string  $w$  can also represent  $M$ .

### 4.1 Giving the $P$ system and its Joint Membrane Approximation Space

Let the  $P$  system be  $\Pi = \langle U, \mu, w_1, R \rangle$ , where

- $U = \{a, b, c, d, e, f\}$  is a finite alphabet;
- $\mu$  is a membrane structure of degree 1;
- the region  $w_1$  is represented by the multiset  $w_1 = ab^{11}c^3d^9e$ ;
- $R = \{\langle ac; out \rangle, \langle b^6d^6; out \rangle, \langle d^3e; out \rangle\}$  is the set of communication rules.

Let the joint membrane approximation space of  $\Pi$  be a strictly set-union type mset approximation space with a generalized Pawlakian approximation pair. That is let  $\text{MAS}(\Pi) = \langle \mathcal{MS}^{<\infty}(U), \mathfrak{B}, \mathfrak{D}_{\mathfrak{B}}, \mathbf{l}, \mathbf{u} \rangle$ , where

- $\mathcal{MS}^{<\infty}(U)$  is the domain of  $\text{MAS}(\Pi)$ ;
- $\mathfrak{B} = \{a^2b, abcdef, ac, b^3cd^2, b^3d^2, b^3d^2f, c, e^3, f^2, f^4\}$  is the base system;
- $\mathfrak{D}_{\mathfrak{B}}$  is the set of definable sets such that
  - $\emptyset \in \mathfrak{D}_{\mathfrak{B}}$ ;
  - $\mathfrak{B}^{\oplus} = \{a^2b, a^4b^2, a^6b^3, \dots, abcdef, a^2b^2c^2d^2e^2f^2, a^3b^3c^3d^2e^3f^3, \dots, ac, a^2c^2, a^3c^3, \dots, b^3cd^2, b^6c^2d^4, b^9c^3d^6, \dots, b^3d^2, b^6d^4, b^9d^6, \dots, b^3d^2f, b^6d^4f^2, b^9d^6f^3, \dots, c, c^2, c^3, \dots, e^3, e^6, e^9, \dots, f^2, f^4, f^6, \dots, f^8, f^{12}\}$ , and for any  $\mathfrak{B}' \subseteq \mathfrak{B}^{\oplus}$ ,  $\bigsqcup \mathfrak{B}' \in \mathfrak{D}_{\mathfrak{B}}$ ;
  - $\mathfrak{D}_{\mathfrak{B}}$  does not have any other member;
- $\langle \mathbf{l}, \mathbf{u} \rangle$  is a Pawlakian mset approximation pair.

Throughout the computation processes, we utilize the fact that  $M_1 \sqsubseteq^n M_2$  if and only if  $\oplus_n M_1 \sqsubseteq^1 M_2$  ( $M_1, M_2 \in \mathcal{MS}^{<\infty}(U)$ ,  $n \in \mathbb{N}^{>0}$ ).

### 4.2 Computing the Pawlakian Lower- and Upper Approximations and the Boundary

Computation of  $\mathbf{l}(w_1)$  By Definition 9,

$$\mathbf{l}(w_1) = \mathbf{l}(ab^{11}c^3d^9e) = \bigsqcup \{\oplus_n B \mid n \in \mathbb{N}^{>0}, B \in \mathfrak{B} \text{ and } B \sqsubseteq^n ab^{11}c^3d^9e\}.$$

The computation process of  $l(w_1)$  can be tracked by Table 1.<sup>5</sup> The result is:

$$\begin{aligned} l(w_1) &= \oplus_1 ac \sqcup \oplus_3 b^3 cd^2 \sqcup \oplus_3 b^3 d^2 \sqcup \oplus_3 c \\ &= ac \sqcup b^9 c^3 d^6 \sqcup b^9 d^6 \sqcup c^3 \\ &= ab^9 c^3 d^6 \end{aligned}$$

Computation of  $b(w_1)$  By Definition 9,

$$\begin{aligned} b(w_1) = b(ab^{11}c^3d^9) &= \bigsqcup \{ \oplus_n B \mid B \in \mathfrak{B}, B \not\sqsubseteq ab^{11}c^3d^9, B \sqcap ab^{11}c^3d^9 \neq \emptyset \\ &\text{and } B \sqcap ab^{11}c^3d^9 \sqsubseteq^n ab^{11}c^3d^9 \}. \end{aligned}$$

The computation process of  $b(w_1)$  can be tracked by Table 2. The result is:

$$\begin{aligned} b(w_1) &= \oplus_1 a^2 b \sqcup \oplus_1 abcdef \sqcup \oplus_3 b^3 d^2 f \sqcup \oplus_1 e^3 \\ &= a^2 b \sqcup abcdef \sqcup b^9 d^6 f^3 \sqcup e^3 = a^2 b^9 cd^6 e^3 f^3 \\ &= a^2 b^9 cd^6 e^3 f^3 \end{aligned}$$

Computation of  $u(w_1)$  By Definition 9,  $u(w_1) = l(w_1) \sqcup b(w_1)$ , and so

$$\begin{aligned} u(w_1) = u(ab^{11}c^3d^9) &= l(ab^{11}c^3d^9) \sqcup b(ab^{11}c^3d^9) \\ &= ab^9 c^3 d^6 \sqcup a^2 b^9 cd^6 e^3 f^3 \\ &= a^2 b^9 c^3 d^6 e^3 f^3. \end{aligned}$$

Since  $u(w_1) \in \mathfrak{D}_{\mathfrak{B}}$ , by Proposition 1(1),  $u(w_1)$  is decomposable, i.e.,  $u(w_1)$  can be formed as a set-type unions of base msets. Its computation can be tracked by Table 3. The result is:

$$\begin{aligned} u(w_1) = u(ab^{11}c^3d^9) &= a^2 b^9 c^3 d^6 e^3 f^3 \\ &= a^2 b \sqcup a^2 b^2 c^2 d^2 e^2 f^2 \sqcup a^2 c^2 \sqcup b^9 c^3 d^6 \sqcup b^9 d^6 \sqcup b^9 d^6 f^3 \sqcup e^3 \sqcup f^2 \\ &= \oplus_1 a^2 b \sqcup \oplus_2 abcdef \sqcup \oplus_2 ac \sqcup \oplus_3 b^3 cd^2 \sqcup \oplus_3 b^3 d^2 \sqcup \oplus_3 b^3 d^2 f \sqcup \oplus_3 c \\ &\quad \sqcup \oplus_1 e^3 \sqcup \oplus_1 f^2 \end{aligned}$$

### 4.3 Computing (Inside/Outside) Membrane Boundaries

Computation of  $\mathbf{bnd}(w_1)$  The membrane boundary and the Pawlakian boundary are equal for the skin membrane, i.e.,  $\mathbf{bnd}(w_1) = b(w_1)$ . Therefore,

$$\mathbf{bnd}(w_1) = b(w_1) = a^2 b^9 cd^6 e^3 f^3.$$

The Pawlakian boundary  $b(w_1) = a^2 b^9 cd^6 e^3 f^3$  was computed by Definition 9 with the help of Table 2. In order to check the equality  $\mathbf{bnd}(w_1) = b(w_1)$ , let us compute  $\mathbf{bnd}(w_1)$  by Definition 13, too.

<sup>5</sup> All tables are placed after the Section Acknowledgments.



The numbers  $N(B, 1)$  ( $B \in \mathfrak{B}$ ) (see Definition 13) can be determined as follows:

$$\begin{aligned}
N(a^2b, 1) &= 1, \text{ because } a^2b \not\sqsubseteq ab^{11}c^3d^9e, a^2b \sqcap ab^{11}c^3d^9e = ab \neq \emptyset, \\
&\quad \text{and } a^2b \sqcap ab^{11}c^3d^9e = ab \sqsubseteq^1 ab^{11}c^3d^9e; \\
N(abcdef, 1) &= 1, \text{ because } abcdef \not\sqsubseteq ab^{11}c^3d^9e, abcdef \sqcap ab^{11}c^3d^9e = abcde \neq \emptyset, \\
&\quad \text{and } abcdef \sqcap ab^{11}c^3d^9e = abcde \sqsubseteq^1 ab^{11}c^3d^9e; \\
N(ac, 1) &= 0, \text{ because } ac \sqsubseteq ab^{11}c^3d^9e; \\
N(b^3cd^2, 1) &= 0, \text{ because } b^3cd^2 \sqsubseteq ab^{11}c^3d^9e; \\
N(b^3d^2, 1) &= 0, \text{ because } b^3d^2 \sqsubseteq ab^{11}c^3d^9e; \\
N(b^3d^2f, 1) &= 3, \text{ because } b^3d^2f \sqcap ab^{11}c^3d^9e = b^3d^2 \sqsubseteq^3 ab^{11}c^3d^9e; \\
N(c, 1) &= 0, \text{ because } c \sqsubseteq ab^{11}c^3d^9e; \\
N(e^3, 1) &= 1, \text{ because } e^3 \not\sqsubseteq ab^{11}c^3d^9e, e^3 \sqcap ab^{11}c^3d^9e = e \neq \emptyset, \\
&\quad \text{and } e^3 \sqcap ab^{11}c^3d^9e = e \sqsubseteq^1 ab^{11}c^3d^9e; \\
N(f^2, 1) &= 0, \text{ because } f^2 \not\sqsubseteq ab^{11}c^3d^9e, f^2 \sqcap ab^{11}c^3d^9e = \emptyset; \\
N(f^4, 1) &= 0, \text{ because } f^4 \not\sqsubseteq ab^{11}c^3d^9e, f^4 \sqcap ab^{11}c^3d^9e = \emptyset.
\end{aligned}$$

Hence, by Definition 13,

$$\begin{aligned}
\mathbf{bnd}(w_1) &= \bigsqcup \{ \oplus_{N(B,1)} B \mid B \in \{a^2b, abcdef, ac, b^3cd^2, b^3d^2, b^3d^2f, c, e^3, f^2, f^4\} \} \\
&= \oplus_1 a^2b \sqcup \oplus_1 abcdef \sqcup \oplus_0 ac \sqcup \oplus_0 b^3cd^2 \sqcup \oplus_0 b^3d^2 \sqcup \oplus_3 b^3d^2f \sqcup \oplus_0 c \\
&\quad \sqcup \oplus_1 e^3 \sqcup \oplus_0 f^2 \sqcup \oplus_0 f^4 \\
&= a^2b \sqcup abcdef \sqcup \emptyset \sqcup \emptyset \sqcup \emptyset \sqcup b^9d^6f^3 \sqcup \emptyset \sqcup e^3 \sqcup \emptyset \sqcup \emptyset \\
&= a^2b^9cd^6e^3f^3.
\end{aligned}$$

Computation of  $\mathbf{bnd}^{\text{out}}(w_1)$  By Definition 13,

$$\begin{aligned}
\mathbf{bnd}^{\text{out}}(w_1) &= \mathbf{bnd}(w_1) \ominus w_1 = \mathbf{b}(w_1) \ominus w_1, \\
&= \mathbf{bnd}^{\text{out}}(ab^{11}c^3d^9e) = a^2b^9cd^6e^3f^3 \ominus ab^{11}c^3d^9e = ae^2f^3.
\end{aligned}$$

Computation of  $\mathbf{bnd}^{\text{in}}(w_1)$  By Definition 13,

$$\begin{aligned}
\mathbf{bnd}^{\text{in}}(w_1) &= \mathbf{bnd}(w_1) \ominus \mathbf{bnd}^{\text{out}}(w_1) = \mathbf{b}(w_1) \ominus \mathbf{bnd}^{\text{out}}(w_1), \\
&= \mathbf{bnd}^{\text{in}}(ab^{11}c^3d^9e) = a^2b^9cd^6e^3f^3 \ominus ae^2f^3 = ab^9cd^6e.
\end{aligned}$$

The computation of inside/outside membrane boundaries can easily be carried out when the msets are represented in Parikh vector form:

1. in Table 4, the rows 2, 3, 5 contain Parikh representations of  $\mathbf{bnd}(w_1)$ ,  $w_1$ , and  $\mathbf{bnd}^{\text{out}}(w_1)$ , respectively;
2. in Table 5, the rows 2, 3, 5 contain Parikh representations of  $\mathbf{bnd}(w_1)$ ,  $\mathbf{bnd}^{\text{out}}(w_1)$ , and  $\mathbf{bnd}^{\text{in}}(w_1)$ , respectively.

#### 4.4 Executions of Communication Rules without Membrane Boundary

For the sake of simplicity and in order to show the differences between two types of computations (without and with membrane boundaries) in the present and the next subsections the communication rules are executed in sequential (one-parallel) way. In this subsection, the communication rules are executed in the region  $w_1 = ab^{11}c^3d^9e$ , i.e., without membrane boundary.

Let the execution sequence of the communication rules as follows:

$$\langle ac; out \rangle, \langle b^6d^6; out \rangle, \langle d^3e; out \rangle.$$

The steps of computation process *without* membrane boundary are the following:

**Step 1.** Since  $ac \sqsubseteq w_1 = ab^{11}c^3d^9e$ , the rule  $\langle ac; out \rangle$  is applicable. This rule indicates that the mset  $ac$  exits the membrane  $w_1$  and goes to the environment:

$$ab^{11}c^3d^9e \xrightarrow{\langle ac; out \rangle} b^{11}c^2d^9e.$$

**Step 2.** Since  $b^6d^6 \sqsubseteq w_1 = b^{11}c^2d^9e$ , the rule  $\langle b^6d^6; out \rangle$  is applicable. This rule indicates that the mset  $b^6d^6$  exits the membrane  $w_1$  and goes to the environment:

$$b^{11}c^2d^9e \xrightarrow{\langle b^6d^6; out \rangle} b^5c^2d^3e.$$

**Step 3.** Since  $d^3e \sqsubseteq w_1 = b^5c^2d^3e$ , the rule  $\langle d^3e; out \rangle$  is applicable. This rule indicates that the mset  $d^3e$  exits the membrane  $w_1$  and goes to the environment:

$$b^5c^2d^3e \xrightarrow{\langle d^3e; out \rangle} b^5c^2.$$

Consequently, in this case, the  $P$  system  $\Pi$  halts with the mset  $b^5c^2$  in the region  $w_1$ .

#### 4.5 Executions of Communication Rules with Membrane Boundary

In this subsection, the executions of communication rules are constrained to the membrane boundary

$$\mathbf{bnd}(w_1) = a^2b^9cd^6e^3f^3 = \oplus_1 a^2b \sqcup \oplus_1 abcdef \sqcup \oplus_3 b^3d^2f \sqcup \oplus_1 e^3.$$

In other words, the communication rules work in the joint membrane approximation space  $\mathbf{MAS}(\Pi)$ . A communication rule of the form  $\langle u; out \rangle$  is applicable if  $u \sqsubseteq \mathbf{bnd}^{\text{in}}(w_1) = ab^9cd^6e$ . If the rule  $\langle u; out \rangle$  is applicable, the mset  $u$  exits the inside membrane boundary  $\mathbf{bnd}^{\text{in}}(w_1) = ab^9cd^6e$  and enters the outside membrane boundary  $\mathbf{bnd}^{\text{out}}(w_1) = ae^2f^3$ . These movements work within the base msets solely which ensures that the joint membrane approximation space will not change during the executions of communication rules.

As before, let the execution sequence of the communication rules as follows:

$$\langle ab; out \rangle, \langle b^6 d^6; out \rangle, \langle d^3 e; out \rangle.$$

Since the inside/outside boundaries are msets but not membranes, the transition of a communication rule execution is denoted by  $\longrightarrow$ .

The steps of computation process *with* membrane boundary are the following:

**Step 1.** Since  $ac \sqsubseteq \mathbf{bnd}^{\text{in}}(w_1) = ab^9 cd^6 e$ , the rule  $\langle ac; out \rangle$  is applicable. This rule indicates that the mset  $ac$  exits the inside membrane boundary  $ab^9 cd^6 e$  and enters outside membrane boundary  $ae^2 f^3$ :

$$(ab^9 cd^6 e, ae^2 f^3) \xrightarrow{\langle ac; out \rangle} (b^9 d^6 e, a^2 ce^2 f^3).$$

**Step 2.** Since  $b^6 d^6 \sqsubseteq \mathbf{bnd}^{\text{in}}(w_1) = b^9 d^6 e$ , the rule  $\langle b^6 d^6; out \rangle$  is applicable. This rule indicates that the mset  $b^6 d^6$  exits the inside membrane boundary  $b^9 d^6 e$  and enters outside membrane boundary  $a^2 ce^2 f^3$ :

$$(b^9 d^6 e, a^2 ce^2 f^3) \xrightarrow{\langle b^6 d^6; out \rangle} (b^3 e, a^2 b^6 cd^6 e^2 f^3).$$

**Step 3.** Since  $d^3 e \not\sqsubseteq \mathbf{bnd}^{\text{in}}(w_1) = b^3 e$ , the rule  $\langle d^3 e; out \rangle$  is not applicable.

In sum, the msets  $ac$ ,  $b^6 d^6$  exit the  $w_1 = ab^{11} c^3 d^9 e$ , i.e.,

$$ab^{11} c^3 d^9 e \xrightarrow{\langle ac; out \rangle, \langle b^6 d^6; out \rangle} b^5 c^2 d^3 e.$$

Therefore, in this case, the  $P$  system  $II$  halts with the mset  $b^5 c^2 d^3 e$  in the region  $w_1$ , the mset  $b^3 e$  in the inside membrane boundary  $\mathbf{bnd}^{\text{in}}(w_1)$ , and the mset  $a^2 b^6 cd^6 e^2 f^3$  in the outside membrane boundary  $\mathbf{bnd}^{\text{out}}(w_1)$ .

## Acknowledgments

The publication was supported by the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project. The project has been supported by the European Union, co-financed by the European Social Fund.

The authors are thankful to György Vaszil and the anonymous referees for valuable suggestions.

**Table 1.** Computation of  $l(w_1) = l(ab^{11}c^3d^9e)$ 

| $\mathfrak{B}$    | $\oplus_1 B \sqsubseteq^? w_1$ | $\oplus_2 B \sqsubseteq^? w_1$ | $\oplus_3 B \sqsubseteq^? w_1$    | $\oplus_4 B \sqsubseteq^? w_1$   |
|-------------------|--------------------------------|--------------------------------|-----------------------------------|----------------------------------|
| $a^2b$            | $a^2b \not\sqsubseteq^1$       | -                              | -                                 | -                                |
| $abcdef$          | $abcdef \not\sqsubseteq^1$     | -                              | -                                 | -                                |
| $\boxed{ac}$      | $\boxed{ac} \sqsubseteq^1$     | $a^2c^2 \not\sqsubseteq^1$     | -                                 | -                                |
| $\boxed{b^3cd^2}$ | $b^3cd^2 \sqsubseteq^1$        | $b^6c^2d^4 \sqsubseteq^1$      | $\boxed{b^9c^3d^6} \sqsubseteq^1$ | $b^{12}c^4d^8 \not\sqsubseteq^1$ |
| $\boxed{b^3d^2}$  | $b^3d^2 \sqsubseteq^1$         | $b^6d^4 \sqsubseteq^1$         | $\boxed{b^9d^6} \sqsubseteq^1$    | $b^{12}d^8 \not\sqsubseteq^1$    |
| $b^3d^2f$         | $b^3d^2f \not\sqsubseteq^1$    | -                              | -                                 | -                                |
| $\boxed{c}$       | $c \sqsubseteq^1$              | $c^2 \sqsubseteq^1$            | $\boxed{c^3} \sqsubseteq^1$       | $c^4 \not\sqsubseteq^1$          |
| $e^3$             | $e^3 \not\sqsubseteq^1$        | -                              | -                                 | -                                |
| $f^2$             | $f^2 \not\sqsubseteq^1$        | -                              | -                                 | -                                |
| $f^4$             | $f^4 \not\sqsubseteq^1$        | -                              | -                                 | -                                |

**Table 2.** Computation of  $b(w_1) = b(ab^{11}c^3d^9e)$ 

|                   |                           | Let $B' = B \sqcap w_1$ |                                 |                                     |                                 |                                 |
|-------------------|---------------------------|-------------------------|---------------------------------|-------------------------------------|---------------------------------|---------------------------------|
| $\mathfrak{B}$    | $B \not\sqsubseteq^? w_1$ | $B' \neq \emptyset$     | $\oplus_1 B' \sqsubseteq^? w_1$ | $\oplus_2 B' \sqsubseteq^? w_1$     | $\oplus_3 B' \sqsubseteq^? w_1$ | $\oplus_4 B' \sqsubseteq^? w_1$ |
| $\boxed{a^2b}$    | $\not\sqsubseteq$         | $ab \neq \lambda$       | $\boxed{ab \sqsubseteq^1}$      | $a^2b^2 \not\sqsubseteq^1$          | -                               | -                               |
| $\boxed{abcdef}$  | $\not\sqsubseteq$         | $abcde \neq \lambda$    | $\boxed{abcde \sqsubseteq^1}$   | $a^2b^2c^2d^2e^2 \not\sqsubseteq^1$ | -                               | -                               |
| $ac$              | $\sqsubseteq$             | -                       | -                               | -                                   | -                               | -                               |
| $b^3cd^2$         | $\sqsubseteq$             | -                       | -                               | -                                   | -                               | -                               |
| $b^3d^2$          | $\sqsubseteq$             | -                       | -                               | -                                   | -                               | -                               |
| $\boxed{b^3d^2f}$ | $\not\sqsubseteq$         | $b^3d^2 \neq \lambda$   | $b^3d^2 \sqsubseteq^1$          | $b^6d^4 \sqsubseteq^1$              | $\boxed{b^9d^6 \sqsubseteq^1}$  | $b^{12}d^8 \not\sqsubseteq^1$   |
| $c$               | $\sqsubseteq$             | -                       | -                               | -                                   | -                               | -                               |
| $\boxed{e^3}$     | $\not\sqsubseteq$         | $e \neq \lambda$        | $\boxed{e \sqsubseteq^1}$       | $e^2 \not\sqsubseteq^1$             | -                               | -                               |
| $f^2$             | $\not\sqsubseteq$         | $= \lambda$             | -                               | -                                   | -                               | -                               |
| $f^4$             | $\not\sqsubseteq$         | $= \lambda$             | -                               | -                                   | -                               | -                               |

**Table 3.** Computation of the base mset decomposition of  $u(w_1) = a^2b^9c^3d^6e^3f^3$ 

| $\mathfrak{B}$    | $\oplus_1 B \sqsubseteq^1 u(w_1)$ | $\oplus_2 B \sqsubseteq^1 u(w_1)$          | $\oplus_3 B \sqsubseteq^1 u(w_1)$      | $\oplus_4 B \sqsubseteq^1 u(w_1)$ |
|-------------------|-----------------------------------|--|--|-----------------------------------|
| $\boxed{a^2b}$    | $\boxed{a^2b} \sqsubseteq^1$      | $a^4b^2 \not\sqsubseteq^1$                 | -                                      | -                                 |
| $\boxed{abcdef}$  | $abcdef \sqsubseteq^1$            | $\boxed{a^2b^2c^2d^2e^2f^2} \sqsubseteq^1$ | $a^3b^3c^3d^3e^3f^3 \not\sqsubseteq^1$ | -                                 |
| $\boxed{ac}$      | $ac \sqsubseteq^1$                | $\boxed{a^2c^2} \sqsubseteq^1$             | $a^4c^4 \not\sqsubseteq^1$             | -                                 |
| $\boxed{b^3cd^2}$ | $b^3cd^2 \sqsubseteq^1$           | $b^6c^2d^4 \sqsubseteq^1$                  | $\boxed{b^9c^3d^6} \sqsubseteq^1$      | $b^{12}c^4d^8 \not\sqsubseteq^1$  |
| $\boxed{b^3d^2}$  | $b^3d^2 \sqsubseteq^1$            | $b^6d^4 \sqsubseteq^1$                     | $\boxed{b^9d^6} \sqsubseteq^1$         | $b^{12}d^8 \not\sqsubseteq^1$     |
| $\boxed{b^3d^2f}$ | $b^3d^2f \sqsubseteq^1$           | $b^6d^4f^2 \sqsubseteq^1$                  | $\boxed{b^9d^6f^3} \sqsubseteq^1$      | $b^{12}d^8f^4 \not\sqsubseteq^1$  |
| $\boxed{c}$       | $c \sqsubseteq^1$                 | $c^2 \sqsubseteq^1$                        | $\boxed{c^3} \sqsubseteq^1$            | $c^3 \not\sqsubseteq^1$           |
| $\boxed{e^3}$     | $\boxed{e^3} \sqsubseteq^1$       | $e^6 \not\sqsubseteq^1$                    | -                                      | -                                 |
| $\boxed{f^2}$     | $\boxed{f^2} \sqsubseteq^1$       | $f^4 \not\sqsubseteq^1$                    | -                                      | -                                 |
| $f^4$             | $f^4 \not\sqsubseteq^1$           | -  | -                                      | -                                 |

**Table 4.** Computation of  $\text{bnd}^{\text{out}}(w_1)$ 

|  | a | b  | c  | d  | e | f |
|--|---|----|----|----|---|---|
| $\text{bnd}(w_1) = \mathbf{b}(w_1) = a^2b^9cd^6e^3f^3$                 | 2 | 9  | 1  | 6  | 3 | 3 |
| $w_1 = ab^{11}c^3d^9e$   | 1 | 11 | 3  | 9  | 1 | 0 |
| row 2 – row 3  | 1 | -2 | -2 | -3 | 2 | 3 |
| $\text{bnd}^{\text{out}}(w_1) = \text{bnd}(w_1) \ominus w_1 = ae^2f^3$ | 1 | 0  | 0  | 0  | 2 | 3 |

**Table 5.** Computation of  $\text{bnd}^{\text{in}}(w_1)$ 

|  | a | b | c | d | e | f |
|--|---|---|---|---|---|---|
| $\text{bnd}(w_1) = \mathbf{b}(w_1) = a^2b^9cd^6e^3f^3$   | 2 | 9 | 1 | 6 | 3 | 3 |
| $\text{bnd}^{\text{out}}(w_1) = ae^2f^3$   | 1 | 0 | 0 | 0 | 2 | 3 |
| row 2 – row 3  | 1 | 9 | 1 | 6 | 1 | 0 |
| $\text{bnd}^{\text{in}}(w_1) = \text{bnd}(w_1) \ominus \text{bnd}^{\text{out}}w_1 = ab^9cd^6e$ | 1 | 9 | 1 | 6 | 1 | 0 |

## References

1. Barbuti, R., Maggiolo-Schettini, A., Milazzo, P., Pardini, G., Tesei, L.: Spatial P systems. *Natural Computing* 10(1), 3–16 (2011)
2. Birkhoff, G.: *Lattice theory*, Colloquium Publications, vol. 25. American Mathematical Society, Providence, Rhode Island, 3rd edn. (1967)
3. Cardelli, L., Gardner, P.: Processes in space. In: Ferreira, F., Löwe, B., Mayordomo, E., Gomes, L.M. (eds.) *CiE. Lecture Notes in Computer Science*, vol. 6158, pp. 78–87. Springer (2010)
4. Csuhaj-Varjú, E., Gheorghe, M., Stannett, M.: P systems controlled by general topologies. In: Durand-Lose, J., Jonoska, N. (eds.) *Unconventional Computation and Natural Computation - 11th International Conference, UCNC 2012, Orléan, France, September 3-7, 2012. Proceedings. LNCS*, vol. 7445, pp. 70–81. Springer, Berlin Heidelberg (2012)
5. Davey, B.A., Priestley, H.A.: *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, second edition edn. (2002)
6. Girish, K.P., John, S.J.: Relations and functions in multiset context. *Information Sciences* 179(6), 758–768 (2009)
7. Grätzer, G.: *General Lattice Theory*. Birkhäuser Verlag, Basel und Stuttgart (1978)
8. Grzymala-Busse, J.: Learning from examples based on rough multisets. In: *Proceedings of the Second International Symposium on Methodologies for intelligent systems*. pp. 325–332. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands (1987)
9. Kudlek, M., Martín-Vide, C., Păun, G.: Toward a formal macroset theory. In: Calude, C., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *WMP. LNCS*, vol. 2235, pp. 123–134. Springer-Verlag, Berlin Heidelberg (2001)
10. Mihálydeák, T., Csajbók, Z.: Membranes with local environments. In: Csuhaj-Varjú, E., Gheorghe, M., Vaszil, G. (eds.) *13th International Conference on Membrane Computing, CMC13, Budapest, Hungary, August 28 - 31, 2012. Proceedings*. pp. 311–322. MTA SZTAKI, the Computer and Automation Research Institute of the Hungarian Academy of Sciences, Budapest, Hungary (2012)
11. Mihálydeák, T., Csajbók, Z.E.: Membranes with boundaries. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) *Membrane Computing. 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers. LNCS*, vol. 7762, pp. 277–294. Springer-Verlag, Berlin Heidelberg (2013)
12. Mihálydeák, T., Csajbók, Z.E.: Partial approximation of multisets and its applications in membrane computing. In: *Proceedings of the 2013 Joint Rough Set Symposium (JRS2013)*. LNCS, Springer-Verlag (in the fall of 2013), Forthcoming
13. Pawlak, Z.: Rough sets. *International Journal of Computer and Information Sciences* 11(5), 341–356 (1982)
14. Pawlak, Z.: *Rough Sets: Theoretical Aspects of Reasoning about Data*. Kluwer Academic Publishers, Dordrecht (1991)
15. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
16. Păun, G.: *Membrane Computing. An Introduction*. Springer-Verlag, Berlin (2002)
17. Păun, G., Rozenberg, G., Salomaa, A. (eds.): *The Oxford Handbook of Membrane Computing*. Oxford Handbooks, Oxford University Press, Inc., New York, NY, USA (2010)

# Towards High-level P Systems Programming using Complex Objects

Radu Nicolescu<sup>1</sup>, Florentin Ipat<sup>2</sup> and Huiling Wu<sup>1</sup>

<sup>1</sup> Department of Computer Science, University of Auckland,  
Private Bag 92019, Auckland, New Zealand  
`r.nicolescu@auckland.ac.nz`, `hwu065@aucklanduni.ac.nz`

<sup>2</sup> Department of Computer Science, University of Bucharest,  
Bucharest, Romania, and  
Department of Computer Science, University of Pitești,  
Pitești, Romania  
`florentin.ipate@ifsoft.ro`

**Abstract.** We develop and formalise our earlier complex objects proposal and show that it enables an efficient high-level programming of P systems.

**Keywords:** P systems, complex objects, generic rules, data structures, control flow, parallel composition, function calls, recursion, numerical P systems, NP-complete, applications.

## 1 Introduction

A P system is a formal parallel and distributed computational model inspired by the structure and interactions of living cells, introduced by Păun [16]; for a recent overview of the domain, see Păun et al.'s recent monograph [18]. Essentially, a P system is specified by its membrane structure, symbols and rules. The underlying structure is a *network* such as a digraph, a directed acyclic graph (dag) or a tree (which seems the most studied case). Each node, here better known as *cell*, transforms its content symbols and sends messages to its neighbours using formal rules inspired by rewriting systems. Rules of the same cell can be applied in parallel (where possible) and all cells work in parallel.

P modules can be *asynchronous*, in the sense used in distributed algorithms and in Nicolescu [13], admitting the more traditional *synchronous* definitions as a special case. Sometimes we also make a fine distinction between (i) generated objects that can be thought, as traditionally in P systems, as being messaged back to the current cell, via a sort of *loopback* channel, and (ii) generated objects which become *immediately* available for the *following* rules, a matrix grammars inspired approach, used by ElGindy et al. [6]. However, here we strictly focus on *single cell* systems, so all these fine distinctions can be safely ignored.

In P systems, the practically very important *modularity* can be achieved by two distinct complementary ways: (i) an *external* modularity, for recursively

aggregating groups of cells into higher order P modules, as described in Dinneen et al. [4], an approach which is not further discussed here, and (ii) an *internal* modularity, possible inside each cell, where we recursively aggregate objects and rules to form higher-order components, a more recent approach which is more systematically discussed and assessed in this paper.

This article presents evidence that *complex objects* can enable a *high-level* programming style, with *data structures*, *control flow*, and several useful *functional programming* elements. We have previously used complex objects to successfully model and even improve large practical applications, ranging from computer vision [9, 10, 8] to complex graph theoretical problems [15, 6] and to well-known critical distributed algorithms [19]. Here we attempt to generalise our field-proven methods and sketch how to apply similar techniques to other, more theoretical, domains: numerical P systems and NP-complete problems.

Because of space constraints, for the rest of the paper we assume that the reader is already familiar with basic definitions used in tissue-like transition P systems, including state based rules, weak priority, promoters and inhibitors. Section 2 presents a formal definition for complex objects, slightly beyond what we have earlier proposed [13, 6]. Section 3 shows how fundamental data structures, such as stacks, trees and dictionaries, can be built and processed using our proposals. Section 4 sketches the basic ideas behind an integer arithmetic package, which can be extended to a rational package. Section 5 covers control flow techniques which can be used to implement higher level operations such as branching statements, parallel compositions, sequential functions definitions and invocations. Section 6 proposes a high-level linguistic support for developing P system models in a simple functional style. Section 7 illustrates a couple of more theoretical applications, not attempted in our earlier modelling projects: numerical P systems and NP-complete problems. Note that the ideas of integer arithmetic, compositional properties and high-level programming, although in different settings, recall similar ideas also presented to carry out arithmetic and register-machine computation, for example, in [2, 11].

## 2 P systems with complex objects

### 2.1 Complex objects

We consider the following formal definition for *complex objects*, which are Prolog-like ground terms, which can include either lists of complex objects or dot-separated strings (here interpreted as sequences) of complex objects:

```

<complex-object> ::= <term-object>

<term-object> ::= <atom> | <functor-object> '(' <object-arguments> ') '

<functor-object> ::= <atom> | <complex-object>

<object-arguments> ::=  $\lambda$  | <object-list> | <object-sequence>

```



$\langle \text{object-list} \rangle ::= \langle \text{complex-object} \rangle (', ' \langle \text{complex-object} \rangle)^*$

$\langle \text{object-sequence} \rangle ::= \langle \text{complex-object} \rangle ('.' \langle \text{complex-object} \rangle)^*$

Atoms (simple objects) are typically denoted by lower case letters, such as  $a, b, c$ , possibly with indices. Example ground complex objects:  $a, a(), a(b, c), a(b(c)), a.b().c, a(b.c), a(b(c))(d(e)), a(b(c), d(e)), a(b(c), d.e), a(b(c).d(e))$ .

We typically reserve sequences to represent natural numbers. For example, considering that  $l$  represents the unary digit, then the following complex objects can be used to describe the contents of a virtual integer variable  $a$ :  $a()$  — the value of  $a$  is 0;  $a(l^3)$  — the value of  $a$  is 3.

We are considering to extend our string objects to mean *bags* (i.e. *multisets*), instead of sequences. This could be useful in some scenarios, but we are not following these ideas here.

## 2.2 Variables and pattern matching

Variables are used for *pattern matching* on object arguments and are typically denoted by uppercase letters, such as  $X, Y, Z$ , possibly with overbars, e.g.  $\bar{X}$ , and with indices, e.g.  $X_1, \bar{X}_2$ . Variable  $'_'$  (underscore) is a wild-card and is used when pattern matching is required but its value is not further used. Using variables require the following redefinitions:

$\langle \text{object-list} \rangle ::= \langle \text{var-or-object} \rangle (', ' \langle \text{var-or-object} \rangle)^*$

$\langle \text{var-or-object} \rangle ::= \langle \text{variable}_1 \rangle \mid \langle \text{complex-object} \rangle$

$\langle \text{object-sequence} \rangle ::= \langle \text{var-or-object-subsequence} \rangle ('.' \langle \text{var-or-object-subsequence} \rangle)^*$

$\langle \text{var-or-object-subsequence} \rangle ::= \langle \text{variable}_2 \rangle \mid \lambda \mid \langle \text{var-or-object} \rangle ('.' \langle \text{var-or-object} \rangle)^*$

With these definitions, a variable can match either:

1. a complex object in a list of arguments or in a string, or
2. any substring of a complex objects sequence, including  $\lambda$ .

Variables of the type 1 will be denoted by symbols without overbars and variables of type 2 will have overbars. For example:

- matching  $a(b(c), d.e.f) = a(X, d.\bar{Y})$  creates the bindings  $X, \bar{Y} = b(c), e.f$
- matching  $a.b().c = X.\bar{Y}$  creates the binding  $X, \bar{Y} = a, b().c$
- matching  $a.b().c = \bar{X}.Y$  creates the binding  $\bar{X}, Y = a.b(), c$
- matching  $a.b().c = \bar{X}.\bar{Y}$  nondeterministically creates one of the following bindings  $\bar{X}, \bar{Y} = \lambda, a.b().c, \bar{X}, \bar{Y} = a, b().c, \bar{X}, \bar{Y} = a.b(), c, \bar{X}, \bar{Y} = a.b().c, \lambda$

With the exception of subsequence matchings, our pattern matching rules are a simplified version of term unification in Prolog-like languages, so they can be implemented with reasonable efficiency. As we will later see, arithmetic operations are based on particular subsequence matchings on unary sequences: these matchings can also be efficiently implemented. However, general subsequence matchings could be expensive, so these should be prudently used, e.g. for proof-of-concept prototyping.

Type 2 (overbarred) variables and much of the pattern matching complexities have been mainly introduced to support efficient arithmetic operations (on unary sequences); the complex objects construction would look much simpler if we would accept natural numbers as primitives in our P modules.

### 2.3 Generic rules

By default, rules are applied top-down, in the so-called *weak priority* order. As we are here exclusively focusing on *single cell* systems, we only consider a simplified generic rule format (with no messaging), of the following type:

$$\text{current-state objects} \rightarrow_{\alpha} \text{target-state objects}' \mid \text{promoters} \neg \text{inhibitors},$$

where

- left-side objects, right-side objects', promoters and inhibitors are *bags of complex objects*, possibly containing (which makes rules *generic*) *variables*, which are *matched* (unified) as described in the previous section;
- $\alpha \in \{\text{min.min}, \text{min.max}, \text{max.min}, \text{max.max}\}$ , is a combined instantiation and rewriting mode, as discussed in Nicolescu et al. [13, 6] (discussion further adapted below).

To explain generics, consider a cell,  $\sigma$ , containing three counter-like complex objects,  $c(c(a))$ ,  $c(c(a))$ ,  $c(c(c(a)))$ , and all four possible instantiation.rewriting modes of the following “decrementing” rule:

$$(\rho_{\alpha}) S_1 c(c(X)) \rightarrow_{\alpha} S_2 c(X).$$

where  $\alpha \in \{\text{min.min}, \text{min.max}, \text{max.min}, \text{max.max}\}$ .

1. If  $\alpha = \text{min.min}$ , rule  $\rho_{\text{min.min}}$  nondeterministically generates *one* of the following rule instances:

$$\begin{aligned} (\rho'_1) S_1 c(c(a)) &\rightarrow_{\text{min}} S_2 c(a) \quad \text{or} \\ (\rho''_1) S_1 c(c(c(a))) &\rightarrow_{\text{min}} S_2 c(c(a)). \end{aligned}$$

In the first case, using  $(\rho'_1)$ , cell  $\sigma$  ends with counters  $c(a)$ ,  $c(c(a))$ ,  $c(c(c(a)))$ . In the second case, using  $(\rho''_1)$ , cell  $\sigma$  ends with counters  $c(c(a))$ ,  $c(c(a))$ ,  $c(c(a))$ .

2. If  $\alpha = \mathbf{max.min}$ , rule  $\rho_{\mathbf{max.min}}$  generates *both* following rule instances:

$$\begin{aligned} (\rho'_2) \quad S_1 \ c(c(a)) &\rightarrow_{\mathbf{min}} S_2 \ c(a) \quad \text{and} \\ (\rho''_2) \quad S_1 \ c(c(c(a))) &\rightarrow_{\mathbf{min}} S_2 \ c(c(a)). \end{aligned}$$

In this case, using  $(\rho'_2)$  and  $(\rho''_2)$ , cell  $\sigma$  ends with counters  $c(a)$ ,  $c(c(a))$ ,  $c(c(a))$ .

3. If  $\alpha = \mathbf{min.max}$ , rule  $\rho_{\mathbf{min.max}}$  nondeterministically generates *one* of the following rule instances:

$$\begin{aligned} (\rho'_3) \quad S_1 \ c(c(a)) &\rightarrow_{\mathbf{max}} S_2 \ c(a) \quad \text{or} \\ (\rho''_3) \quad S_1 \ c(c(c(a))) &\rightarrow_{\mathbf{max}} S_2 \ c(c(a)). \end{aligned}$$

In the first case, using  $(\rho'_3)$ , cell  $\sigma$  ends with counters  $c(a)$ ,  $c(a)$ ,  $c(c(c(a)))$ .  
In the second case, using  $(\rho''_3)$ , cell  $\sigma$  ends with counters  $c(c(a))$ ,  $c(c(a))$ ,  $c(c(a))$ .

4. If  $\alpha = \mathbf{max.max}$ , rule  $\rho_{\mathbf{min.max}}$  generates *both* following rule instances:

$$\begin{aligned} (\rho'_4) \quad S_1 \ c(c(a)) &\rightarrow_{\mathbf{max}} S_2 \ c(a) \quad \text{and} \\ (\rho''_4) \quad S_1 \ c(c(c(a))) &\rightarrow_{\mathbf{max}} S_2 \ c(c(a)). \end{aligned}$$

In this case, using  $(\rho'_4)$  and  $(\rho''_4)$ , cell  $\sigma$  ends with counters  $c(a)$ ,  $c(a)$ ,  $c(c(a))$ .

The interpretation of  $\mathbf{min.min}$ ,  $\mathbf{min.max}$  and  $\mathbf{max.max}$  modes is straightforward. While other interpretations could be considered, the mode  $\mathbf{max.min}$  indicates that the generic rule is instantiated as *many* times as possible, without *superfluous* instances (i.e. without duplicates or instances which are not applicable) and each one of the instantiated rules is applied *once*, if possible.

For all modes, the instantiations are *conceptually* created when rules are tested for applicability and are also *ephemeral*, i.e. they disappear at the end of the step. P system implementations are encouraged to directly apply high-level generic rules, if this is more efficient (it usually is); they may, but need not, start by transforming high-level rules into low-level rules, by way of instantiations.

This type of generic rules allow (i) a reasonably fast parsing and processing of subcomponents, and (ii) algorithm descriptions with *fixed size alphabets* and *fixed sized rulesets*, independent of the size of the problem and number of cells in the system (sometimes impossible with only atomic symbols).

### 3 Data structures

#### 3.1 Stacks

A  $n$ -size *stack*  $s$ , with contents  $a_1, a_2 \dots a_{n-1}, a_n$  (top), can be represented by a complex object  $s(a_n(a_{n-1}(\dots a_2(a_1()) \dots)))$ . Essentially, this is a *simple linked list* where the list head is the stack top. Examples:  $s()$  — an empty stack,  $s$ ;  $s(a(b(c())))$  — a stack,  $s$ , with contents  $a, b, c$ .

**Fundamental operations** on stacks include:

- *construct* an empty stack

$$S_1 \rightarrow_{\min.\min} S_2 s()$$

- replace  $a$  by  $b$ , if  $s$  is *empty*

$$S_1 a \rightarrow_{\min.\min} S_2 b \mid s()$$

- *clear* a stack

$$S_1 s(X) \rightarrow_{\min.\min} S_2 s()$$

- *push*  $a$ , if  $a$  is in the current contents

$$S_1 a s(X) \rightarrow_{\min.\min} S_2 s(a(X))$$

- *push* the content of  $c$ , if this exists and is a term (not sequence)

$$S_1 c(T) s(X) \rightarrow_{\min.\min} S_2 s(T(X))$$

- conditional *pop*  $a$ , if  $a$  is on top

$$S_1 s(a(X)) \rightarrow_{\min.\min} S_2 a s(X)$$

- unconditional *pop*, if  $s$  is not empty

$$S_1 s(T(X)) \rightarrow_{\min.\min} S_2 T s(X)$$

- conditional *peek*  $a$ , if  $a$  is on top

$$S_1 s(a(X)) \rightarrow_{\min.\min} S_2 a s(a(X))$$

- unconditional *peek*, if  $s$  is not empty

$$S_1 s(T(X)) \rightarrow_{\min.\min} S_2 T s(T(X))$$

- *reverse* stack  $s$  on stack  $t$

$$S_1 s(T(X)) t(Y) \rightarrow_{\max.\min} S_2 s(X) t(T(Y))$$

**Complexity.** Each of the above stack operations can be accomplished in a single P step,  $O(1)$ , except the stack reversal, which may take longer (in this case the number of steps required equals the length of the stack).

**Extensions.** All preceding *stack* operations can be formally redefined to work on strings, instead of nested terms. *Queues* can also be implemented as strings, essentially by renaming pop as dequeue, and replacing push by an enqueue operation (adding to the other end):

- unconditional *dequeue*, if  $q$  is not empty

$$S_1 q(T.\overline{X}) \rightarrow_{\min.\min} S_2 T q(\overline{X})$$

- *enqueue* the content of  $c$ , if this exists and assuming is a term (not sequence)

$$S_1 c(T) q(\overline{X}) \rightarrow_{\min.\min} S_2 q(\overline{X}.T)$$

Alternatively, *queues* can be also be implemented as pairs of stacks, using stack reversals when needed. This can be reasonably efficient, as reversal costs will normally amortize in the long run.

### 3.2 Trees

Trees can be represented as nested terms, in a straightforward manner. For example: (1) a leaf node with contents  $X$  can be represented as  $f(X)$ ; (2) an intermediary node, with contents  $X$  and two subnodes, can be represented as  $n(X, Y, Z)$ , where  $Y$  and  $Z$  can be leaves or other intermediary nodes. For example, the following term describes a binary tree consisting of 2 intermediary nodes and 3 leaves, all with integer contents:

$$n(l^{10}, n(l^{20}, f(l^{30}), f(l^{40})), f(l^{50}))$$

Most tree operations are either recursive or have rather elaborate descriptions (needed to simulate recursion). As recursion is discussed later in the article, here we only show a simple operation which, in P systems, does not really need recursion: a destructive summation of all values in a binary tree,  $n$ , with integer contents. The first rule creates placeholder for the total sum,  $s$ , and stores a copy of the original tree in a backup store,  $b$ :

$$\begin{array}{ll} r_1 : S_0 n(X, Y, Z) & \rightarrow_{\min.\min} S_1 n(X, Y, Z) s() b(n(X, Y, Z)) \\ r_2 : S_1 s(T) v(X) & \rightarrow_{\min.\min} S_1 s(T.X) \\ r_3 : S_1 v(X) v(Y) & \rightarrow_{\max.\min} S_1 v(X.Y) \\ r_4 : S_1 f(X) & \rightarrow_{\max.\min} S_1 v(X) \\ r_5 : S_1 n(X, f(Y), f(Z)) & \rightarrow_{\max.\min} S_1 v(X) v(Y) v(Z) \\ r_6 : S_1 n(X, f(Y), n(Z, Z_1, Z_2)) & \rightarrow_{\max.\min} S_1 v(X) v(Y) n(Z, Z_1, Z_2) \\ r_7 : S_1 n(X, n(Y, Y_1, Y_2), f(Z)) & \rightarrow_{\max.\min} S_1 v(X) n(Y, Y_1, Y_2) v(Z) \\ r_8 : S_1 n(X, n(Y, Y_1, Y_2), n(Z, Z_1, Z_2)) & \rightarrow_{\max.\min} S_1 v(X) n(Y, Y_1, Y_2) n(Z, Z_1, Z_2) \end{array}$$

For the above sample tree, the result is  $s(l^{150})$ , as indicated by the following traces, where  $b(\dots)$  represents the backed up tree,  $b(n(l^{10}, n(l^{20}, f(l^{30}), f(l^{40})), f(l^{50})))$ :

$$\begin{aligned}
& n(l^{10}, n(l^{20}, f(l^{30}), f(l^{40})), f(l^{50})) \\
& \xRightarrow{r_1} s() \ n(l^{10}, n(l^{20}, f(l^{30}), f(l^{40})), f(l^{50})) \ b(\dots) \\
& \xRightarrow{r_7} s() \ v(l^{10}) \ n(l^{20}, f(l^{30}), f(l^{40})) \ v(l^{50}) \ b(\dots) \\
& \xRightarrow{r_2} s(l^{10}) \ n(l^{20}, f(l^{30}), f(l^{40})) \ v(l^{50}) \ b(\dots) \\
& \xRightarrow{r_5} s(l^{10}) \ v(l^{20}) \ v(l^{30}) \ v(l^{40}) \ v(l^{50}) \ b(\dots) \\
& \xRightarrow{r_2} s(l^{30}) \ v(l^{30}) \ v(l^{40}) \ v(l^{50}) \ b(\dots) \\
& \xRightarrow{r_3} s(l^{30}) \ v(l^{120}) \ b(\dots) \\
& \xRightarrow{r_2} s(l^{150}) \ b(\dots)
\end{aligned}$$

The complexity of this snippet is  $O(h)$  P steps, where  $h$  is the height of the tree.

### 3.3 Dictionaries

Dictionaries are key/value mappings. Typical dictionaries have unique keys; their efficient implementations use hash tables or balanced trees (e.g. red-black trees). A dictionary,  $d$ , can be represented by a string of complex objects of the form  $m(k, v)$ , where  $k$  is the key and  $v$  is the value. Examples:  $d()$  — an empty dictionary,  $d$ ;  $d(m(a, b).m(c, d))$  — a dictionary,  $d$ , with two mappings,  $a \rightarrow b$  and  $c \rightarrow d$ .

**Fundamental operations** on dictionaries include:

- *construct* an empty dictionary

$$S_1 \rightarrow_{\min.\min} S_2 \ d()$$

- *clear* a dictionary

$$S_1 \ d(\overline{X}) \rightarrow_{\min.\min} S_2 \ d()$$

- *add*  $a \rightarrow b$ , if key  $a$  is not already present (to preserve key uniqueness)

$$S_1 \ m(a, b) \ d(\overline{X}) \rightarrow_{\min.\min} S_2 \ d(m(a, b).\overline{X}) \neg d(\overline{Y}.m(a, V).\overline{Z})$$

- non-destructive *query* of the mapping for key  $a$ , if it exists

$$S_1 \ a \ d(\overline{X}.m(a, V).\overline{Y}) \rightarrow_{\min.\min} S_2 \ m(a, V) \ d(\overline{X}.m(a, V).\overline{Y})$$

- *reset* the mapping for key  $a$  to a new value, if  $a$  has a mapping (and also return the old value for this key)

$$S_1 \ m(a, b) \ d(\overline{X}.m(a, V).\overline{Y}) \rightarrow_{\min.\min} S_2 \ m(a, V) \ d(\overline{X}.m(a, b).\overline{Y})$$

- *remove* the mapping for key  $a$ , if it exists

$$S_1 \ a \ d(\overline{X}.m(a, V).\overline{Y}) \rightarrow_{\min.\min} S_2 \ d(\overline{X}.\overline{Y})$$

**Complexity.** Apparently, each of the above dictionary operations can be accomplished in a single P step,  $O(1)$ . However, these rules use a generalized string unification which probably is not efficient for practical purposes. Thus, this dictionary structure should be reserved for theoretical proofs-of-concept or prototype implementations.

Assuming a natural order on atoms, we can define a more efficient dictionary implementation based on balanced trees; however, we are not following this idea here.

## 4 Arithmetic

Recall that we use complex objects with sequence contents to represent natural numbers. For example, considering that  $l$  represents the unary digit, then the following complex objects can indicate that:  $a()$  — the value of  $a$  is 0;  $a(l^3)$  — the value of  $a$  is 3.

**Fundamental arithmetic operations** on natural numbers include:

- $c := a + b$ , destructive *addition*:

$$S_1 a(\bar{X}) b(\bar{Y}) \rightarrow_{\min.\min} S_2 c(\bar{X}.\bar{Y})$$

- $c := a - b$ , destructive *subtraction*:

$$S_1 a(\bar{X}.\bar{Y}) b(\bar{Y}) \rightarrow_{\min.\min} S_2 c(\bar{X})$$

- $c := a * b$ , *multiplication*, which destroys  $a$ :

$$\begin{array}{ccc} S_1 & & \rightarrow_{\min.\min} S_2 c() \\ S_2 a(l.\bar{X}) b(\bar{Y}) c(\bar{Z}) & \rightarrow_{\max.\min} & S_2 a(\bar{X}) b(\bar{Y}) c(\bar{Y}.\bar{Z}) \end{array}$$

- $c, d := a / b, a \% b$ , *division*, which destroys  $a$ :

$$\begin{array}{ccc} S_1 & & \rightarrow_{\min.\min} S_2 c() \\ S_2 a(\bar{X}.\bar{Y}) b(\bar{Y}) c(\bar{Z}) & \rightarrow_{\max.\min} & S_2 a(\bar{X}) b(\bar{Y}) c(l \bar{Z}) \\ S_2 a(\bar{X}) & \rightarrow_{\max.\min} & S_3 d(\bar{X}) \end{array}$$

**Complexity.** Additions and subtractions can be performed in single P steps,  $O(1)$ , but multiplications and divisions may take longer. For multiplication, the number of steps equals the value of  $a$  plus one, whereas for division this is the value of the quotient  $c$  plus two.

If desired, non destructive operations can be implemented in a straightforward manner. Alternatively, we can define arithmetic operations using counter stacks, but this is much slower.

These ideas can be extended to define more complete arithmetic packages for integer numbers and for rational numbers.

## 5 Control flow

Composing bigger chunks out of smaller rule snippets can require careful object relabelling, to ensure continuity and avoid clashes. This is probably best done automatically, using a well designed composition model. However, we do not follow this here; we just present a proof of concept where all required relabeling has been manually done.

### 5.1 Basic composition

Basic composition includes sequencing and conditional transfers, which can be further used to define higher-level structured constructs, such as if-then-else conditionals and while loops (not detailed here).

- $\text{BR}(S')$ , *branch*, unconditional branch to state  $S'$ :

$$S \rightarrow_{\text{min.min}} S'$$

- $\text{BP}(S'; p_1, p_2, \dots)$ , *branch on promoters*, branch to state  $S'$ , given promoters  $p_1, p_2, \dots$ :

$$S \rightarrow_{\text{min.min}} S' \mid p_1 p_2 \dots$$

- $\text{BI}(S'; i_1, i_2, \dots)$ , *branch on inhibitors*, branch to state  $S'$ , given inhibitors  $i_1, i_2, \dots$ :

$$S \rightarrow_{\text{min.min}} S' \neg i_1 i_2 \dots$$

- $\text{BPI}(S'; p_1, p_2, \dots; i_1, i_2, \dots)$ , *branch on promoters and inhibitors*, branch to state  $S'$ , given promoters  $p_1, p_2, \dots$  and inhibitors  $i_1, i_2, \dots$ :

$$S \rightarrow_{\text{min.min}} S' \mid p_1 p_2 \dots \neg i_1 i_2 \dots$$

Other branching primitives are described in the sections for function calls.

### 5.2 Parallel composition

Consider running in parallel two rule fragments,  $\Pi_1$ , with  $M$  states, and  $\Pi_2$ , with  $N$  states. In general, the composed system,  $\Pi_1 \times \Pi_2$ , will need  $M \cdot N$  states, thus it will need  $O(M \cdot N)$  rules.

However, using complex state objects, we can define an equivalent parallel system,  $\Pi_1 \parallel \Pi_2$ , with just  $O(M + N)$  rules — essentially the same rules initially used for describing  $\Pi_1$  and  $\Pi_2$ . Additional semantics is required for matching variables on components of state objects.

We illustrate this on a simple ad-hoc example, not doing any meaningful work, except that  $\Pi_1$  loops over three states and  $\Pi_2$  loops over two states.



- $\Pi_1$ , a fragment with 3 states and 3 rules:

$$\begin{aligned} S_1 a &\rightarrow_{\min} S_2 b \\ S_2 b &\rightarrow_{\min} S_3 c \\ S_3 c &\rightarrow_{\min} S_1 a \end{aligned}$$

- $\Pi_2$ , a fragment with 2 states and 2 rules:

$$\begin{aligned} S_1 d &\rightarrow_{\min} S_2 e \\ S_2 e &\rightarrow_{\min} S_1 d \end{aligned}$$

- $\Pi_1 \times \Pi_2$ , has 6 ( $= 3 \cdot 2$ ) states and 18 ( $= 3 \cdot 2 \cdot 3$ ) rules:

$$\begin{array}{ll} S_{11} a d \rightarrow_{\min} S_{22} b e & S_{22} b e \rightarrow_{\min} S_{31} c d \\ S_{11} a \rightarrow_{\min} S_{21} b & S_{22} b \rightarrow_{\min} S_{32} c \\ S_{11} d \rightarrow_{\min} S_{12} e & S_{22} e \rightarrow_{\min} S_{21} d \\ S_{12} a e \rightarrow_{\min} S_{21} b d & S_{31} c d \rightarrow_{\min} S_{12} a e \\ S_{12} a \rightarrow_{\min} S_{22} b & S_{31} c \rightarrow_{\min} S_{11} a \\ S_{12} e \rightarrow_{\min} S_{11} d & S_{31} d \rightarrow_{\min} S_{32} e \\ S_{21} b d \rightarrow_{\min} S_{32} c e & S_{32} c e \rightarrow_{\min} S_{11} a d \\ S_{21} b \rightarrow_{\min} S_{31} c & S_{32} c \rightarrow_{\min} S_{12} a \\ S_{21} d \rightarrow_{\min} S_{22} e & S_{32} e \rightarrow_{\min} S_{31} d \end{array}$$

- $\Pi_1 \parallel \Pi_2$ , also has 6 ( $= 3 \cdot 2$ ) states, but only 5 ( $= 3 + 2$ ) rules:

$$\begin{aligned} \Theta(S_1, Y) a &\rightarrow_{\min} \Theta(S_2, Y) b \\ \Theta(S_2, Y) b &\rightarrow_{\min} \Theta(S_3, Y) c \\ \Theta(S_3, Y) c &\rightarrow_{\min} \Theta(S_1, Y) a \\ \Theta(X, S_1) d &\rightarrow_{\min} \Theta(X, S_2) e \\ \Theta(X, S_2) e &\rightarrow_{\min} \Theta(X, S_1) d \end{aligned}$$

Note that, although  $\Pi_1 \parallel \Pi_2$  has, in general, an order of magnitude fewer user-written rules than  $\Pi_1 \times \Pi_2$ , as  $O(M + N) \ll O(M \cdot N)$ , their state sets are isomorphic. Figure 1 shows state charts for  $\Pi_1$ ,  $\Pi_2$  and  $\Pi_1 \times \Pi_2 \stackrel{\text{states}}{\simeq} \Pi_1 \parallel \Pi_2$ .

### 5.3 Parameterless sequential functions

We need states and a global stack for return states, let it be  $\rho()$ . Consider that:  $S_f$  is the entry state of function  $f$ 's ruleset,  $S_c$  is the current state and  $S_r$  is the return state (to be entered after function  $f$  completes). We define the following high-level boiler-plate P macros:

- **BAL**( $S_f, S_r$ ), *branch and link* to state  $S_f$ , i.e. to function  $f$ , and request return to state  $S_r$ :

$$S_c \rho(X) \rightarrow_{\min.\min} S_f \rho(S_r(X))$$

- **RET**, *return* from function  $f$  (assuming that its last state is  $S_g$ ):

$$S_g \rho(Z(X)) \rightarrow_{\min.\min} Z \rho(X)$$

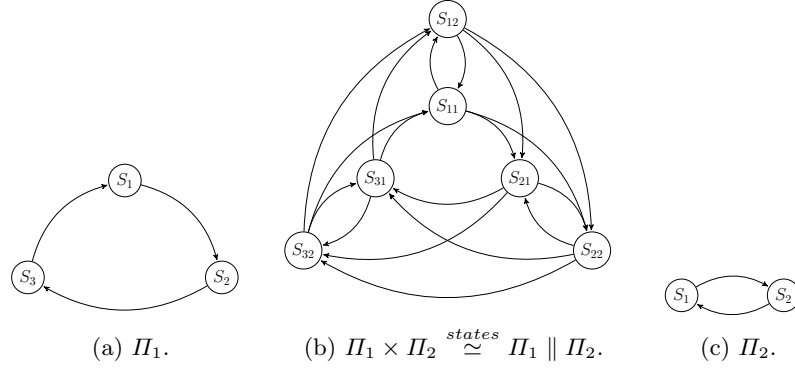


Fig. 1: State charts of  $H_1$ ,  $H_2$  and  $H_1 \times H_2 \stackrel{\text{states}}{\simeq} H_1 \parallel H_2$ .

#### 5.4 Sequential functions with parameters

We need one more stack for each parameter. Alternatively, we can combine all parameters in a single complex object, so just one additional stack would suffice. We can even combine this with the return stack, to mimic a typical runtime stack frame.

Here we consider a single global stack for all parameters,  $\pi$ , and a global placeholder for function results,  $\phi$ . If needed, but not shown here, global stack  $\pi$  could also be used to create slots for local variables. Additional high-level boiler-plate P macros:

- $\text{PUSHP}(p_1, p_2, \dots)$ , *push parameters*, push contents of objects with functors  $p_1, p_1, \dots$  on  $\pi$ , and create an empty  $\phi()$ , as a placeholder for the expected results (we assume that this does not yet exist):

$$S_c p_1(X_1) p_2(X_2) \dots \pi(X) \rightarrow_{\min.\min} S_c \pi(p(X_1, X_2, \dots)(X)) \phi()$$

- $\text{PEEKP}(p_1, p_2, \dots)$ , *peek parameters*, peek top of  $\pi$  into contents of objects with functors  $p_1, p_1, \dots$ :

$$S_c \rightarrow_{\min.\min} S_c p_1(X_1) p_2(X_2) \dots \mid \pi(p(X_1, X_2, \dots)(X))$$

- $\text{POPP}()$ , *pop parameters*, pop top of  $\pi$ :

$$S_c \pi(T(X)) \rightarrow_{\min.\min} S_c \pi(X)$$

- $\text{POPP}(p_1, p_2, \dots)$ , *pop parameters*, pop top of  $\pi$  into contents of objects with functors  $p_1, p_1, \dots$ :

$$S_c \pi(p(X_1, X_2, \dots)(X)) \rightarrow_{\min.\min} S_c p_1(X_1) p_2(X_2) \dots \pi(X)$$

- **RESULT**( $r_1, r_2, \dots$ ), *set result*, set  $\phi$  using contents of objects with functors  $r_1, r_2, \dots$ :

$$S_c r_1(X_1) r_2(X_2) \dots \phi() \rightarrow_{\min.\min} S_c \phi(X_1, X_2, \dots)$$

- **POPR**( $q_1, q_2, \dots$ ), *pop results*, extract  $\phi$ 's contents into objects with functors  $q_1, q_2, \dots$ :

$$S_c \phi(X_1, X_2, \dots) \rightarrow_{\min.\min} S_c q_1(X_1) q_2(X_2) \dots$$

For convenience, the following P macro combinations are also defined:

- **CALL**( $S_f; p_1, p_2, \dots; S_r; q$ ) = **PUSHP**( $p_1, p_2, \dots$ ); **BAL**( $S_f, S_r$ ); **POPR**( $q$ )
- **FUNC**( $p_1, p_2, \dots$ ) = **PEEKP**( $p_1, p_2, \dots$ )
- **RETURN**( $r_1, r_2, \dots$ ) = **RESULT**( $r_1, r_2, \dots$ ); **POPP**; **RET**

### 5.5 A first example

Consider a snippet calling an arithmetic multiply function, to compute  $z = x * y$ .

- Pseudo P code, using our high-level P macros (its essential two lines are exactly as given in Section 4):

|                                       |   |
|---------------------------------------|---|
| <i>% calling program</i>              | <i>% inputs: <math>x(\overline{X}) y(\overline{Y})</math></i>                       |
| $S_c \text{ CALL}(S_m; x, y; S_r; z)$ | <i>% PUSHP(<math>x, y</math>); BAL(<math>S_m, S_r</math>); POPR(<math>z</math>)</i> |
| $S_r \dots$                           | <i>% output: <math>z(\overline{Z})</math></i>                                       |

|   |  |
|---|--|
| <i>% function mult <math>S_m</math></i>                 |  |
| $S_m \text{ FUNC}(a, b)$                                | <i>% creates: <math>a(\overline{X}) b(\overline{Y})</math></i>                             |
| $S_m$   | $\rightarrow_{\min.\min} S_n c()$  |
| $S_n a(l.\overline{X}) b(\overline{Y}) c(\overline{Z})$ | $\rightarrow_{\max.\min} S_n a(\overline{X}) b(\overline{Y}) c(\overline{Y}.\overline{Z})$ |
| $S_n a(-) b(-)$   | $\rightarrow_{\min.\min} S_o$  |
| $S_o \text{ RETURN}(c)$                                 | <i>% RESULT(<math>c</math>); POPP; RET</i>   |

- Direct translation to P rules:

|   |  |
|---|--|
| <i>% calling program:</i>                               | $x(\overline{X}) y(\overline{Y})$  |
| $S_c x(\overline{X}) y(\overline{Y}) \pi(P)$            | $\rightarrow_{\min.\min} S_m \pi(p(\overline{X}, \overline{Y})(P) \phi())$                               |
| $S_c \rho(R)$   | $\rightarrow_{\min.\min} S_m \rho(S_r(R))$   |
| $S_r \phi(\overline{Z})$                                | $\rightarrow_{\min.\min} S_r z(\overline{Z})$  |
| $\dots$   |  |
| <i>% function mult <math>S_m</math></i>                 |  |
| $S_m$   | $\rightarrow_{\min.\min} S_n a(\overline{X}) b(\overline{Y}) \mid \pi(p(\overline{X}, \overline{Y})(P))$ |
| $S_m$   | $\rightarrow_{\min.\min} S_n c()$  |
| $S_n a(l.\overline{X}) b(\overline{Y}) c(\overline{Z})$ | $\rightarrow_{\max.\min} S_n a(\overline{X}) b(\overline{Y}) c(\overline{Y}.\overline{Z})$               |
| $S_n a(-) b(-)$   | $\rightarrow_{\min.\min} S_o$  |
| $S_o c(\overline{Z}) \phi()$                            | $\rightarrow_{\min.\min} S_o \phi(\overline{Z})$   |
| $S_o \pi(T(P))$   | $\rightarrow_{\min.\min} S_o \pi(P)$   |
| $S_o \rho(Z(R))$  | $\rightarrow_{\min.\min} Z \rho(R)$  |

### 5.6 A recursive example

As a more elaborated example, consider the classical naive definition of factorial:

```
fact n = if n = 0 then 1 else (fact (n-1)) * n
```

The following versions show the call  $y = \text{fact } x$ .

- Pseudo P code, using our high-level P macros:

|   |                           |  |
|---|---------------------------|--|
| <i>% calling program</i>                | —                         | <i>% input: <math>x(\overline{X})</math></i>                                     |
| $S_c \text{ CALL}(S_f; x; S_r; y)$      |                           | <i>% PUSH<math>P(x)</math>; BAL(<math>S_f, S_r</math>); POP<math>R(y)</math></i> |
| $S_r \dots$                             | —                         | <i>% output: <math>y(\overline{Y})</math></i>                                    |
|   |                           |  |
| <i>% function fact <math>S_f</math></i> | —                         | <i>defined with macros</i>   |
| $S_f \text{ FUNC}(n)$                   |                           | <i>% creates: <math>n(\overline{N})</math></i>                                   |
| $S_f n()$                               | $\rightarrow_{\min.\min}$ | $S_h f(l)$   |
| $S_f n(l.\overline{N})$                 | $\rightarrow_{\min.\min}$ | $S_f n(\overline{N})$  |
| $S_f \text{ CALL}(S_f; n; S_g; f)$      |                           | <i>% creates: <math>f(-)</math></i>  |
| $S_g \text{ PEEKP}(n)$                  |                           | <i>% recreates: <math>n(\overline{N})</math></i>                                 |
| $S_g \text{ CALL}(S_m; f, n; S_h; f)$   |                           | <i>% call mult</i>   |
| $S_h \text{ RETURN}(f)$                 |                           | <i>% RESULT(<math>f</math>); POP<math>P</math>; RET</i>                          |

- The above function definition can be more efficiently (but less readably) implemented by the following rules, which peek parameter values directly from the stack, inline the mult call and use two temporary objects,  $\sigma$  and  $\tau$ , to evaluate the product.

|  |                           |   |
|--|---------------------------|---|
| <i>% calling program</i>   | —                         | <i>% input: <math>x(\overline{X})</math></i>                                  |
| $S_c x(\overline{X}) \pi(Y) \rho(R)$                               | $\rightarrow_{\min.\min}$ | $S_f \phi() \pi(n(\overline{X})(Y)) \rho(S_r(R))$                             |
| $S_r \dots$  | —                         | <i>% output: <math>\phi(\overline{Y})</math></i>                              |
| ...  |                           |   |
| <i>% function fact <math>S_f</math></i>                            | —                         | <i>manually optimised code</i>  |
| $S_f \phi() \pi(n() (Y)) \rho(Z(X))$                               | $\rightarrow_{\min.\min}$ | $Z \phi(l) \pi(Y) \rho(X)$  |
| $S_f \phi() \pi(n(l.\overline{N})(Y)) \rho(Z(X))$                  | $\rightarrow_{\min.\min}$ | $S_f \phi() \pi(n(\overline{N})(n(l.\overline{N})(Y)))$                       |
|  |                           | $\rho(S_f(Z(X)))$   |
| $S_f \phi(\overline{F}) \pi(n(\overline{N})(Y)) \rho(Z(X))$        | $\rightarrow_{\min.\min}$ | $S_g \phi() \sigma(\overline{N}) \tau(\overline{F}) \pi(Y) \rho(Z(X))$        |
| $S_g \phi(\overline{P}) \sigma(l.\overline{N}) \tau(\overline{F})$ | $\rightarrow_{\max.\min}$ | $S_g \phi(\overline{F}.\overline{P}) \sigma(\overline{N}) \tau(\overline{F})$ |
| $S_g \sigma() \tau(\overline{F}) \rho(Z(X))$                       | $\rightarrow_{\min.\min}$ | $Z \rho(X)$   |

- In the particular case  $x = 5$ :  $X = l^5$ ,  $Y = l^{120}$ ,  $y = 120$ .

## 6 Linguistic support

With proper linguistic support, the factorial sample can be rewritten at a more user-friendly high level, where the user needs only develop application's specific

```

1  function main =
2      state  $S_c =$ 
3           $\rightarrow_{\min.\min} x(l^3)$ 
4          set  $y = \text{fact } x$  continue  $S_r$ 
5      state  $S_r =$ 
6          ...
7
8  function fact  $n =$ 
9      state  $S_f =$ 
10          $n() \rightarrow_{\min.\min} S_h f(l)$  % explicit target,  $S_h$ 
11          $\rightarrow_{\min.\min} n_1(\bar{N}) \mid n(l.\bar{N})$  % implicit target,  $S_f$ 
12         set  $f_1 = \text{fact } n_1$  continue  $S_g$ 
13     state  $S_g =$ 
14         set  $f = \text{mult } f_1 \ n$  continue  $S_h$ 
15     state  $S_h =$ 
16         return  $f$ 
17
18  function mult  $a \ b =$ 
19      state  $S_m =$ 
20          $\rightarrow_{\min.\min} c()$ 
21      state  $S_n =$ 
22          $a(l.\bar{X}) \ b(\bar{Y}) \ c(\bar{Z}) \rightarrow_{\max.\min} a(\bar{X}) \ b(\bar{Y}) \ c(\bar{Y}.\bar{Z})$ 
23      return  $c$ 

```

Fig. 2: High-level factorial sample. There are only 5 user defined “business” specific P rules (lines 3, 10, 11, 20, 22); the other P rules are automatically generated.

“business” P rules and the system completes the required boiler-plate templates required by function invocations.

As shown in Figure 2, our proposed high-level language includes the following elements:

- Except hidden system objects, such as a parameter stack and a return stack, *no global* objects should be used (but the system does *not* enforce this recommendation).
- States, parameters and variables are *local* (not visible outside the enclosing function).
- Statement **function** introduces a function, followed by an optional (space separated) list of parameters.
- A function invocation consists of (i) the keyword **set**, (ii) a parameter or variable name (which will receive the result), (iii) the function name, (iv) a (space separated) list of arguments, (v) the keyword **continue**, and (vi) the state to which the function must return.
- Each function argument is either (a) the name of a parameter or variable, or (b) the functor of a complex object.

- Each parameter or variable is implemented by a complex object with the same functor name, which contains its value.
- Complex objects which implement parameters and variables are automatically managed, but are also fully accessible within P system rules.
- We propose that function invocations use call-by-reference evaluations, followed by a copy-on-write, for parameters that are subsequently changed.
- Statement **state** starts a group of rules sharing the same start state (which is now omitted in individual rules). By default, if not explicit, the target state of each rule remains the current state.
- Inside a group, statements and rules are executed top-down, using a weak priority order.
- There is no implicit fall-through from one state to the textually following state.

The traces shown in Table 1 highlight critical steps which occur in the invocation of (fact 3).

We are still considering a default convention for the implicit return-to state after a function invocation, and, more important, extensions for parallel function invocations (which need parallel stacks). However, we are not further developing these ideas here.

## 7 Applications

### 7.1 Numerical P systems

Consider first the numerical P system sample  $\Pi_1$ , given in Păun [17], which sequentially generates numbers in  $\{n^2 | n \geq 0\}$ .  $\Pi_1$  is equivalent to a P module,  $\Pi'_1$ , with one single cell and a single generic rule involving three complex objects,  $a$ ,  $b$ ,  $c$ :

$$S_1 a(\overline{X}) b(\overline{Y}) c(\overline{Z}) \rightarrow_{\min.\min} S_1 a(\overline{XY\overline{Y}l}) b(\overline{Zl}) c(\overline{Zl})$$

Assuming that initially all three objects are empty,  $a()$   $b()$   $c()$ , after  $n$  steps,  $a$  contains  $a(l^{n^2})$ , which represents the number  $n^2$ .

Considering the arithmetic operations that can be efficiently modelled in P modules, we emit the following conjecture:

*Conjecture 1.* All numerical P systems with arithmetic functions on integers and rational numbers can be simulated in real-time by single cell P modules with complex objects.

Note that, if we are not interested in a faithful simulation, the above system can be straightforwardly implemented by the following single rule, which directly maps the algebraic rule  $(n+1)^2 = n^2 + 2n + 1$ :

$$S_1 a(\overline{X}) b(\overline{Y}) \rightarrow_{\min.\min} S_1 a(\overline{XY\overline{Y}l}) b(\overline{Yl})$$

Table 1: Traces for fact of 3. This table asserts the contents of (i) global hidden stacks and (ii) local parameters and variables, *before* a line starts. The small intervals between lines 14–16 indicate calls to mult, which are not detailed here.

| Frame  | Line | $\rho()$             | $\pi()$                            | $n()$     | $n_1()$   | $f_1()$ | $f()$ |
|--------|------|----------------------|------------------------------------|-----------|-----------|---------|-------|
| fact 3 |      |                      |                                    |           |           |         |       |
| "      | 10   | $S_r$                | $p(l^3)$                           | $l^3$     |           |         |       |
| "      | 11   | $S_r$                | $p(l^3)$                           | $l^3$     |           |         |       |
| "      | 12   | $S_r$                | $p(l^3)$                           | $l^3$     | $l^2$     |         |       |
| fact 2 |      |                      |                                    |           |           |         |       |
| "      | 10   | $S_g(S_r)$           | $p(l^2)(p(l^3))$                   | $l^2$     |           |         |       |
| "      | 11   | $S_g(S_r)$           | $p(l^2)(p(l^3))$                   | $l^2$     |           |         |       |
| "      | 12   | $S_g(S_r)$           | $p(l^2)(p(l^3))$                   | $l^2$     | $l$       |         |       |
| fact 1 |      |                      |                                    |           |           |         |       |
| "      | 10   | $S_g(S_g(S_r))$      | $p(l)(p(l^2)(p(l^3)))$             | $l$       |           |         |       |
| "      | 11   | $S_g(S_g(S_r))$      | $p(l)(p(l^2)(p(l^3)))$             | $l$       |           |         |       |
| "      | 12   | $S_g(S_g(S_r))$      | $p(l)(p(l^2)(p(l^3)))$             | $l$       | $\lambda$ |         |       |
| fact 0 |      |                      |                                    |           |           |         |       |
| "      | 10   | $S_g(S_g(S_g(S_r)))$ | $p(\lambda)(p(l)(p(l^2)(p(l^3))))$ | $\lambda$ |           |         |       |
| "      | 16   | $S_g(S_g(S_g(S_r)))$ | $p(\lambda)(p(l)(p(l^2)(p(l^3))))$ | $\lambda$ |           |         | $l$   |
| fact 1 |      |                      |                                    |           |           |         |       |
| "      | 14   | $S_g(S_g(S_r))$      | $p(l)(p(l^2)(p(l^3)))$             | $l$       |           | $l$     |       |
| "      | 16   | $S_g(S_g(S_r))$      | $p(l)(p(l^2)(p(l^3)))$             | $l$       |           |         | $l$   |
| fact 2 |      |                      |                                    |           |           |         |       |
| "      | 14   | $S_g(S_r)$           | $p(l^2)(p(l^3))$                   | $l^2$     |           | $l$     |       |
| "      | 16   | $S_g(S_r)$           | $p(l^2)(p(l^3))$                   | $l^2$     |           |         | $l^2$ |
| fact 3 |      |                      |                                    |           |           |         |       |
| "      | 14   | $S_r$                | $p(l^3)$                           | $l^3$     |           | $l^2$   |       |
| "      | 16   | $S_r$                | $p(l^3)$                           | $l^3$     |           |         | $l^6$ |

## 7.2 NP-complete problems

With complex objects, we can solve NP-complete problems using a single cell, a fixed-sized alphabet and a fixed-sized set of generic rules.

Consider, for example, the SAT problem; see Nagy [12] for a comprehensive overview of this problem and current state-of-art P solutions.

We start with an example. Consider the following formula, with  $n = 3$  boolean variables:

$$f = (x_1 \vee \bar{x}_2) \wedge (x_1 \vee \bar{x}_3).$$

This formula can be expressed as a complex object, in fact a list of disjunctions, where each item is a list of conjunctions:

$$f = \wedge(\vee(x_1(\neg(x_2))))(\vee(x_1(\neg(x_3)))).$$

As such formulas can quickly become unwieldy, we use a simplified notation for list structures, inspired from list structures in System F based functional

programming languages:

$$a(d)(b(e)(c(f))) = [a(d); b(e); c(f)] = [a(d) : [b(e); c(f)]] = [a(d) : [b(e) : [c(f)]]]$$

With this notations, our formula  $f$  can be represented as:

$$f = \wedge[\vee[x_1; \neg(x_2)]; \vee[x_1; \neg(x_3)]].$$

To map our formula to a fixed vocabulary, we represent  $x_n$  by the complex object  $x(l^n)$ . Finally, our formula  $f$  can be represented as:

$$f = \wedge[\vee[x(l); \neg(x(ll))]; \vee[x(l); \neg(x(lll))]].$$

To check its satisfiability, we use a naive brute force approach: we create  $2^n = 2^3 = 8$  dictionary complex objects, all named  $v$ , corresponding to all possible truth (0/1) assignments of our  $n = 3$  variables:

$$\begin{aligned} &v(m(x(l), 0).m(x(ll), 0).m(x(lll), 0)) \\ &v(m(x(l), 0).m(x(ll), 0).m(x(lll), 1)) \\ &v(m(x(l), 0).m(x(ll), 1).m(x(lll), 0)) \\ &v(m(x(l), 0).m(x(ll), 1).m(x(lll), 1)) \\ &v(m(x(l), 1).m(x(ll), 0).m(x(lll), 0)) \\ &v(m(x(l), 1).m(x(ll), 0).m(x(lll), 1)) \\ &v(m(x(l), 1).m(x(ll), 1).m(x(lll), 0)) \\ &v(m(x(l), 1).m(x(ll), 1).m(x(lll), 1)) \end{aligned}$$

All these dictionaries can be built in parallel by the following rules, starting from an empty dictionary,  $v()$ , and a variable  $n(\overline{N})$ , which indicates the number of boolean variables; if this number is not given, it can be easily computed by scanning the given formula (this step is not detailed here):

$$\begin{aligned} S_1 \ n(\overline{N}) \ v(\overline{M}) &\rightarrow_{\max.\min} S_1 \ n(\overline{N}) \ v(m(x(l\overline{N}), 0).\overline{M}) \ v(m(x(l\overline{N}), 1).\overline{M}) \\ S_1 \ n() \ v(\overline{M}) &\rightarrow_{\max.\min} S_2 \ w(t(1), s(0), v(\overline{M})) \end{aligned}$$

Next, we evaluate the given formula,  $f$ , in parallel over all existing dictionaries,  $v$ , which are now enclosed in larger complex objects,  $w$ . The partial results are stored in variables  $s$ , for the current disjunction, initially  $s(0)$ , and  $t$ , for the whole formula (a conjunction), initially  $t(1)$ . These variables start with default values for their corresponding boolean operations and are updated while the formula is evaluated left-to-right. The evaluation looks at the top variable in the top conjunction and picks its value from the associated dictionary. When the top conjunction becomes empty, the value of  $s$  is and-ed to the value of  $t$ , and then variable  $s$  is reset to 0,  $s(0)$ , to start the next disjunction. When there is no other disjunction, the evaluation has ended and  $t$  contains the correct evaluation value according to the current dictionary.

For clarity, we use the convenience *abbreviations* (abbreviations are *not* objects)  $v_{ijk} = v(m(x(l), i).m(x(ll), j).m(x(lll), k))$ , i.e.  $x_1 = i$ ,  $x_2 = j$ ,  $x_3 = k$ . The



following two derivations illustrate the step-by-step evaluation of our formula,  $f$ , using the dictionaries  $v_{000}$  and  $v_{001}$ :

$$\begin{aligned}
& \wedge[\vee[\mathbf{x}(\mathbf{I}); \neg(x(l))]; \vee[x(l); \neg(x(l))]] w(t(1), s(0), v_{000}) \Rightarrow \\
& \quad \wedge[\vee[\neg(\mathbf{x}(\mathbf{II}))]; \vee[x(l); \neg(x(l))]] w(t(1), s(0), v_{000}) \Rightarrow \\
& \quad \quad \wedge[\vee[]; \vee[x(l); \neg(x(l))]] w(t(1), s(1), v_{000}) \Rightarrow \\
& \quad \quad \quad \wedge[\vee[\mathbf{x}(\mathbf{I}); \neg(x(l))]] w(t(1), s(0), v_{000}) \Rightarrow \\
& \quad \quad \quad \quad \wedge[\vee[\neg(\mathbf{x}(\mathbf{III}))]] w(t(1), s(0), v_{000}) \Rightarrow \\
& \quad \quad \quad \quad \quad \wedge[\vee[]] w(t(1), s(1), v_{000}) \Rightarrow \\
& \quad \quad \quad \quad \quad \quad \wedge[] w(t(1), s(0), v_{000}) \Rightarrow \\
& \quad \quad \quad \quad \quad \quad \quad t(1)
\end{aligned}$$

and

$$\begin{aligned}
& \wedge[\vee[\mathbf{x}(\mathbf{I}); \neg(x(l))]; \vee[x(l); \neg(x(l))]] w(t(1), s(0), v_{001}) \Rightarrow \\
& \quad \wedge[\vee[\neg(\mathbf{x}(\mathbf{II}))]; \vee[x(l); \neg(x(l))]] w(t(1), s(0), v_{001}) \Rightarrow \\
& \quad \quad \wedge[\vee[]; \vee[x(l); \neg(x(l))]] w(t(1), s(1), v_{001}) \Rightarrow \\
& \quad \quad \quad \wedge[\vee[\mathbf{x}(\mathbf{I}); \neg(x(l))]] w(t(1), s(0), v_{001}) \Rightarrow \\
& \quad \quad \quad \quad \wedge[\vee[\neg(\mathbf{x}(\mathbf{III}))]] w(t(1), s(0), v_{001}) \Rightarrow \\
& \quad \quad \quad \quad \quad \wedge[\vee[]] w(t(1), s(0), v_{001}) \Rightarrow \\
& \quad \quad \quad \quad \quad \quad \wedge[] w(t(0), s(0), v_{001}) \Rightarrow \\
& \quad \quad \quad \quad \quad \quad \quad t(0)
\end{aligned}$$

Without using usual boolean shortcuts, these evaluations can be completed by following rules (using the simplified list notation, for clarity):

$$\begin{aligned}
& S_2 \wedge [\vee[\neg(\mathbf{x}(\overline{\mathbf{X}})) : D] : C] w(t(T), s(S), v(\overline{P}.m(\mathbf{x}(\overline{\mathbf{X}}), V).\overline{Q})) \\
& \quad \rightarrow_{\max.\min} S_2 \wedge [\vee[D] : C] w(t(T), s(Z), v(\overline{P}.m(\mathbf{x}(\overline{\mathbf{X}}), V).\overline{Q})) \mid e(S, V, Z) \\
& S_2 \wedge [\vee[\mathbf{x}(\overline{\mathbf{X}}) : D] : C] w(t(T), s(S), v(\overline{P}.m(\mathbf{x}(\overline{\mathbf{X}}), V).\overline{Q})) \\
& \quad \rightarrow_{\max.\min} S_2 \wedge [\vee[D] : C] w(t(T), s(Z), v(\overline{P}.m(\mathbf{x}(\overline{\mathbf{X}}), V).\overline{Q})) \mid d(S, V, Z) \\
& S_2 \wedge [\vee[] : C] w(t(T), s(S), v(\overline{M})) \\
& \quad \rightarrow_{\max.\min} S_2 \wedge [C] w(t(Z), s(0), v(\overline{M})) \mid c(T, S, Z) \\
& S_2 \wedge [] w(t(T), s(S), v(\overline{M})) \\
& \quad \rightarrow_{\max.\min} S_3 t(T)
\end{aligned}$$

where  $c()$ ,  $d()$  and  $e$  are “read-only” internal tables for required boolean operations, given as complex objects (intuitively, these represent tables for  $x \wedge y$ ,  $x \vee y$ ,  $x \vee \bar{y}$ , respectively):

$$\begin{aligned}
& c(0, 0, 0) \ c(0, 1, 0) \ c(1, 0, 0) \ c(1, 1, 1) \\
& d(0, 0, 0) \ d(0, 1, 1) \ d(1, 0, 1) \ d(1, 1, 1) \\
& e(0, 0, 1) \ e(0, 1, 0) \ e(1, 0, 1) \ e(1, 1, 1)
\end{aligned}$$

The final rules collect and reduce the individual results,  $t()$ . In our case, the formula  $f$  is satisfiable, for example for  $x_1 = 0$ ,  $x_2 = 0$ ,  $x_3 = 0$ .

We have used a single cell, a fixed alphabet,  $\{0, 1, x, l, \vee, \wedge, \neg, c, d, e, v, m, w, s, t\}$ , and essentially just 6 generic rules and 3 states. In this example,

we used only the most basic brute force approach; however, better variants are possible.

A similar approach seems to work well for other NP-complete problems, for example, the graph colouring problem; see Gheorghe et al. [7] for state-of-art P solutions of this problem. We emit the following conjecture:

*Conjecture 2.* Any NP-complete problems can be solved by a single cell P module with a fixed sized atomic alphabet and a fixed sized set of generic rules.

## 8 Conclusions

Despite their exceptional theoretical and modelling power, P systems seem to remain difficult to use for large practical applications, apparently requiring large varying size unstructured rulesets that can be difficult to verify. We want to show that this need not be the case, that there are ways to increase their usability.

This paper presents evidence that complex objects can enable a high-level programming style, with fixed sized alphabets and rulesets, adequate data structures and useful functional programming elements. We have previously used complex objects to successfully model and even improve large practical applications, ranging from computer vision to complex graph theoretical problems and to well-known critical distributed algorithms. Here we attempt to generalise our field-proven methods and sketch how to apply similar techniques to other, more theoretical, domains: numerical P systems and NP-complete problems.

The presented evidence suggests that complex objects could enable a more advanced high-level functional programming style, including: local functions (functions inside functions), closures, memoizations (i.e. top-down dynamic programming), combinators (e.g. the Y combinator), monads and meta-programming. A follow-up paper will address these topics.

Many of our extensions can be directly mapped on modern computing platforms, bypassing a possible translation to traditional simpler objects and rules, which opens the way towards more efficient general purpose simulators.

**Acknowledgments.** The work of RN and FI was partially supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-II-ID-PCE-2011-3-0688. We are indebted to the anonymous reviewers for their valuable comments and suggestions.

## References

1. Alhazov, A., Ivanov, S., Rogozhin, Y.: Polymorphic P systems. In: Gheorghe, M., Hinze, T., Păun, G., Rozenberg, G., Salomaa, A. (eds.) *Membrane Computing*, Lecture Notes in Computer Science, vol. 6501, pp. 81–94. Springer Berlin Heidelberg (2011)
2. Artiom Alhazov, Cosmin Bonchis, G.C., Izbasa, G.: Encodings and arithmetic operations in P systems. In: Gutierrez-Naranjo, M., Păun, G., Riscos-Nunez, A., Romero-Campero, F.J. (eds.) *Brainstorming Week on Membrane Computing*, vol. 2, pp. 1–28. Universidad de Sevilla (2006)

3. Bălănescu, T., Nicolescu, R., Wu, H.: Asynchronous P systems. *International Journal of Natural Computing Research* 2(2), 1–18 (2011)
4. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: A faster P solution for the Byzantine agreement problem. In: Gheorghe, M., Hinze, T., Păun, G. (eds.) *Conference on Membrane Computing. Lecture Notes in Computer Science*, vol. 6501, pp. 175–197. Springer-Verlag, Berlin Heidelberg (2010)
5. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: P systems and the Byzantine agreement. *Journal of Logic and Algebraic Programming* 79(6), 334–349 (2010)
6. ElGindy, H., Nicolescu, R., Wu, H.: Fast distributed DFS solutions for edge-disjoint paths in digraphs. In: Csuhaj-Varj, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) *Membrane Computing, Lecture Notes in Computer Science*, vol. 7762, pp. 173–194. Springer Berlin Heidelberg (2013)
7. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M., Turcanu, A., Valencia Cabrera, L., Garccia-Quismondo, M., Mierla, L.: 3-col problem modelling using simple kernel P systems. *Int. J. Comput. Math.* 90(4), 816–830 (Apr 2013)
8. Gimelfarb, G., Gong, R., Nicolescu, R., Delmas, P.: Concurrent propagation for solving ill-posed problems of global discrete optimisation. In: *Pattern Recognition (ICPR), 2012 21st International Conference on*. pp. 1864–1867 (2012)
9. Gimelfarb, G., Nicolescu, R., Ragavan, S.: P systems in stereo matching. In: Real, P., Diaz-Pernil, D., Molina-Abril, H., Berciano, A., Kropatsch, W. (eds.) *Computer Analysis of Images and Patterns, Lecture Notes in Computer Science*, vol. 6855, pp. 285–292. Springer Berlin Heidelberg (2011)
10. Gimelfarb, G., Nicolescu, R., Ragavan, S.: P system implementation of dynamic programming stereo. *Journal of Mathematical Imaging and Vision* pp. 1–14 (2012)
11. Manca, V., Lombardo, R.: Computing with multi-membranes. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) *Membrane Computing, Lecture Notes in Computer Science*, vol. 7184, pp. 282–299. Springer Berlin Heidelberg (2012)
12. Nagy, B.: On efficient algorithms for SAT. In: Csuhaj-Varj, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) *Membrane Computing, Lecture Notes in Computer Science*, vol. 7762, pp. 295–310. Springer Berlin Heidelberg (2013)
13. Nicolescu, R.: Parallel and distributed algorithms in P systems. In: Gheorghe, M., Păun, G., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) *Membrane Computing, CMC 2011, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 7184, pp. 35–50. Springer Berlin / Heidelberg (2012)
14. Nicolescu, R., Wu, H.: BFS solution for disjoint paths in P systems. In: Calude, C., Kari, J., Petre, I., Rozenberg, G. (eds.) *Unconventional Computation, Lecture Notes in Computer Science*, vol. 6714, pp. 164–176. Springer Berlin Heidelberg (2011)
15. Nicolescu, R., Wu, H.: New solutions for disjoint paths in P systems. *Natural Computing* 11, 637–651 (2012)
16. Păun, G.: Computing with membranes. *Journal of Computer and System Sciences* 61(1), 108–143 (2000)
17. Păun, G., Păun, R.: Membrane computing and economics: Numerical P systems. *Fundam. Inf.* 73(1,2), 213–227 (Jul 2006)
18. Păun, G., Rozenberg, G., Salomaa, A.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA (2010)
19. Wu, H.: Minimum spanning tree in P systems. In: *Proceedings of the Asian Conference on Membrane Computing (ACMC2012), Wuhan, China, October 15-18, 2012*. pp. 88–104 (2012)



# In Search of a Structure of Fractals by using Membranes as Hyperedges

Adam Obtułowicz

Institute of Mathematics, Polish Academy of Sciences  
Śniadeckich 8, P.O.B. 21, 00-956 Warsaw, Poland  
e-mail: A.Obtulowicz@impan.pl

**Abstract.** The internal structure of the iterations of Koch curve and Sierpiński gasket—the known fractals [4]—is described in terms of multi-hypergraphical membrane systems related to membrane structures [13] and whose membranes are hyperedges of multi-hypergraphs used to define gluing patterns for the components of the iterations of the considered fractals.

## 1 Introduction

One finds in [10] a more or less explicit conclusion that the birth of functional analysis was accompanied by the emergence of various mathematical structures (from vector space, abstract metric spaces and topological spaces to Hilbert spaces, including spaces of functions) which were an antidotum against ‘capricious’ intuitiveness of symbolic ‘calculations’ of early calculus.

This conclusion inspired the author of the present paper to search for structures of fractals and self-similarity against their intuitive explanations<sup>1</sup> proposed e.g. in [9]:

*‘Local’ statements of self-similarity say something like ‘almost any small pattern observed in one part of the object can be observed throughout the object, at all scales’. Global statements say something like ‘the whole object consists of several smaller copies of itself glued together’; more generally, there may be a whole family of objects, each of which can be described as several objects in the family glued together.*

*Viewed from another angle, a theory of global self-similarity is a theory of recursive decomposition.*

One should point out here that in a large extent the concepts of fractals and self-similarity have been already described precisely in terms of iterated function systems with their attractors constructed by using the tools of functional analysis (Hahn–Banach fix point theorem) [4] and domain theory (Tarski fix point theorem) [3]. But a translation from the language of the above intuitive explanation to the language of some derived concepts from the precise description of

---

<sup>1</sup> the explanations suggested by the visual presentations of the iterations of some fractals seen in the books and many articles about fractals.

fractals and self-similarity (e.g. the trees induced by iterated function systems, cf. [3]) is not effortless and not yet ready.

Thus searching for structure of fractals and self-similarity is approached by various mathematicians, cf. [7], [9], not necessarily motivated explicitly by a need of the above translation.

The goal of the paper is to propose an approach to searching for structure of fractals which could provide the above translation. We describe in Section 3 the internal structure of the iterations of Koch curve and Sierpiński gasket—the known fractals [4]—in terms of multi-hypergraphical membrane systems related to membrane structures [13] and whose membranes are hyperedges of multi-hypergraphs used to define gluing patterns for the components of the iterations of the considered fractals.

## 2 Multi-hypergraphical membrane systems

We introduce the following new concepts.

By a *directed multi-hypergraph* we mean a structure  $\mathcal{G}$  given by its set  $E(\mathcal{G})$  of *hyperedges*, its set  $V(\mathcal{G})$  of *vertices* and the *source* and *target* mappings

$$s_{\mathcal{G}} : E(\mathcal{G}) \rightarrow \mathcal{P}(V(\mathcal{G})), \quad t_{\mathcal{G}} : E(\mathcal{G}) \rightarrow \mathcal{P}(V(\mathcal{G}))$$

such that  $V(\mathcal{G})$  together with

$$\{(\mathcal{V}_1, \mathcal{V}_2) \mid s_{\mathcal{G}}(e) = \mathcal{V}_1 \text{ and } t_{\mathcal{G}}(e) = \mathcal{V}_2 \text{ for some } e \in E(\mathcal{G})\}$$

form a directed hypergraph as in [5], where  $\mathcal{P}(X)$  denotes the set of all subsets of a set  $X$ .

We say that two directed multi-hypergraphs  $\mathcal{G}, \mathcal{G}'$  are *isomorphic* if there exist two bijections  $h : V(\mathcal{G}) \rightarrow V(\mathcal{G}')$ ,  $h' : E(\mathcal{G}) \rightarrow E(\mathcal{G}')$  such that

$$s_{\mathcal{G}'}(h'(e)) = \{h(v) \mid v \in s_{\mathcal{G}}(e)\} \text{ and } t_{\mathcal{G}'}(h'(e)) = \{h(v) \mid v \in t_{\mathcal{G}}(e)\}$$

for all  $e \in E(\mathcal{G})$ .

Membrane structures in [13] are simply finite trees with nodes labelled by multisets, where the finite trees have a natural visual presentation by Venn diagrams and the tree nodes are called *membranes*.

We introduce (*directed*) *multi-hypergraphical membrane systems* to be finite trees with nodes labelled by (directed) multi-hypergraphs.

We consider directed multi-hypergraphical membrane systems of a special feature described formally in the following way.

A *multi-hyperedge membrane system*  $\mathcal{S}$  is given by:

- the *underlying tree*  $\mathbb{T}_{\mathcal{S}}$  of  $\mathcal{S}$  which is a finite graph given by its set  $V(\mathbb{T}_{\mathcal{S}})$  of *vertices*, its set  $E(\mathbb{T}_{\mathcal{S}}) \subseteq V(\mathbb{T}_{\mathcal{S}}) \times V(\mathbb{T}_{\mathcal{S}})$  of *edges*, and its *root*  $r$  which is a distinguished vertex such that for every vertex  $v$  different from  $r$  there exists a unique path from  $v$  into  $r$  in  $\mathbb{T}_{\mathcal{S}}$ , where for every vertex  $v$  we define  $\text{rel}(v) = \{v' \mid (v', v) \in E(\mathbb{T}_{\mathcal{S}})\}$  and in trivial case  $V(\mathbb{T}_{\mathcal{S}}) = \{r\}$  we assume  $E(\mathbb{T}_{\mathcal{S}}) = \emptyset$ ;

- a family  $(G_v | v \in V(\mathbb{T}_S))$  of finite directed multi-hypergraphs for  $G_v$  given by its set  $V(G_v)$  of *vertices*, its set  $E(G_v)$  of *edges*, its *source function*  $s_v : E(G_v) \rightarrow \mathcal{P}(V(G_v))$ , and its *target function*  $t_v : E(G_v) \rightarrow \mathcal{P}(V(G_v))$  such that the following conditions hold:
  - 1)  $E(G_v) = \text{rel}(v)$ ,
  - 2)  $V(G_v)$  is empty for every *elementary* vertex  $v$ , i.e. such that  $\text{rel}(v)$  is empty.

The above multi-hypergraphical membrane systems can be drawn by using Venn diagrams with discs or boxes  $d_v$  corresponding to vertices  $v$  of  $\mathbb{T}_S$ .

One can expect the applications of multi-hypergraphical membrane systems for modelling various hierarchically organized systems of nested modules (hyperedges) interconnected by many input and output lines (vertices), where the module interactions are described by source and target functions. These systems of modules appear in computer science, where the modules are complex actions, instructions, transitions (e.g. of structured Petri nets [2]), etc., from state charts [6], models of systemC components [17], the systems discussed in [1], to the semantics of some extensions of formal systems in [12], [17], and hierarchical specifications [15].

### 3 Koch curve and Sierpiński gasket

We describe in this section the iterations of Koch curve and Sierpiński gasket [8], [4], [14] in terms of multi-hypergraphical membrane systems.

For natural numbers  $n > 0$  and  $i \in \{\text{Koch}, \text{Sierp}\}$  we define multi-hyperedge membrane systems  $\mathcal{S}_n^i$  in the following way:

- the underlying tree  $\mathbb{T}_n^i$  of  $\mathcal{S}_n^i$  is such that
  - the set  $V(\mathbb{T}_n^i)$  of vertices is the set of all strings (sequences) of length not greater than  $n$  of digits in  $D^{\text{Sierp}} = \{1, 2, 3\}$  for  $i = \text{Sierp}$ , and in  $D^{\text{Koch}} = \{1, 2, 3, 4\}$  for  $i = \text{Koch}$ ,
  - the set  $E(\mathbb{T}_n^i)$  of edges of  $\mathbb{T}_n^i$  is such that  $E(\mathbb{T}_n^i) = \{(\Gamma j, \Gamma) | \{\Gamma j, \Gamma\} \subset V(\mathbb{T}_n^i) \text{ and } j \in D^i\}$  with source and target functions being the projections on the first and the second component, respectively, where  $\Gamma j$  is the string obtained by juxtaposition a new digit  $j$  on the right end of  $\Gamma$ ,
- the family  $(G_\Gamma^i | \Gamma \in V(\mathbb{T}_n^i))$  of directed multi-hypergraphs of  $\mathcal{S}_n^i$  is such that for every non-elementary vertex  $\Gamma \in V(\mathbb{T}_n^i)$ , i.e. with  $\text{rel}(\Gamma) \neq \emptyset$ ,  $G_\Gamma^i$  is determined in the following way:
  - for  $i = \text{Koch}$  if  $\Gamma$  is the empty string, then the directed multi-hypergraph  $G_\Gamma^i$  is such that  $V(G_\Gamma^i)$  is a five element set  $\{v_0, \dots, v_4\}$ ,  $E(G_\Gamma^i) = \{\Gamma j | j \in D^i\}$ , and the source and target functions of  $G_\Gamma^i$  are given by

$$s_{G_\Gamma^i}(\Gamma j) = \{v_{j-1}\}, \quad t_{G_\Gamma^i}(\Gamma j) = \{v_j\} \text{ for all } j \in \{1, \dots, 4\},$$

where

$$v_0 = (0, 0), \quad v_1 = (\frac{1}{3}, 0), \quad v_2 = (\frac{1}{2}, \frac{2}{2\sqrt{3}}), \quad v_3 = (\frac{2}{3}, 0), \quad v_4 = (1, 0),$$

- for  $i = \text{Sierp}$  if  $\Gamma$  is the empty string, then the directed multi-hypergraph  $G_\Gamma^i$  is such that  $V(G_\Gamma^i)$  is a six element set  $\{v_0, \dots, v_5\}$ ,  $E(G_\Gamma^i) = \{\Gamma j \mid j \in D^i\}$ , and the source and target functions of  $G_\Gamma^i$  are given by

$$\begin{aligned} s_{G_\Gamma^i}(\Gamma 3) &= \{v_1, v_2\}, & t_{G_\Gamma^i}(\Gamma 3) &= \{v_0\}, \\ s_{G_\Gamma^i}(\Gamma j) &= \{v_{j+2}, v_{j+3}\}, & t_{G_\Gamma^i}(\Gamma j) &= \{v_j\} \text{ for } j \in \{1, 2\}, \end{aligned}$$

where

$$\begin{aligned} v_0 &= (\tfrac{1}{2}, \tfrac{\sqrt{3}}{2}), \\ v_1 &= (\tfrac{1}{4}, \tfrac{\sqrt{3}}{4}), & v_2 &= (\tfrac{3}{4}, \tfrac{\sqrt{3}}{4}), \\ v_3 &= (0, 0), & v_4 &= (\tfrac{1}{2}, 0), & v_5 &= (1, 0), \end{aligned}$$

- if a non-elementary vertex  $\Gamma$  of  $\mathbb{T}_n^i$  is of the form<sup>2</sup>  $k\Omega$  for  $k \in D^i$  and a string  $\Omega$  of digits in  $D^i$ , then

$$V(G_\Gamma^i) = \{f_k^i(v) \mid v \in V(G_\Omega^i)\}, \quad E(G_\Gamma^i) = \{\Gamma j \mid j \in D^i\},$$

and

$$\delta_{G_\Gamma^i}(\Gamma j) = \{f_k^i(v) \mid v \in \delta_{G_\Omega^i}(\Omega j)\} \quad \text{for all } j \in D^i \text{ and } \delta \in \{s, t\}$$

where  $f_k^i$  is the  $k$ -th function of the iterated function system given in [14] for Koch curve in the case  $i = \text{Koch}$  and for Sierpiński gasket in the case  $i = \text{Sierp}$ , respectively.

**Lemma.** *For all natural numbers  $n > 0$  and  $i \in \{\text{Koch}, \text{Sierp}\}$  the multi-hyperedge membrane system  $\mathcal{S}_n^i$  is such that for every non-elementary vertex  $\Gamma$  of  $\mathbb{T}_n^i$  the directed multi-hypergraph  $G_\Gamma^i$  is isomorphic to  $G_\Lambda^i$  for empty string  $\Lambda$ —the root of  $\mathbb{T}_n^i$ .*

*Proof.* We prove the lemma by induction on  $n$  and by using the property of the functions of the iterated function systems for Koch curve and Sierpiński gasket that they are injections.

For all natural numbers  $n > 0$  and  $i \in \{\text{Koch}, \text{Sierp}\}$  we define a *geometrical realization* of  $\mathcal{S}_n^i$ , denoted by  $\text{space}(\mathcal{S}_n^i)$ , to be a subset of  $\mathbb{R}^2$  ( $\mathbb{R}^2$  is a Cartesian product of two copies of the set  $\mathbb{R}$  of real numbers) which is the  $n$ -th iteration of Koch curve for  $i = \text{Koch}$  and the  $n$ -th iteration of Sierpiński gasket for  $i = \text{Sierp}$ , i.e.

$$\begin{aligned} \text{space}(\mathcal{S}_1^{\text{Koch}}) &= \bigcup_{j \in D^{\text{Koch}}} f_j^{\text{Koch}}(\text{interval}), \\ \text{space}(\mathcal{S}_1^{\text{Sierp}}) &= \bigcup_{j \in D^{\text{Sierp}}} f_j^{\text{Sierp}}(\text{equitriang}), \\ \text{space}(\mathcal{S}_{n+1}^i) &= \bigcup_{j \in D^i} f_j^i(\text{space}(\mathcal{S}_n^i)) \text{ for } i \in \{\text{Koch}, \text{Sierp}\} \end{aligned}$$

<sup>2</sup> the form  $k\Omega$  of  $\Gamma$  is understood that the first element of  $\Gamma$  is  $k$  followed by the string  $\Omega$ .



where  $f_j^i(X)$  is the image of a set  $X$  for  $f_j^i$ ,  $\text{interval} = \{(t, 0) \mid t \in \mathbb{R}, 0 \leq t \leq 1\}$ , and equitriang is the union of the interior and the frontier of the equilateral triangle in  $\mathbb{R}^2$  whose vertices are  $(0, 0)$ ,  $(\frac{1}{2}, \frac{\sqrt{3}}{2})$ ,  $(1, 0)$ .

**Theorem.** *For all natural numbers  $n > 0$  and  $i \in \{\text{Koch}, \text{Sierp}\}$  the set space  $(\mathcal{S}_n^i)$  is not an amorphous set of points of  $\mathbb{R}^2$  but it is a structured set by its hierarchically organized decomposition into subsets according to the underlying tree  $\mathbb{T}_n^i$  of  $\mathcal{S}_n^i$ , where the components of the decomposition form a family  $C_\Gamma^{i,n}$  ( $\Gamma \in V(\mathbb{T}_n^i)$ ,  $\Gamma$  is non-empty and is not an elementary vertex of  $\mathbb{T}_n^i$ ) such that:*

- if  $\Gamma$  is of the form  $j\Omega$  for  $j \in D^i$  and a string  $\Omega$  of digits in  $D^i$ , then
  - for the empty string  $\Omega$  the component  $C_{j\Omega}^{i,n}$  is  $f_j^i(\text{space}(\mathcal{S}_{n-1}^i))$ ,
  - for a non-empty string  $\Omega$  the component  $C_{j\Omega}^{i,n}$  is  $f_j^i(C_\Omega^{i,n-1})$  for the  $\Omega$ -th component  $C_\Omega^{i,n-1}$  of  $\text{space}(\mathcal{S}_{n-1}^i)$ ,
- for  $m_i = \max D^i$  the  $m_i$  components  $C_{\Gamma_1}^{i,n}, \dots, C_{\Gamma_{m_i}}^{i,n}$  are glued according to the pattern given by  $G_\Gamma^i$  understood that

$$\delta(\Gamma j') \cap \gamma(\Gamma j'') = C_{\Gamma j'}^{i,n} \cap C_{\Gamma j''}^{i,n}$$

for all  $\delta, \gamma, j', j''$  with  $\{\delta, \gamma\} \subseteq \{s_{G_\Gamma^i}^i, t_{G_\Gamma^i}^i\}$ ,  $\{j', j''\} \subseteq D^i$ , and  $j' \neq j''$ .

*Proof.* The theorem is an immediate consequence of the adopted definitions.

The above multi-hypergraphical membrane systems can be drawn by using Venn diagrams with discs or boxes  $d_\Gamma$  corresponding to vertices  $\Gamma$  of  $\mathbb{T}_n^i$  such that  $d_{\Gamma_j}$  is an immediate subset of  $d_\Gamma$ .

## Conclusion

The above lemma and theorem provide the translation claimed in the introduction of the paper for iterations of fractals in the cases of Koch curve and Sierpiński gasket. In this translation the main feature of self-similarity described in its ‘local’ statement corresponds to the isomorphisms of hypergraphs ‘giving’ the gluing patterns (see the above theorem) for every level of hierarchical organization of the decomposition, where the levels of hierarchical organization coincide with scale layers.

The iterations of  $jD$ -Cantor set ( $j \in \{1, 2, 3\}$ ) require another approach which is proposed in [11], where multigraphical membrane systems are used with vertices as membranes. Thus one may say that the approach proposed in the present paper is a ‘hyperedges as membranes’ approach.

## References

1. Bruni, R., Gadduci, F., Lluch Lafuente, A., *An algebra of hierarchical graphs*, in: Trustworthy Global Computing, Lecture Notes in Comput. Sci. 6084, Springer, Berlin, 2010, pp. 205–211.

2. Cherkasova, L. A., Kotov, V. E., *Structured nets*, in: Mathematical Foundations of Computer Science, Lecture Notes in Comput. Sci. 118, Springer, Berlin, 1981, pp. 242–251.
3. Edalat, A., *Domains for computation in mathematics, physics and exact real arithmetic*, The Bulletin of Symbolic Logic 3 (1997), pp. 401–452.
4. Falconer, K., *Fractal Geometry. Mathematical Foundations and Applications*, Wiley, Hoboken, NJ, 2003.
5. Gallo, G., Longo, G., Pallottino, S., Nguyen, S., *Directed hypergraphs and applications*, Discrete Appl. Math. 42 (1993), pp. 177–201.
6. Harel, D., *On Visual Formalisms*, Comm. ACM 31 (1988), pp. 514–530.
7. Hasuo, I., Jacobs, B., Niqui, M., *Coalgebraic representation theory of fractals*, Electron. Notes Theor. Comput. Sci. 265 (2010), pp. 351–368.
8. Hutchinson, J. E., *Fractals and self-similarity*, Indiana Univ. Math. J. 30 (1981), pp. 713–747.
9. Leinster, T., *A general theory of self-similarity*, Adv. Math. 226 (2011), pp. 2935–3017.
10. Narici, L., Beckenstein, E., *The Hahn–Banach theorem: the life and times*, Topology Appl. 77 (1997), pp. 193–211.
11. Obtułowicz, A., *Multigraphical membrane systems revisited*, in: Membrane Computing, Lecture Notes in Comput. Sci. 7762, Springer, Berlin, 2013, pp. 311–322.
12. Orlarey, Y., Fober, D., Letz, S., Bilton, M., *Lambda calculus and music calculi*, International Computer Music Conference ICMA 1994.
13. Păun, Gh., *Membrane Computing. An Introduction*, Springer-Verlag, Berlin 2002.
14. Riddle, L., *Classic iterated function systems, Koch curve, Sierpiński gasket*,  
<http://ecademy.agnesscott.edu/~lriddle/ifs/kcurve/kcurve.htm>  
<http://ecademy.agnesscott.edu/~lriddle/ifs/siertri/siertri.htm>
15. Rozenkrantz, D. J., Hunt III, H. B., *The complexity of processing hierarchical specifications*, SIAM J. Comput. 22 (1993), pp. 627–649.
16. Stefanescu, Gh., *The algebra of flownomials*, Report, Technical University Munich, 1994.
17. Vallée, N., Monsuez, B., *A formal model of system components using fractal hypergraphs*, in: Proc. of the Int. Multiconference of Engineers and Computer Scientists 2010, Vol. II, IMECS 2010, Hong Kong.

# The Relevance of the Environment on the Efficiency of Tissue P Systems

M.J. Pérez-Jiménez<sup>1</sup>, A. Riscos-Núñez<sup>1</sup>, M. Rius-Font<sup>2</sup>, L. Valencia-Cabrera<sup>1</sup>

<sup>1</sup> Research Group on Natural Computing  
Department of Computer Science and Artificial Intelligence  
University of Sevilla

Avda. Reina Mercedes s/n, 41012 Sevilla, Spain  
E-mail: { marper, ariscosn, lvalencia } @us.es

<sup>2</sup> Department of Applied Mathematics IV  
Universitat Politècnica de Catalunya, Spain  
E-mail: mrius@ma4.upc.edu

**Abstract.** The efficiency of computational devices is usually expressed in terms of their capability to solve computationally hard problems in polynomial time. This paper focuses on tissue P systems, whose efficiency has been shown for several scenarios where the number of cells in the system can grow exponentially, e.g. by using cell division rules or cell separation rules. Moreover, in the first case it suffices to consider very short communication rules with length bounded by two, and in the second one it is enough to consider communication rules with length at most three. This kind of systems have an environment with the property that objects initially located in it appear in an arbitrarily large number of copies, which is a somewhat unfair condition from a computational complexity point of view. In this context, we study the role played by the environment and its ability to handle infinitely many objects, in particular we consider tissue P systems whose environment is initially empty.

## 1 Introduction

Several different models of cell-like P systems have been successfully used to efficiently solve computationally hard problems by trading space for time. An exponential workspace is created in polynomial time by using some kind of rules, and then massive parallelism is used to simultaneously check all the candidate solutions. Inspired by living cells, several ways for obtaining exponential workspace in polynomial time were proposed: membrane division (*mitosis*) [12], membrane creation (*autopoiesis*) [5], and membrane separation (*membrane fission*) [8]<sup>3</sup>. These three ways have given rise to the following models: *P systems with active membranes*, *P systems with membrane creation*, and *P systems with membrane separation*, respectively.

---

<sup>3</sup> The name *separation rule* appeared earlier in [1], but with a slightly different definition.

A new type of P systems, the so-called *tissue P systems*, was introduced in [7]. The hierarchical membrane structure that was commonly used in the first models, inspired on the way vesicles and compartments are arranged within a cell, is discarded. Instead, an arbitrary graph of connections among elementary membranes (now called *cells*) is considered. That is, the inspiration comes now not from a single cell but from a collection of cooperating cells within a multicellular organism, e.g. in a tissue. Moreover, the functioning of tissue P systems heavily relies on the intercellular communication, since objects can move under symport/antiport rules, but cannot be rewritten.

This paper addresses two models of tissue P systems which are of a great interest from a computational complexity point of view. The first one was presented in [14], where the definition of tissue P systems is combined with aspects of the definition of P systems with active membranes, yielding *tissue P systems with cell division*. In these models, cells may replicate, that is, the two new cells generated by a division rule have exactly the same objects except for at most one differing pair of objects. The second model that will be considered is *tissue P systems with cell separation* [9]. In this case, an alternative method for generating an exponential number of cells in linear time is used. When a cell divides, its contents are not replicated, but distributed, according to a fixed partition of the alphabet.

The paper is organized as follows. First, we recall the basic mathematical and theoretical background underlying the definitions of the two tissue P systems models mentioned above, together with the definition of complexity class in the membrane computing framework. Then, Section 3 compares the computational power achieved by cell division and by cell separation, evaluating in both cases the role of the environment. Some concluding remarks summarizing the borderlines of efficiency discussed in the paper are given in Section 4.

## 2 Tissue P Systems

Let us recall that an *alphabet*  $\Gamma$  is a non-empty set whose elements are called *symbols*. A *multiset*  $m$  over an alphabet  $\Gamma$  is a pair  $m = (\Gamma, f)$  where  $f : \Gamma \rightarrow \mathbb{N}$  is a mapping. If  $m = (\Gamma, f)$  is a multiset then its *support* is defined as  $\text{supp}(m) = \{x \in \Gamma \mid f(x) > 0\}$ . A multiset is finite if its support is a finite set. Let  $\text{supp}(m) = \{a_1, \dots, a_k\}$  be the support of a finite multiset,  $m$ , then we will denote  $m = a_1^{f(a_1)} \dots a_k^{f(a_k)}$  (here the order is irrelevant), and we say that  $f(a_1) + \dots + f(a_k)$  is the cardinal of  $m$ , denoted by  $|m|$ . The empty multiset is denoted by  $\lambda$ . We also denote by  $M_f(\Gamma)$  the set of all finite multisets over  $\Gamma$ .

Let  $m_1 = (\Gamma, f_1)$  and  $m_2 = (\Gamma, f_2)$  multisets over  $\Gamma$ . The *union* of  $m_1$  and  $m_2$ , denoted by  $m_1 + m_2$  is the multiset  $(\Gamma, g)$ , where  $g = f_1 + f_2$ , that is,  $g(x) = f_1(x) + f_2(x)$  for each  $x \in \Gamma$ . The *relative complement* of  $m_2$  in  $m_1$ , denoted by  $m_1 \setminus m_2$  is the multiset  $(\Gamma, g)$ , where  $g(x) = f_1(x) - f_2(x)$  if  $f_1(x) \geq f_2(x)$  and  $g(x) = 0$  otherwise.

**Definition 1.** A basic tissue P system of degree  $q \geq 1$  is a tuple  $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$ , where:

1.  $\Gamma$  is a finite alphabet and  $\mathcal{E}$  is a subset of  $\Gamma$ .
2.  $\Sigma$  is an (input) alphabet strictly contained in  $\Gamma$  such that  $\mathcal{E} \cap \Sigma = \emptyset$ .
3.  $\mathcal{M}_1, \dots, \mathcal{M}_q$  are finite multisets over  $\Gamma \setminus \Sigma$ .
4.  $\mathcal{R}$  is a finite set of communication rules of the form  $(i, u/v, j)$ , for  $i, j \in \{0, 1, 2, \dots, q\}$ ,  $i \neq j$ ,  $u, v \in M_f(\Gamma)$ , and  $|u + v| \neq 0$ ;
5.  $i_{in} \in \{1, 2, \dots, q\}$ , and  $i_{out} \in \{0, 1, \dots, q\}$ .

A basic tissue P system  $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$  of degree  $q \geq 1$  can be viewed as a set of  $q$  cells, labelled by  $1, \dots, q$ , with an environment labelled by 0 such that: (a)  $\mathcal{M}_1, \dots, \mathcal{M}_q$  are finite multisets over  $\Gamma$  representing the objects (elements in  $\Gamma$ ) initially placed in the  $q$  cells of the system; (b)  $\Sigma$  is the input alphabet and  $\mathcal{E}$  is the set of objects located initially in the environment of the system, all of them appearing in an arbitrary number of copies; and (c)  $i_{in}$  represents the input cell, and  $i_{out} \in \{0, 1, \dots, q\}$  indicates the region that stores the output of the system (which can be either a distinguished cell when  $i_{out} \in \{1, \dots, q\}$ , or the environment when  $i_{out} = 0$ ). If  $\mathcal{E} = \emptyset$  then we say that the tissue P system is *without environment*.

A communication rule  $(i, u/v, j)$  is *applicable* to regions  $i, j$  if the multiset  $u$  is contained in region  $i$  and multiset  $v$  is contained in region  $j$ . When applying a communication rule  $(i, u/v, j)$ , the objects of multiset  $u$  are sent from region  $i$  to region  $j$  and, simultaneously, the objects of multiset  $v$  are sent from region  $j$  to region  $i$ . The *length* of communication rule  $(i, u/v, j)$  is defined as  $|u| + |v|$ .

The rules are used in a non-deterministic maximally parallel manner as customary in membrane computing. At each step, we apply a multiset of rules which is *maximal*: no further applicable rule can be added.

A *configuration* at any instant of a basic tissue P system is described by all multisets of objects over  $\Gamma$  associated with all the cells present in the system, and the multiset of objects over  $\Gamma \setminus \mathcal{E}$  associated with the environment at that moment. Recall that there are infinitely many copies of objects from  $\mathcal{E}$  in the environment, and hence this set is not properly changed along the computation. For each multiset  $m$  over the input alphabet  $\Sigma$ , the *initial configuration* with input  $m$  is  $\mathcal{C}_0 = (\mathcal{M}_1, \dots, \mathcal{M}_{i_{in}} + m, \dots, \mathcal{M}_q; \emptyset)$ . Therefore, we have an initial configuration associated with each input multiset  $m$  (over the input alphabet  $\Sigma$ ) in this kind of systems. We will use the notation  $(\Pi + m)$  to refer to a P system  $\Pi$  such that its initial configuration is the one associated with  $m$ . A configuration is a *halting configuration* if no rule of the system is applicable to it. We say that configuration  $\mathcal{C}_1$  yields configuration  $\mathcal{C}_2$  in one *transition step*, denoted by  $\mathcal{C}_1 \Rightarrow_{\Pi} \mathcal{C}_2$ , if we can pass from  $\mathcal{C}_1$  to  $\mathcal{C}_2$  by applying the rules from  $\mathcal{R}$  following the previous remarks.

A *computation* of  $\Pi$  is a (finite or infinite) sequence of configurations such that: (a) the first term of the sequence is the initial configuration  $\mathcal{C}_0$  of the system associated with a given input; (b) for each  $n \geq 2$  the  $n$ -th configuration of the sequence is obtained from the previous configuration by applying a maximal

multiset of rules of the system as described above; and (c) if the sequence is finite (called *halting computation*) then the last term of the sequence must be a halting configuration. Only halting computations give a result, which is encoded by the objects present in the output region  $i_{out}$  in the halting configuration. The result of a computation can be defined in various ways, just like in the cell-like case. Obviously, when the output is collected in the environment, symbols from  $\mathcal{E}$  must be ignored.

If  $\mathcal{C} = \{\mathcal{C}_t\}_{0 \leq t \leq r}$  of  $\Pi$  ( $r \in \mathbb{N}$ ) is a halting computation, then the *length* of  $\mathcal{C}$ , denoted by  $|\mathcal{C}|$ , is  $r$ .

## 2.1 Cell division and cell separation

Reproduction is doubtlessly one of the fundamental mechanisms on every living being. Thus, there is a clear motivation to try to get inspiration from the various processes that generate new cells (or new membranes, in general) and to adapt them into the tissue P systems framework. Moreover, as mentioned in the Introduction, division rules (*mitosis*), and separation rules (*membrane fission*) have been already introduced for cell-like P systems [12, 8].

**Definition 2.** A tissue P system with cell division of degree  $q \geq 1$  is a tuple  $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$ , where:

1.  $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_c, i_{in}, i_{out})$  is a basic tissue P system, where  $\mathcal{R}_c$  is the set of communication rules in  $\mathcal{R}$ .
2.  $\mathcal{R}$  may also contain cell division rules of the form  $[a]_i \rightarrow [b]_i[c]_i$ , where  $i \in \{1, 2, \dots, q\}$ ,  $i \neq i_{out}$  and  $a, b, c \in \Gamma$ .

**Definition 3.** A tissue P system with cell separation of degree  $q \geq 1$  is a tuple  $\Pi = (\Gamma, \Gamma_1, \Gamma_2, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}, i_{out})$ , where:

1.  $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \dots, \mathcal{M}_q, \mathcal{R}_c, i_{in}, i_{out})$  is a basic tissue P system, where  $\mathcal{R}_c$  is the set of communication rules in  $\mathcal{R}$ .
2.  $\{\Gamma_1, \Gamma_2\}$  is a partition of  $\Gamma$ , that is,  $\Gamma = \Gamma_1 \cup \Gamma_2$ ,  $\Gamma_1, \Gamma_2 \neq \emptyset$ ,  $\Gamma_1 \cap \Gamma_2 = \emptyset$ .
3.  $\mathcal{R}$  may also contain cell separation rules of the form  $[a]_i \rightarrow [\Gamma_1]_i[\Gamma_2]_i$ , where  $i \in \{1, \dots, q\}$ ,  $a \in \Gamma$  and  $i \neq i_{out}$ .

A tissue P system with cell division is a basic tissue P system that allows cell division rules. When applying a division rule  $[a]_i \rightarrow [b]_i[c]_i$ , under the influence of object  $a$ , the cell with label  $i$  is divided into two cells with the same label; in the first copy, object  $a$  is replaced by object  $b$ , in the second one, object  $a$  is replaced by object  $c$ ; all the other objects are replicated and copies of them are placed in the two new cells.

A tissue P system with cell separation is a basic tissue P system that allows cell separation rules. When applying a separation rule  $[a]_i \rightarrow [\Gamma_1]_i[\Gamma_2]_i$ , in reaction with an object  $a$ , the cell  $i$  is separated into two cells with the same label; at the same time, object  $a$  is consumed; all the other objects in the cell are distributed (not replicated): those from  $\Gamma_1$  are placed in the first cell, while those

from  $I_2$  are placed in the second cell. The output cell  $i_{out}$  cannot be divided nor separated.

The label of a cell precisely identifies the rules which can be applied to it. Note that in the previous definitions  $\{1, \dots, q\}$  is used as the set of labels, but without loss of generality any finite set can be considered instead. The rules are used in a non-deterministic maximally parallel manner with the following restriction: when a cell is divided (or separated), the objects inside that cell do not get involved in any communication rule during this step. The two new resulting cells could participate in the interaction with other cells or the environment by means of communication rules at the next step – provided that they are not divided (or separated) again.

## 2.2 Recognizer Tissue P Systems

A *decision problem* is a pair  $(I_X, \theta_X)$  where  $I_X$  is a language over a finite alphabet (whose elements are called *instances*) and  $\theta_X$  is a total Boolean function over  $I_X$ . There are many different ways to describe instances of a decision problem, but we assume that each problem has associated with it a fixed *reasonable encoding scheme* (in the sense of [3], page 10) which provides a string associated with each problem instance. The *size* of an instance  $u \in I_X$  is the length of the string associated with it by means of a reasonable encoding scheme.

A correspondence between decision problems and languages over a finite alphabet, can be established as follows. Given a decision problem  $X = (I_X, \theta_X)$ , its associated language is  $L_X = \{w \in I_X : \theta_X(w) = 1\}$ . Conversely, given a language  $L$  over an alphabet  $\Sigma$ , its associated decision problem is  $X_L = (I_{X_L}, \theta_{X_L})$ , where  $I_{X_L} = \Sigma^*$ , and  $\theta_{X_L} = \{(x, 1) \mid x \in L\} \cup \{(x, 0) \mid x \notin L\}$ . The solvability of decision problems is defined through the recognition of the languages associated with them by means of languages recognizer devices.

**Definition 4.** A tissue P system of degree  $q \geq 1$  is a recognizer system if:

1. The working alphabet  $\Gamma$  has two distinguished objects **yes** and **no** being, at least, one copy of them present in some initial multisets, but none of them are present in the alphabet of the environment.
2. All computations halt.
3. If  $\mathcal{C}$  is a computation of  $\Pi$ , then either object **yes** or object **no** (but not both) must have been released into the environment, and only at the last step of the computation.

Note that, because of the first condition, the presence or absence of objects **yes** and **no** in the environment can be accounted for in any configuration. Note also that all computations are finite as a consequence of the second condition, and thus it is possible to refer to their “last step”.

Given a recognizer tissue P system  $\Pi$  and a computation  $\mathcal{C}$  of  $\Pi$ , we say that  $\mathcal{C}$  is an *accepting computation* (respectively, *rejecting computation*) if object **yes** (respectively, object **no**) appears in the environment associated with the corresponding halting configuration of  $\mathcal{C}$ . Note that, since  $\Pi$  is a recognizer

system, neither object **yes** nor **no** appears in the environment associated with any non-halting configuration of  $\mathcal{C}$ .

For each natural number  $k \geq 1$ , we denote by  $\mathbf{TDC}(k)$  (respectively,  $\mathbf{TSC}(k)$ ) the class of recognizer tissue P systems with cell division (respectively, with cell separation) and communication rules with length at most  $k$ . We denote by  $\widehat{\mathbf{TDC}}(k)$  (respectively,  $\widehat{\mathbf{TSC}}(k)$ ) the class of recognizer tissue P systems with cell division (respectively, with cell separation), with communication rules with length at most  $k$ , and without environment.

Now, we define what it means to solve a decision problem in the framework of tissue P systems efficiently and in a uniform way. Since we define each tissue P system to work on a finite number of inputs, to solve a decision problem we define a numerable family of tissue P systems.

**Definition 5.** *We say that a decision problem  $X = (I_X, \theta_X)$  is solvable in a uniform way and polynomial time by a family  $\Pi = \{\Pi(n) \mid n \in \mathbb{N}\}$  of recognizer P systems if the following holds:*

1. *The family  $\Pi$  is polynomially uniform by Turing machines, that is, there exists a deterministic Turing machine working in polynomial time which constructs the system  $\Pi(n)$  from  $n \in \mathbb{N}$ .*
2. *There exists a pair  $(\text{cod}, s)$  of polynomial-time computable functions over  $I_X$  such that:*
  - (a) *for each instance  $u \in I_X$ ,  $s(u)$  is a natural number and  $\text{cod}(u)$  is an input multiset of the system  $\Pi(s(u))$ ;*
  - (b) *for each  $n \in \mathbb{N}$ ,  $s^{-1}(n)$  is a finite set;*
  - (c) *the family  $\Pi$  is polynomially bounded with regard to  $(X, \text{cod}, s)$ , that is, there exists a polynomial function  $p$ , such that for each  $u \in I_X$  every computation of  $\Pi(s(u))$  with input  $\text{cod}(u)$  is halting and it performs at most  $p(|u|)$  steps;*
  - (d) *the family  $\Pi$  is sound with regard to  $(X, \text{cod}, s)$ , that is, for each  $u \in I_X$ , if there exists an accepting computation of  $\Pi(s(u))$  with input  $\text{cod}(u)$ , then  $\theta_X(u) = 1$ ;*
  - (e) *the family  $\Pi$  is complete with regard to  $(X, \text{cod}, s)$ , that is, for each  $u \in I_X$ , if  $\theta_X(u) = 1$ , then every computation of  $\Pi(s(u))$  with input  $\text{cod}(u)$  is an accepting one.*

From the soundness and completeness conditions above we deduce that every P system  $\Pi(n)$  is *confluent*, in the following sense: every computation of a system with the *same* input multiset must always give the *same* answer.

Let  $\mathbf{R}$  be a class of recognizer P systems. We denote by  $\mathbf{PMC}_{\mathbf{R}}$  the set of all decision problems which can be solved in a uniform way and polynomial time by means of families of systems from  $\mathbf{R}$ . The class  $\mathbf{PMC}_{\mathbf{R}}$  is closed under complement and polynomial-time reductions [16].



### 3 Computational Efficiency of Tissue P Systems without environment

It is well known that tissue P systems with cell division and tissue P systems with cell separation are able to solve computationally hard problems efficiently. Specifically, **NP**-complete problems have been solved in polynomial time in [19] by using families of tissue P systems with cell division and communication rules of length at most 2, and by using families of tissue P systems with cell separation and communication rules of length at most 3. Thus,

$$\mathbf{NP} \cup \mathbf{co-NP} \subseteq \mathbf{PMC}_{\mathbf{TDC}(2)} \cap \mathbf{PMC}_{\mathbf{TSC}(3)}$$

In [4, 9, 10] it has been proved that only tractable problems can be efficiently solved by using families of tissue P systems with cell division and communication rules of length 1 (or with cell separation and communication rules of length bounded by 2). That is,  $\mathbf{P} = \mathbf{PMC}_{\mathbf{TDC}(1)} = \mathbf{PMC}_{\mathbf{TSC}(1)} = \mathbf{PMC}_{\mathbf{TSC}(2)}$ . Therefore, in the framework of tissue P systems with cell division (respectively, cell separation), passing the maximum length of communication rules of the systems from 1 to 2 (respectively, from 2 to 3) amounts to passing from non-efficiency to efficiency, assuming that  $\mathbf{P} \neq \mathbf{NP}$ . That is, the cooperation of 2 objects (respectively, 3 objects) in the communication rules is a key feature that allows efficient solutions of **NP**-complete problems.

#### 3.1 Efficiency of tissue P systems with cell division and without environment

In this section, we give a family of tissue P systems with cell division, communication rules of length at most 2, and without environment which solves the **HAM-CYCLE** problem, a well known **NP**-complete problem [3], in polynomial time, according to Definition 5.

Let us recall that the **HAM-CYCLE** problem is the following: *given a directed graph, to determine whether or not there exists a Hamiltonian cycle in the graph.*

The idea is the following: starting from the family  $\Pi = \{\Pi(n) \mid n \in \mathbb{N}\}$  of tissue P systems from  $\mathbf{TDC}(2)$  provided in [19], we construct a family  $\Pi' = \{\Pi'(n) \mid n \in \mathbb{N}\}$  of tissue P systems from  $\mathbf{TDC}(2)$  such that  $\Pi'(n)$  processes all instances  $G$  of **HAM-CYCLE** with  $n$  nodes. The construction is implemented according to Definition 6.2 in [15], in such a way that each  $\Pi'(n)$  *simulates* its counterpart  $\Pi(n)$  in an efficient way. We refer to [15] for details, but informally speaking, each computation from  $\Pi'(n)$  matches (or “simulates”) an equivalent one from  $\Pi(n)$ , except for a polynomial amount of additional auxiliary steps.

Let us recall that for each  $n \in \mathbb{N}$ ,  $\Pi(n)$  is the following tissue P system:

$$\begin{aligned} \Pi(n) = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_{in}, \mathcal{M}_h, \mathcal{M}_y, \mathcal{M}_{yes}, \mathcal{M}_{no}, \mathcal{M}_{out}, \\ \mathcal{M}_{e_{i,j,k}}(1 \leq i, j, k \leq n), \mathcal{M}_{c_i}(1 \leq i \leq n), \mathcal{R}, i_{in}, i_{out}) \end{aligned}$$

- The input alphabet is  $\Sigma = \{(i, j)_k \mid 1 \leq i, j, k \leq n\}$ .

- The working alphabet is

$$\begin{aligned} \Gamma = & \{(i, j)_k, (i, j)'_k, (i, j)''_k \mid 1 \leq i, j, k \leq n\} \cup \\ & \{(i, j)_{k,r}, (i, j)'_{k,r}, (i, j)''_{k,r} \mid 1 \leq i, j, k \leq n \wedge 1 \leq r \leq n^3\} \cup \\ & \{w_i \mid 1 \leq i \leq n^3 + 6\} \cup \{c_r, h_r, y_r \mid 1 \leq r \leq n^3\} \cup \\ & \{w, c, c', c'', h, h', h'', h''', y, y', y'', y''', y'''' , x, yes, no, \#\} \end{aligned}$$

- The alphabet of the environment is

$$\mathcal{E} = \{w_i \mid 1 \leq i \leq n^3 + 5\} \cup \{w, c'', y'', h'', y''', h''', y''''\}$$

- The initial multisets are

$$\left\{ \begin{array}{l} \mathcal{M}_{in} = c^n y h \\ \mathcal{M}_{e_{i,j,k}} = (i, j)''_{k,n^3}, \quad 1 \leq i, j, k \leq n \\ \mathcal{M}_{c_i} = c_{n^3}, \quad 1 \leq i \leq n \\ \mathcal{M}_h = h_{n^3} \\ \mathcal{M}_y = y_{n^3} \\ \mathcal{M}_{yes} = yes \\ \mathcal{M}_{no} = w_{n^3+6} no \\ \mathcal{M}_{out} = x \end{array} \right.$$

- The set  $\mathcal{R}$  consists of the following rules:

- (1)  $(no, w_r / w_{r-1}, 0)$ , for  $2 \leq r \leq n^3 + 6$ .
- (2)  $(no, w_1 / w, 0)$ .
- (3)  $[(i, j)_k]_{in} \rightarrow [(i, j)'_k]_{in} [\#]_{in}$ , for  $1 \leq i, j, k \leq n$ .
- (4)  $[(i, j)''_{k,r}]_{e_{i,j,k}} \rightarrow [(i, j)''_{k,r-1}]_{e_{i,j,k}} [(i, j)''_{k,r-1}]_{e_{i,j,k}}$ ,  
for  $1 \leq i, j, k \leq n$  and  $2 \leq r \leq n^3$ .
- (5)  $[(i, j)''_{k,1}]_{e_{i,j,k}} \rightarrow [(i, j)''_k]_{e_{i,j,k}} [(i, j)''_k]_{e_{i,j,k}}$ , for  $1 \leq i, j, k \leq n$ .
- (6)  $[c_r]_{c_i} \rightarrow [c_{r-1}]_{c_i} [c_{r-1}]_{c_i}$ , for  $1 \leq i \leq n \wedge 1 \leq r \leq n^3$ .
- (7)  $[y_r]_y \rightarrow [y_{r-1}]_y [y_{r-1}]_y$ , for  $1 \leq r \leq n^3$ .
- (8)  $[h_r]_h \rightarrow [h_{r-1}]_h [a_{r-1}]_h$ , for  $1 \leq r \leq n^3$ .
- (9)  $(in, (i, j)'_k / (i, j)''_k, e_{i,j,k})$ , for  $1 \leq i, j, k \leq n$ .
- (10)  $(in, c / c', c_i)$ , for  $1 \leq i \leq n$ .
- (11)  $(in, y / y', y)$ .
- (12)  $(in, h / h', h)$ .
- (13)  $(in, (i, j)''_k (i, j')''_{k'} / \lambda, 0)$ , for  $1 \leq i, j, j', k, k' \leq n$ .
- (14)  $(in, (i, j)''_k (i', j')''_{k'} / \lambda, 0)$ , for  $1 \leq i, i', j, k, k' \leq n$ .
- (15)  $(in, (i, j)''_k (i', j')''_{k+1} / \lambda, 0)$ , for  $1 \leq i, i', j, j', k \leq n$ , and  $j \neq i'$ .
- (16)  $(in, (i, j)''_k (i', j')''_k / \lambda, 0)$ , for  $1 \leq i, i', j, j', k \leq n$ .
- (17)  $(in, c' / c'', 0)$ .
- (18)  $(in, y' / y'', 0)$ .
- (19)  $(in, h' / h'', 0)$ .
- (20)  $(in, (i, j)''_k c'' / \lambda, 0)$  for  $1 \leq i, j, k \leq n$ .
- (21)  $(in, y'' / y''', 0)$ .
- (22)  $(in, h'' / h''', 0)$ .

- (23)  $(in, c'' h''' / \lambda, 0)$ .
- (24)  $(in, y''' / y'''' , 0)$ .
- (25)  $(in, h''' y''' / \lambda, yes)$ .
- (26)  $(yes, y''' yes / \lambda, out)$ .
- (27)  $(out, x yes / \lambda, 0)$ .
- (28)  $(no, w no / \lambda, out)$ .
- (29)  $(out, x no / \lambda, 0)$ .

- The input cell is  $i_{in} = in$ .
- The output region is the environment,  $i_{out} = 0$ .

Let us notice that  $|\Gamma| = 3n^4 + 7n^3 + 23$ ,  $|\mathcal{E}| = n^3 + 12$  and the degree of  $\Pi(n)$  is  $q = n^3 + n + 6$ . Let  $Lab_n$  denote the set of labels of cells in  $\Pi(n)$ . Besides, the execution-time is  $n^3 + 7$  if the answer is affirmative and it is  $n^3 + 8$  if the answer is negative. We thus consider  $p(n) = n^3 + 8$  as the polynomial bound on the number of steps needed for the construction, according to Definition 6.2 in [15].

Now, for each  $n \in \mathbb{N}$ , let us construct, using  $\Pi(n)$  as a starting point, a tissue P system from  $\widehat{\mathbf{TDC}}(2)$  of degree  $q_1 = 1 + (n^3 + n + 6) \cdot (n^3 + 10) + (n^3 + 12)$ ,

$$\Pi'(n) = (\Gamma', \Sigma', \mathcal{E}', \mathcal{M}'_0, \mathcal{M}'_1, \dots, \mathcal{M}'_{q_1-1}, \mathcal{R}', i'_{in}, i'_{out})$$

defined as follows:

- $\Gamma' = \Gamma \cup \{\alpha_j \mid 0 \leq j \leq n^3 + 7\}$ .
- $\Sigma' = \Sigma$  and  $\mathcal{E}' = \emptyset$ .
- Each one of the  $q$  cells of  $\Pi(n)$  provides a cell of  $\Pi'(n)$  with the same label. In addition,  $\Pi'(n)$  has:
  - For each one of the  $q$  cells of  $\Pi(n)$ ,  $n^3 + 9$  new cells, labelled by  $(i, 0), \dots, (i, n^3 + 8)$ , respectively, where  $i$  stands for the original label of the cell in  $\Pi(n)$ .
  - A distinguished cell labelled by 0.
  - A new cell, labelled by  $l_b$ , for each  $b \in \mathcal{E}$ .
- $\mathcal{M}'_{l_b} = \{\alpha_0\}$ , for each  $b \in \mathcal{E}$ ,  $\mathcal{M}'_{(i,0)} = \mathcal{M}_i$ , for each  $i \in Lab_n$ , and every other multiset of  $\Pi'(n)$  is initially empty.
- $\mathcal{R}' = \mathcal{R} \cup \{[\alpha_j]_{l_b} \rightarrow [\alpha_{j+1}]_{l_b} \mid b \in \mathcal{E} \wedge 0 \leq j \leq n^3 + 6\}$ 

$$\cup \{[\alpha_{n^3+7}]_{l_b} \rightarrow [b]_{l_b} \mid b \in \mathcal{E}\}$$

$$\cup \{(l_b, b/\lambda, 0) \mid b \in \mathcal{E}\}$$

$$\cup \{((i, j), a/\lambda, (i, j+1)) \mid a \in \Gamma \wedge i \in Lab_n \wedge 0 \leq j \leq n^3 + 7\}$$

$$\cup \{((i, n^3 + 8), a/\lambda, i) \mid a \in \Gamma \wedge i \in Lab_n\}$$
- $i'_{in} = (i_{in}, 0)$ , and  $i'_{out} = 0$ .

Let us notice that  $\Pi'(n)$  can be considered as an *extension* of  $\Pi(n)$  *without environment*, in the following sense:

- ★  $\Gamma \subseteq \Gamma', \Sigma \subseteq \Sigma'$  and  $\mathcal{E}' = \emptyset$ .
- ★ Each cell in  $\Pi$  is also a cell in  $\Pi'(n)$ .

- ★ There is a distinguished cell in  $\Pi'(n)$  labelled by 0 which plays the role of environment of  $\Pi(n)$ .
- ★  $\mathcal{R} \subseteq \mathcal{R}'$ , and now 0 is the label of a “normal cell” in  $\Pi'(n)$ .

Note also that this construction does not affect the maximum length of the communication rules, since the communication rules in  $\mathcal{R}' \setminus \mathcal{R}$  are of type symport and length 1.

### An Overview of the Computations

Let  $G = (V, E)$ , with  $V = \{1, \dots, n\}$  and  $E = \{(u_1, v_1), \dots, (u_p, v_p)\}$ , be an arbitrary instance of the HAM-CYCLE problem.

The *size* mapping on the set of instances is defined as  $s(G) = n$ , and the encoding of the instance is the multiset

$$\text{cod}(G) = \{(u_i, v_i)_k \mid 1 \leq i \leq p \wedge 1 \leq k \leq n \wedge (u_i, v_i) \in E\}$$

Each object  $(u_i, v_i)_k$  can be interpreted as considering arc  $(u_i, v_i)$  being “placed” in the “ $k$ -th position” in a sequence of  $n$  arcs that could be a Hamiltonian cycle.

This way of encoding arcs by means of objects is one of the keys to understand the design of the solution. A brute force approach is followed, generating all possible combinations by division and subsequently checking for each subset of  $n$  objects from  $\text{cod}(G)$  whether it represents a Hamiltonian cycle or not.

Let us now informally describe how system  $\Pi'(s(G))$  with input multiset  $\text{cod}(G)$ , denoted by  $\Pi'(s(G)) + \text{cod}(G)$ , works, in order to process the instance  $G$  of the HAM-CYCLE problem.

At the initial configuration of  $\Pi'(s(G)) + \text{cod}(G)$  we have the following:

- Cell labelled by 0 is empty.
- For each  $i \in \text{Lab}_n$ , the contents of cell  $i$  is empty and the contents of cell  $(i, 0)$  is  $\mathcal{M}_i$  (except for the case  $i = i_{in}$ , where  $\mathcal{M}'_{(in,0)} = \mathcal{M}_{in} + \text{cod}(G)$ ).
- For each  $i, j$  ( $i \in \text{Lab}_n$  and  $1 \leq j \leq n^3 + 8$ ), the contents of cell  $(i, j)$  is empty.
- For each  $b \in \mathcal{E}$ , cell labelled by  $l_b$  contains only object  $\alpha_0$ .

It is easy to check that the rules of a system  $\Pi(n)$  of the family are recursively defined from  $n$  and the amount of resources needed to build an element of the family is of a polynomial order in  $n$ . Therefore, there exists a deterministic Turing machine that builds the system  $\Pi(n)$  in time polynomial with respect to  $n$ .

At the first  $n^3 + 9$  steps of any computation  $\mathcal{C}'$  of  $\Pi'(n)$ , only the following rules can be applied:

- $\{[\alpha_j]_{l_b} \rightarrow [\alpha_{j+1}]_{l_b} \mid b \in \mathcal{E} \wedge 0 \leq j \leq n^3 + 6\}$
- $\{[\alpha_{n^3+7}]_{l_b} \rightarrow [b]_{l_b} \mid b \in \mathcal{E}\}$
- $\{(l_b, b/\lambda, 0) \mid b \in \mathcal{E}\}$
- $\{((i, j), a/\lambda, (i, j+1)) \mid a \in \Gamma \wedge i \in \text{Lab}_n \wedge 0 \leq j \leq n^3 + 7\}$

$$- \{((i, n^3 + 8), a/\lambda, i) \mid a \in \Gamma \wedge i \in Lab_n\}$$

Besides, they are applied in a deterministic manner. Then, the configuration  $C'_{n^3+9}$  of any computation  $C'$  of  $\Pi'(s(G)) + cod(G)$  is characterized by the following:

- (1) The contents of cell 0 is  $b_1^{2^{n^3+8}} \dots b_\alpha^{2^{n^3+8}}$ , where  $\mathcal{E} = \{b_1, \dots, b_\alpha\}$ .
- (2) For each  $i \in Lab_n$ , the contents of cell  $i$  is  $\mathcal{M}_i$  (except for the case  $i = i_{in}$ , that contains  $\mathcal{M}_{i_{in}} + cod(G)$ ).
- (3) For  $i, j$  ( $i \in Lab_n$  and  $0 \leq j \leq n^3 + 8$ ) the contents of cell  $(i, j)$  is empty.
- (4) For each  $b \in \mathcal{E}$ , there exist  $2^{n^3+8}$  cells labelled by  $l_b$  whose content is empty.

Basically, this is the “initial” configuration of the system  $\Pi(s(G)) + cod(G)$ , together with a large number of empty cells. Therefore, from step  $n^3 + 9$  any computation of  $\Pi'(s(G)) + cod(G)$  “reproduces” a computation of the system  $\Pi(s(G)) + cod(G)$  with a delay.

Bearing in mind that the family  $\Pi = \{\Pi(n) \mid n \in \mathbb{N}\}$  solves **HAM-CYCLE** problem in polynomial time, we deduce that the system  $\Pi' = \{\Pi'(n) \mid n \in \mathbb{N}\}$  also solves **HAM-CYCLE** problem in polynomial time. Hence, we have the following result:

**Theorem 1.**  $\text{HAM-CYCLE} \in \text{PMC}_{\widehat{\text{TDC}}(2)}$ .

That is, a uniform solution working in polynomial time has been found for an **NP**-complete problem using an empty environment alphabet. Hence, the environment does not play a relevant role in recognizer tissue P systems with cell division with respect to the efficiency of these models.

### 3.2 Non-efficiency of tissue P systems with cell separation and without environment

In [6] it has been proved that only tractable problems can be efficiently solved by using tissue P systems with cell separation where there is not an environment having infinitely many copies of some objects. Thus, tissue P systems with cell separation and without environment are non-efficient in the sense that they are not capable to solve **NP**-complete problems in polynomial time, according to Definition 5, assuming that  $\mathbf{P} \neq \mathbf{NP}$ .

**Theorem 2.** For each  $k \in \mathbb{N}, k \geq 1$  we have  $\mathbf{P} = \text{PMC}_{\widehat{\text{TSC}}(k)}$ .

Hence, the environment plays a relevant role in recognizer tissue P systems with cell separation with respect the efficiency of these models. That is, by using environment, **NP**-complete problems can be solved in polynomial time, but this is not possible when the initial environment is empty.

Another interesting consequence of the previous result is the following. In the framework of recognizer tissue P systems without environment, the kind of rules provides a frontier of the efficiency, that is, passing from division rules to separation rules amounts to passing from efficiency to non-efficiency, assuming that  $\mathbf{P} \neq \mathbf{NP}$ .

## 4 Conclusions

In this paper we have discussed how allowing an infinite supply of objects in the environment determines (or not) that the model of tissue P systems considered will be efficient or not.

More precisely, we have highlighted the key role that environment plays in the case of tissue P systems with cell separation. It does actually constitute a borderline between efficiency and non-efficiency for the classes  $\mathbf{TSC}(k)$  and  $\widehat{\mathbf{TSC}}(k)$ , for every  $k \geq 3$ . However, it is important to note that cooperation (of at least 3 objects) in the communication rules is another important ingredient, since we cannot get efficient solutions with tissue P systems with cell separation and communication rules of length bounded by 2, irrespectively of using environment or not [10].

On the other hand, the environment has been shown to be an irrelevant ingredient in the case of tissue P systems with cell division. Indeed, a uniform polynomial solution has been described for  $\mathbf{HAM-CYCLE}$  using a family of tissue P systems with cell division and without environment from  $\widehat{\mathbf{TDC}}(2)$ . Note that the borderline of efficiency concerning the length of communication rules remains the same as what was already known when environment is allowed: symport of length 1 versus cooperation of 2 objects.

## Acknowledgements

The work was supported by TIN2012-37434 Project of the Ministerio de Ciencia e Innovación of Spain and Project of Excellence with *Investigador de Reconocida Valía*, from Junta de Andalucía, grant P08 – TIC 04200, both cofinanced by FEDER funds.

## References

1. A. Alhazov, T.O. Ishdorj. Membrane operations in P systems with active membranes. In Gh. Păun *et al.* (eds.) *Proceedings of the Second Brainstorming Week on Membrane Computing*, Seville, Spain, February 2-7, 2004, Technical Report 01/2004, University of Seville, pp. 37-44.
2. D. Díaz-Pernil, M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. Computational efficiency of cellular division in tissue-like P systems. *Romanian Journal of Information Science and Technology* **11**, 3, (2008), 229–241.
3. M.R. Garey, D.S. Johnson. *Computers and Intractability A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, (1979).
4. R. Gutiérrez-Escudero, M.J. Pérez-Jiménez, M. Rius-Font. Characterizing tractability by tissue-like P systems. *Lecture Notes in Computer Science* **5957**, (2010), 289–300.
5. M. Ito, C. Martín Vide, Gh. Păun. A characterization of Parikh sets of ET0L languages in terms of P systems. In M. Ito, Gh. Păun, S. Yu (eds.) *Words, Semi-groups and Transducers*, World Scientific, Singapore, 2001, 239-254.

6. L.F. Macías-Ramos, M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font. The efficiency of tissue P systems with cell separation relies on the environment. E. Csuhaj-Varjú, M. Gheorghe, G. Vaszil (eds.) *Proceedings of the 13th International Conference on Membrane Computing*, Budapest, Hungary, August 28-31, 2012, pp. 277-290.
7. C. Martín Vide, J. Pazos, Gh. Păun, A. Rodríguez Patón. A New Class of Symbolic Abstract Neural Nets: Tissue P Systems. *Lecture Notes in Computer Science* **2387**, (2002), 290-299.
8. L. Pan, T.-O. Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, **10**, 5, (2004), 630-649.
9. L. Pan, M.J. Pérez-Jiménez. Computational complexity of tissue-like P systems. *Journal of Complexity*, **26**, 3 (2010), 296-315.
10. L. Pan, M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font. New frontiers of the efficiency in tissue P systems. In L. Pan, Gh. Păun, T. Song (eds.) *Pre-proceedings of Asian Conference on Membrane Computing*, Huazhong University of Science and Technology, Wuhan, China, October 15-18, 2012, pp. 61-73.
11. A. Păun, Gh. Păun. The power of communication: P systems with symport/antiport. *New Generation Computing*, **20**, 3, (2002), 295-305.
12. Gh. Păun. Attacking NP-complete problems. In *Unconventional Models of Computation, UMC'2K* (I. Antoniou, C. Calude, M. J. Dinneen, eds.), Springer-Verlag, 2000, pp. 94-115.
13. Gh. Păun. *Membrane Computing. An Introduction*. Springer-Verlag, Berlin, (2002).
14. Gh. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez. Tissue P systems with cell division. *Int. J. of Computers, Communications and Control*, **3**, 3, (2008), 295-303.
15. M.J. Pérez-Jiménez, A. Riscos-Núñez, M. Rius-Font, F.J. Romero-Campero. The role of the environment in tissue P systems with cell division. In M. García-Quismondo *et al.* (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing, Volume II*, Seville, Spain, January 30- February 3, 2012, Report RGNC 02/2012, Fénix Editora, 2012, pp. 89-104.
16. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini. Complexity classes in models of cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265-285.
17. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini. A polynomial complexity class in P systems using membrane division. *Journal of Automata, Languages and Combinatorics*, **11**, 4, (2006), 423-434.
18. M.J. Pérez-Jiménez, P. Sosík. Improving the efficiency of tissue P systems with cell separation. In M. García-Quismondo *et al.* (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing, Volume II*, Seville, Spain, January 30- February 3, 2012, Report RGNC 02/2012, Fénix Editora, 2012, pp. 105-140.
19. A.E. Porreca, N. Murphy, and M.J. Pérez-Jiménez. An optimal frontier of the efficiency of tissue P systems with cell division. In M. García-Quismondo *et al.* (eds.) *Proceedings of the Tenth Brainstorming Week on Membrane Computing, Volume II*, Seville, Spain, January 30- February 3, 2012, Report RGNC 02/2012, Fénix Editora, 2012, pp. 141-166.





## Short Papers



# Expressing Active Membranes by using Priorities, Catalysts and Cooperation

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science  
Blvd. King Carol I no.8, 700505 Iași, Romania  
baman@iit.tuiasi.ro, gabriel@info.uaic.ro

**Abstract.** We simulate a P system with active membranes by a transitional P system involving priorities, catalysts and cooperation. The difference in the number of applied rules depends on the fact that in a P system with active membranes a change of configuration is done by a single rule application (one step), while in the corresponding transitional P system involving priorities, catalysts and cooperation this is achieved through several steps.

## 1 Introduction

The family of membrane systems (also called P systems) is presented in the handbook [2], while several applications of membrane computing are presented in [1]. Membrane systems are distributed, parallel and non-deterministic computing models inspired by biological entities.

A P system with active membranes [2] is a construct

$$\Pi = (V, T, H, \mu, w_1, \dots, w_n, \alpha_1, \dots, \alpha_n, R), \text{ where:}$$

1.  $n \geq 1$  represents the number of membranes;
2.  $V$  is an alphabet (the total alphabet of the system);
3.  $T \subseteq V$  (the terminal alphabet);
4.  $H$  is a finite set of labels for membranes;
5.  $\mu$  is a membrane structure, consisting of  $n$  membranes, labelled in a one-to-one manner with elements of  $H$ ;
6.  $w_1, \dots, w_n$  are strings over  $V$ , describing the multisets of objects placed in the  $n$  regions of  $\mu$ ;
7.  $\alpha_1, \dots, \alpha_n$ , with  $\alpha_i \in \{+, -, 0\}$  for  $i \in \{1, \dots, n\}$ , are the initial polarisations of the membranes;
8.  $R$  is a finite set of developmental rules, of the following forms:
  - (a)  $[a \rightarrow v]_h^\alpha$ , for  $h \in H$ ,  $\alpha \in \{+, -, 0\}$ ,  $a \in V$ ,  $v \in V^*$  object evolution  
An object  $a$  placed inside a membrane evolves into a multiset of objects  $v$ , depending on the label  $h$  and the charge  $\alpha$  of the membrane. The membrane does not take part in the application of the rule and is not modified by it.

- (b)  $a[ ]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2}$ , for  $h \in H$ ,  $\alpha_1, \alpha_2 \in \{+, -, 0\}$ ,  $a, b \in V$  communication  
 An object  $a$  is introduced in the membrane labelled  $h$  and with charge  $\alpha_1$ . Furthermore, the object  $a$  may be modified to  $b$  and the polarisation may be changed from  $\alpha_1$  to  $\alpha_2$  during the operation. The label of the membrane remains unchanged.
- (c)  $[a]_h^{\alpha_1} \rightarrow [ ]_h^{\alpha_2} b$ , for  $h \in H$ ,  $\alpha_1, \alpha_2 \in \{+, -, 0\}$ ,  $a, b \in V$  communication  
 An object  $a$  is removed from the membrane labelled  $h$  and with charge  $\alpha_1$ . Furthermore, the object  $a$  may be modified to  $b$  and the polarisation may be changed from  $\alpha_1$  to  $\alpha_2$  during the operation. The label of the membrane remains unchanged.
- (d)  $[a]_h^\alpha \rightarrow b$ , for  $h \in H$ ,  $\alpha \in \{+, -, 0\}$ ,  $a, b \in V$  dissolving  
 An object  $a$  dissolves the surrounding membrane labelled  $h$  and with charge  $\alpha$ . Furthermore, the object  $a$  may be modified to  $b$  during the operation.
- (e)  $[a]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2} [c]_h^{\alpha_3}$ , for  $h \in H$ ,  $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$ ,  $a, b, c \in V$  division of elementary membranes  
 In reaction with an object  $a$ , a membrane labelled  $h$  and with charge  $\alpha$  is divided into two membranes with the same label, of potentially different polarisations. Furthermore, the object  $a$  may be replaced in the two new membranes by possibly new objects.

The above rules are applied in the usual non-deterministic maximally parallel manner (every rule that is applicable inside a region *has to* be applied in that region), subject to the following constraints: (i) any object can be subject of only one rule of any type and any membrane can be subject of only one rule of type (b)-(e); (ii) rules of type (a) are not counted as applied to membranes, but only to objects; (iii) if a membrane is dissolved, then all the objects and membranes in its region are incorporated in the surrounding region; (iv) the rules are applied in a bottom-up manner; (v) the skin membrane cannot be dissolved or divided, but it can be the subject of in/out operations.

## 2 Expressing P Systems with Active Membranes

In a P system the evolution rules can be applied according to a *priority relation*, given in the form of a partial order relation: the rule with the highest priority among the applicable rules is always the one actually applied. If there are rules that specify the evolution of several objects at the same time, then the system is *cooperative*. An intermediate case is that where there are certain objects, called *catalysts*, that do not evolve alone, but appear together with other objects in evolution rules and they are not modified by the use of the rules. (e.g.,  $cu \rightarrow cv$ ,  $u \in V^+$ ,  $v \in V^*$ ).

**Proposition 1.** *Given a P system  $\Pi$  with active membranes, its operations can be expressed by a P system  $\Pi^P$  involving priorities, catalysts and cooperative rules.*

*Proof.* Starting from the P system  $\Pi$ , we construct the P system

$$\Pi^p = (V^p, T^p, [\ ], w^p, (R^p, \rho^p)), \text{ where:}$$

1.  $V^p = \{a_h \mid a \in V, h \in H \cup H_{\mathbb{N}}\} \cup \{\alpha_h, \alpha_h^{di}, \alpha_h^m \mid \alpha \in \{+, -, 0\}, h, i \in H \cup H_{\mathbb{N}}\} \cup \{p_{ij}, p'_{ij} \mid i, j \in H \cup H_{\mathbb{N}}\}$ 
  - $a_h$  - models an object  $a$  from membrane  $h$ ;
  - $\alpha_h$  - models the polarisation  $\alpha$  of membrane  $h$ ;
  - $p_{ij}$  - represents the fact that membrane  $i$  is included in membrane  $j$ ; this object is used to model the membrane structure of the initial system;
  - $H_{\mathbb{N}} \subseteq \{h_i \mid h \in H, i \in \mathbb{N}\}$  is used to uniquely track the copies of the membranes from  $H$  created by division;
2.  $T^p = \{a_h \mid a \in T, h \in H \cup H_{\mathbb{N}}\}$  the terminal alphabet considers all combinations of terminal objects and locations from the initial system;
3.  $w^p = \{a_h, a'_h \mid a \in w_h, h \in H\} \cup \{\alpha_h \mid h \in H\} \cup \{p_{ij} \mid \text{membrane } i \text{ is included in membrane } j \text{ in } \mu\}$ 
  - $a_h$  - models an object  $a$  from the initial multiset  $w_h$ ;
  - $\alpha_h$  - models the initial polarisation  $\alpha$  of membrane  $h$ ;
- $R^p$  is a finite set of evolution rules:
  - (a) a rule  $[a \rightarrow v]_h^\alpha$  is simulated by the rule:
    - i.  $\alpha_h a_h \rightarrow \alpha_h v'_h$ ;  $v'_h$  denotes the fact that a label  $h$  is added to all objects from the multiset  $v$ , and that the newly created objects cannot be used by any other rule in the current evolution step;
  - (b) a rule  $a[\ ]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2}$  is simulated by the rule:
    - i.  $p_{hi} a_i \alpha_{1h} \rightarrow p_{hi} b'_h \alpha_{2h}$ ;
  - (c) a rule  $[a]_h^{\alpha_1} \rightarrow [\ ]_h^{\alpha_2} b$  is simulated by the rule:
    - i.  $p_{hi} a_h \alpha_{1h} \rightarrow p_{hi} b'_i \alpha_{2h}$ ;
  - (d) a rule  $[a]_h^\alpha \rightarrow b$  is simulated by the rules:
    - i.  $p_{hi} a_h \alpha_h \rightarrow p'_{hi} \alpha_h^{di} b'_i$ ;  $\alpha_h^{di}$  represents a special object that models the fact that the membrane labelled  $h$  in the initial membrane structure is marked to be dissolved ( $d$  - symbolises dissolution,  $i$  - the parent membrane of dissolved membrane  $h$ ). When a membrane is marked for dissolution, some objects  $p_{ij}$  need to be modified in order to keep track with the modification of the initial structure. To do this, the object  $p_{hi}$  is replaced with the object  $p'_{hi}$  in order to announce the children of membrane  $h$  (if any) to change their parent to  $i$ .
    - ii.  $p'_{hi} p_{jh} \rightarrow p'_{hi} p_{ji}$ ; any membrane contained in the dissolved membrane  $h$ , if any, changes its parent from  $h$  to  $i$ .
    - iii.  $p'_{hi} \rightarrow \varepsilon$ ; if there are no more membranes with parent  $h$ , the intermediate object  $p'_{hi}$  is removed.
    - iv.  $\alpha_h^{di} a_h \rightarrow \alpha_h^{di} a'_i$ ; any object from membrane  $h$  moves to membrane  $i$ .
    - v.  $\alpha_h^{di} \rightarrow \varepsilon$ ; if there are no more objects in the dissolving membrane  $h$ , then intermediate object  $\alpha_h^{di}$  is removed.

The rules are applied according to the following sequence of priorities:

$$(d).i > (d).ii > (d).iii > (d).iv > (d).v$$
  - (e) a rule  $[a]_h^{\alpha_1} \rightarrow [b]_h^{\alpha_2} [c]_h^{\alpha_3}$  is simulated by the rules:

- i.  $p_{hi}a_h\alpha_{1h} \rightarrow p_{h_1i}b'_{h_1}p_{h_2i}c'_{h_2}\alpha_h^m$ ;  $\alpha_h^m$  represents a special object that models the fact that the membrane labelled  $h$  in the initial membrane structure is multiplied ( $m$  - symbolises multiplication). In order to keep track which objects belong to which membrane in the initial system, we consider that the new copies of the membrane and its inner objects have labels that uniquely identify them, namely  $h_1$  and  $h_2$ . To be able to apply similar rules as for the initial membrane  $h$  also to the newly created membranes labelled by  $h_1$  and  $h_2$  we duplicate the rules from  $h$  in the newly created membranes by replacing the objects  $a_h$  by the objects  $a_{h_1}$  or  $a_{h_2}$ , respectively.
- ii.  $\alpha_h^m a_h \rightarrow \alpha_h^m a'_{h_1} a'_{h_2}$ ; in the presence of the object  $\alpha_h^m$  any object from the initial membrane  $h$  is duplicated in the two new copies of membrane  $h$ : the object  $a_h$  is replaced by the objects  $a'_{h_1}$  and  $a'_{h_2}$ .
- iii.  $\alpha_h^m \rightarrow \alpha_{h_1}\alpha_{h_2}$ ; after all objects of membrane  $h$  are replicated into the new copies of membrane  $h$ , namely  $h_1$  and  $h_2$ , the object  $\alpha_h^m$  is replaced with the objects  $\alpha_{h_1}$  and  $\alpha_{h_2}$ , representing the polarisations of the newly created membranes.

The rules are applied according to the following sequence of priorities:

$$(e).i > (e).ii > (e).iii$$

- (f)  $a'_h \rightarrow a_h$ ; in order to move to the next step of the evolution, the primed objects (e.g.,  $a'_h$ ) are transformed into simple objects (e.g.,  $a_h$ ).

The rules are simulated using the application constraints of P systems with active membranes, except for rule (f) which has the lowest priority among all rules and is applied last in order to prepare the system for a new evolution step.  $\square$

*Remark 1.* We end by emphasising the size of the P system  $\Pi^p$  with respect to that of  $\Pi$ . Consider that the largest configuration has  $m$  membranes. Thus, the cardinality of the alphabet  $V^p$  is,

$$\text{card}(V^p) = m \times (\text{card}(V) + 10 \times m)$$

while the cardinality of the rule set  $R^p$  is,

$$\begin{aligned} \text{card}(R^p) = m \times (\text{card}(R(a)) + \text{card}(R(a)) + \text{card}(R(c)) + \\ + 5 \times \text{card}(R(d)) + 3 \times \text{card}(R(e)) + \text{card}(V^p)), \end{aligned}$$

where  $R(a)$  denotes the number of ( $a$ ) rules from the set of rules  $R$ .

## References

1. G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez. *Applications of Membrane Computing*, Springer, Natural Computing Series, 2006.
2. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.) *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.

# Causal Sequences and Indexed Multisets

Gabriel Ciobanu<sup>1</sup>, Dragoş Sburlan<sup>2</sup>

<sup>1</sup> Romanian Academy, Institute of Computer Science  
Blvd. Carol I no. 8, 700505 Iasi, gabriel@info.uaic.ro

<sup>2</sup> Ovidius University of Constanta, Romania  
Faculty of Mathematics and Informatics, dsburlan@univ-ovidius.ro

**Abstract.** We explore the principle of causality in the parallel multiset rewriting systems.

## 1 Introduction

This paper explores the concept of causality in membrane and multiset rewriting systems. Its goal is to capture the causal dependencies existing among executions of rules, while abstracting away from other aspects. In the membrane computing literature there are several attempts to formalize causal semantics (e.g., [1], [2] and [3]), most of them proposing a notion of causality based on the temporal order of single rule applications. Our new approach introduces regular expressions to define the causal relation between executions of rules; the time between the moments when these rules compete for objects can be also specified in the definition of regular expressions. We define causal sequences as a method for modelling different possible evolutions in metabolic networks, and their causal relationships.

We also use indexed multisets in the context of multiset rewriting systems. The indexes are used to keep track of causal relations. An indexed rewriting system has rules of the form  $(r, i)$  which correspond to a rule  $r$  applied at step  $i$ . Indexing allows to identify the rules which can be applied “early” without changing the result of the evolution. In this situation the “early” applied rule  $r$  is causally independent from the rules which can be postponed to be applied after  $r$ , which leads to a specific notion of causality. When a rule is applied as early as possible, the rules preceding it represent its cause. This approach naturally extends to multisets of rules. Indexing is suitable for the causal analysis of the evolution strategy used in membrane systems (maximally parallel application) as well as for the one used in Petri nets (unconditional application).

### 1.1 Preliminaries

A multiset rewriting system is a tuple  $(O, \mathcal{R}, w_0)$  consisting of an alphabet of objects  $O$ , a set of rules  $\mathcal{R}$  and an initial multiset of objects  $w_0$ . Each rule  $r \in \mathcal{R}$  has two associated non-empty multisets over  $O$ , denoted by  $lhs(r)$  and  $rhs(r)$ ; the notation used for a rule is  $r : lhs(r) \rightarrow rhs(r)$ . The set of labels

of the rules from  $\mathcal{R}$  (uniquely identifiable) is denoted by  $\mathcal{L}$ . When considering a multiset  $F$  of rules we extend the notations for left hand side and right hand side to the entire multiset:  $lhs(F) = \sum_{r \in \mathcal{R}} F(r) \cdot lhs(r)$  and similarly  $rhs(F) = \sum_{r \in \mathcal{R}} F(r) \cdot rhs(r)$ .

A multiset rewriting system  $(O, \mathcal{R}, w_0)$  evolves by applying a multiset of rules to the initial multiset (configuration), then applying yet another multiset of rules to the multiset obtained from the first application and so on, possibly imposing certain restrictions on the applied multisets of rules.

We call *step sequence* a sequence of multisets  $F_1 \dots F_n$  of rules such that there exists a sequence of transitions of the form  $w_0 \xrightarrow{F_1} w_1 \dots w_{n-1} \xrightarrow{F_n} w_n$ . A *maximally parallel step sequence* is a step sequence given by a transition sequence  $w_0 \xrightarrow{F_1} w_1 \dots w_{n-1} \xrightarrow{F_n} w_n$  such that every multiset of rules  $F_i$  applied throughout is maximal with respect to application to  $w_{i-1}$ .

## 2 Causal Sequence System

A *Causal Sequence System* (CSS, for short) is a multiset rewriting system  $\Pi = (O, C, \mathcal{R}, w_0, E)$ , where  $C \subseteq O$  (the *catalysts*); the set  $E$  is a finite set of regular expressions over  $\mathcal{L} \cup \{d\}$  (where  $d$  is a special symbol (the “delay” symbol),  $d \notin \mathcal{L}$ ); moreover, if  $e \in E$  then  $L(e) \subseteq (\mathcal{L} \cup \{d\})^* \mathcal{L} (\mathcal{L} \cup \{d\})^*$ . Here we consider *non-cooperative* rules  $l : a \rightarrow v$  or *catalytic* rules  $l : ca \rightarrow cv$ , where  $l \in \mathcal{L}$ ,  $a \in O \setminus C$ ,  $v \in (O \setminus C)^*$ , and  $c \in C$ ;

The evolution of  $\Pi$  is determined by the followings. Let  $E = \{e_1, \dots, e_s\}$ ; in addition, let  $L_1, \dots, L_s$  be the corresponding regular languages. A word  $l_0 \dots l_t \in L_i$ ,  $1 \leq i \leq s$  is called a *causal sequence* and illustrates the fact that the corresponding rules (if there exist such corresponding rules; recall that  $d$  is not associated with any rule) are applied (if possible) in the implicit order of symbols. Given a multiset of objects  $w$ , a causal sequence  $l_0 \dots l_t$  is *applicable* to  $w$  if the rule having the label  $l_0$  is applicable to  $w$  or  $l_0 = d$ ; similarly, a causal sequence is *started* if the rule labelled with  $l_0$  is applied to  $w$  or  $l_0 = d$ . A computation of  $\Pi$  is determined by a maximally parallel step sequence such that at each step the multiset of rules to be applied is selected as follows. For a causal sequence  $l_0 \dots l_t$  that is started in configuration  $k$ , the rule labelled  $l_i$ ,  $1 \leq i \leq t$ ,  $l_i \neq d$ , compete for objects in configuration  $k + i$  iff the rules labelled  $l_{i-j}$ ,  $1 \leq j \leq i$ ,  $l_{i-j} \neq d$  were applied in the corresponding configurations  $k + i - j$ . A started causal sequence is said to be *entirely applied* if the rules corresponding to all labels were applied in the given order, in consecutive configurations; in case there exists a rule labelled  $l_i$ ,  $1 \leq i \leq t$ ,  $l_i \neq d$ , that lost the competition on objects, or if the rule cannot be applied, then the started causal sequence is said to be *interrupted*; the executions of the remaining rules in the subsequent configurations are dropped. In any configuration, new causal sequences from each  $L_i$ ,  $1 \leq i \leq s$ , are nondeterministically selected for applications. Given such a causal sequence and a configuration  $w_k$ , if the first label of rule appears in the causal sequence on position  $l \geq 0$  (the first symbols all being  $d$ ), then the corresponding rule competes for objects with other rules (from the causal



sequences in progress) after  $l$  computational steps. For configuration  $w_k$ , there might exist new causal sequences, causal sequences in progress, and interrupted causal sequences, which determine the rules to be applied in order to obtain the next configuration  $w_{k+1}$ .

A computation of  $\Pi$  is a *halting* one if no rule can be applied (all the started causal sequences are interrupted and no matter how a new causal sequence is selected for application it becomes interrupted at the first symbol corresponding to a rule) in the *halting configuration*; the result is the number of objects from a set  $\Sigma \subseteq O$  in the halting configuration. Non-halting computation yields no result. By collecting the results of all possible halting computations of  $\Pi$ , one gets  $N(\Pi)$  – the set of all natural numbers generated by  $\Pi$ .

The family of all sets of numbers computed by CSS systems and with a list of features  $f$  is denoted by  $NOCSS(f)$ . The features considered in this paper are *ncoo* (systems using only non-cooperative rules) and *cat<sub>k</sub>* (systems using non-cooperative rules and catalytic rules with at most  $k$  catalysts).

**Theorem 1.**  $NOCSS(cat_1) = NRE$ .

A particular case is when all possible causal sequences used by a CSS system  $\Pi$  are of type  $d^{l_1}w_1d^{l_2}w_2d^{l_3}\dots d^{l_k}w_kd^{l_{k+1}}$ , where  $w_i \in \mathcal{L}_i^+$ ,  $l_i \in \mathbb{N}$ ,  $1 \leq i \leq k+1$ . We consider that the regular expressions from  $E$  are of type

$$d^* \alpha_1 d^* \alpha_2 d^* \dots d^* \alpha_k d^*,$$

where each  $\alpha_j$ ,  $1 \leq j \leq k$ , are regular expressions over  $\mathcal{L}_i$  which use only grouping and Boolean OR operations in their definitions

For a causal sequence  $x = d^{l_1}w_1d^{l_2}w_2d^{l_3}\dots d^{l_k}w_kd^{l_{k+1}} \in L_i$ ,  $1 \leq i \leq s$ , we define  $deg(x) = \max_{1 \leq i \leq k} \{|w_i|\}$ . For a given CSS system  $\Pi$  we define the *degree of synchronization* as  $deg(\Pi) = \max\{deg(x) \mid x \in L_i, 1 \leq i \leq s\}$ .

The family of all sets of numbers computed by CSS systems with the feature  $f \in \{ncoo, cat\}$  and of synchronization degree at most  $n$  is denoted by  $NOCSS^{d_n}(f)$ .

*Example 1.* The set  $\{2^n \mid n \geq 1\}$  is generated by  $\Pi_1 = (O, \mathcal{R}, w_0, E)$  where

$$\begin{aligned} O &= \{a, b\}; \quad \mathcal{R} = \{r_1 : b \rightarrow b, r_2 : a \rightarrow aa, r_3 : b \rightarrow \lambda\}; \\ w_0 &= ab; \quad E = \{d^* r_1 d^* r_2 d^* r_3 d^*\}. \end{aligned}$$

**Theorem 2.** For any  $n \geq 2$  and  $k \geq 1$ , we have

$$NOCSS(f) \supseteq NOCSS^{d_n}(f) \supseteq NOCSS^{d_{n-1}}(f), \quad f \in \{ncoo, cat_k\}.$$

The following results reveal various relations between the family of all the sets of numbers computed by CSS some families from formal languages [4].

**Proposition 1.**  $NOCSS^{d_1}(ncoo) \supset NCF = NREG$ .

**Theorem 3.**  $NOCSS^{d_2}(ncoo) \supseteq NET0L$ .

**Theorem 4.**  $NOCSS^{d_3}(cat_1) = NRE$ .

### 3 Indexed Multisets

A multiset  $\alpha$  over  $X \times \mathbb{N}$  is called an *indexed multiset over  $X$* . Given a multiset  $w$  over a set  $X$  we let  $(w, n)$  denote the indexed multiset over  $X$  defined by  $(w, n)(a, i) = 0$ , if  $i \neq n$  and  $(w, n)(a, i) = w(a)$  if  $i = n$ .

The index  $i$  of a pair  $(a, i)$  is used to specify that an object  $a$  is a part of a multiset produced by the  $i$ -th transition, if  $a$  denotes an object, or that a rule  $a$  is applied in the  $i$ -th transition, if  $a$  denotes a rule. Consider an indexed multiset  $\alpha$  over  $X$ . We let  $\alpha_n$  denote the multiset over  $X$  defined by  $\alpha_n(a) = \alpha(a, n)$ .

Given a multiset rewriting system  $\Pi = (O, \mathcal{R}, w_0)$  we associate to it another multiset rewriting system  $\Pi^{ind} = (O^{ind}, \mathcal{R}^{ind}, w_0^{ind})$ , with  $O^{ind} = O \times \mathbb{N}$ ,  $w_0^{ind} = (w_0, 0)$  and  $\mathcal{R}^{ind}$  defined as follows. Let  $\mathcal{R}_\tau$  be the set  $\mathcal{R} \cup \{\tau_a \mid a \in O\}$ , where  $\tau_a : a \rightarrow a$ . Then  $\mathcal{R}^{ind}$  is the set  $\mathcal{R}_\tau \times \{1, 2, \dots\}$ , namely each element of  $\mathcal{R}^{ind}$  is a pair  $(s, i)$  given by a rule, of the form  $s = r \in \mathcal{R}$  or  $s = \tau_a$ , and a non-zero index. The rules of  $\Pi^{ind}$  are of form  $(s, i) : (lhs(s), i-1) \rightarrow (rhs(s), i)$ . Here,  $s$  can be either a rule  $r$  of the initial multiset rewriting system  $\Pi$  or a special rule  $\tau_a$ . For the first situation, an application of the indexed rule  $\sigma = (r, i)$  is meant to correspond to an application of rule  $r$  during transition  $i$  in  $\Pi$ : it consumes the multiset  $lhs(r)$  (which was produced during transition  $i-1$ ) and produces the multiset  $rhs(r)$ . An application of a rule  $\sigma = (\tau_a, i)$  in a step sequence of  $\Pi^{ind}$  is meant to correspond to an object  $a$  not being consumed by any rule during a transition  $i$  in a step sequence in  $\Pi$ ; hence the effect on  $a$  is manifested only in the increment of index  $i-1$  by 1.

Consider a step sequence  $\mathbf{F} = F_1 \dots F_n$  in  $\Pi$ ,  $w_0 \xrightarrow{F^1} w_1 \dots w_{n-1} \xrightarrow{F^n} w_n$ . We associate a sequence  $\tilde{\mathbf{F}} = \tilde{F}_1 \dots \tilde{F}_n$  of multisets over  $\mathcal{R}_\tau$  to  $\mathbf{F}$  as follows: each  $\tilde{F}_i$  is obtained by adding a rule  $\tau_a$  to the multiset  $F_i$  for each object  $a$  which is not consumed in the  $i$ -th transition  $w_{i-1} \xrightarrow{F_i} w_i$ . Let  $S_{\mathbf{F}}$  denote the set of step sequences  $\mathbf{S}$  in  $\Pi^{ind}$  for which  $sum(\mathbf{S}) = (\tilde{F}_1, 1) + \dots + (\tilde{F}_n, n)$ .

**Theorem 5.** *For any step sequence  $\mathbf{F}$  in  $\Pi$ , every step sequence  $\mathbf{S} \in S_{\mathbf{F}}$  is complete. For any complete step sequence  $\mathbf{S}$  in  $\Pi^{ind}$  there exists a unique step sequence  $\mathbf{F}$  in  $\Pi$  such that  $\mathbf{S} \in S_{\mathbf{F}}$ .*

**Theorem 6.** *For any maximally parallel step sequence  $\mathbf{F}$  in  $\Pi$ , every step sequence  $\mathbf{S} \in S_{\mathbf{F}}$  is safe and complete. For any safe and complete step sequence  $\mathbf{S}$  in  $\Pi^{ind}$  there exists a unique maximally parallel step sequence  $\mathbf{F}$  in  $\Pi$  such that  $\mathbf{S} \in S_{\mathbf{F}}$ .*

**Definition 1.** *For a sequence of multisets of rules  $\mathbf{S} = S^1 \dots S^n$  in  $\Pi^{ind}$  we denote by  $\rho(\mathbf{S})$  the sequence  $(S_1, 1) \dots (S_k, k)$  where  $S = sum(\mathbf{S})$  and  $k+1 = deg(S)$ .*

**Definition 2.** *Consider two step sequences  $\mathbf{S} = S^1 \dots S^n$  and  $\mathbf{T}$  in  $\Pi^{ind}$ . Let  $\sim$  be the relation on step sequences given by  $\mathbf{T} \sim \mathbf{S}$  whenever one of the following takes place:*

- $\mathbf{T} = S^1 \dots S^{i-1}(S^i + S^{i+1})S^{i+2} \dots S^n$  for some  $i$ ;

–  $\mathbf{T} = S^1 \dots S^{i-1}(S^i - X)(S^{i+1} + X)S^{i+2} \dots S^n$  for some  $i$  and  $X < S^i$ .

**Theorem 7.** *Let  $\equiv$  be the equivalence relation generated by  $\sim$ .  
If  $\mathbf{S}$  is a step sequence in  $\Pi^{ind}$ , then  $\mathbf{S} \equiv \rho(\mathbf{S})$ .*

**Theorem 8.** *If  $\mathbf{S}$  and  $\mathbf{T}$  are two step sequences in  $\Pi^{ind}$ , then  
 $\mathbf{S} \equiv \mathbf{T} \Leftrightarrow \text{sum}(\mathbf{S}) = \text{sum}(\mathbf{T})$ .*

*Conclusion:* Inspired by biological phenomena, we have defined and studied causal sequence in membrane and multiset rewriting systems. Regular expressions are used to express the causal dependence relations existing between various executions of rules. Indexed multisets provide information concerning the timing of events such as rule application and objects being produced. Indexing represents here the idea of tracking the moments in which resources or rules become active.

## References

1. Agrigoroaiei, O. and Ciobanu, G. (2012) Quantitative Causality in Membrane Systems. *LNCS*, vol. 7184, pp. 62–72.
2. Ciobanu, G. and Lucanu, D. (2007) Events, Causality, and Concurrency in Membrane Systems, *LNCS*, vol. 4860, pp. 209–227.
3. Sburlan, D. (2012), P Systems with Chained Rules, *LNCS* vol.7184, pp. 359–370.
4. Rozenberg, G., Salomaa, A. (Eds.): *Handbook of Formal Languages*, Springer, 2004.



# Undecidability and Computational Completeness for P Systems with One- and Two-dimensional Array Insertion and Deletion Rules

Henning Fernau<sup>1</sup>, Rudolf Freund<sup>2</sup>, Sergiu Ivanov<sup>3</sup>,  
Marion Oswald<sup>2</sup>, Markus L.Schmid<sup>1</sup>, and K.G. Subramanian<sup>4</sup>

<sup>1</sup> Fachbereich 4 – Abteilung Informatikwissenschaften, Universität Trier  
D-54296 Trier, Germany

Email: {fernau,MSchmid}@uni-trier.de

<sup>2</sup> Technische Universität Wien, Institut für Computersprachen  
Favoritenstr. 9, A-1040 Wien, Austria

Email: rudi@emcc.at

<sup>3</sup> Laboratoire d'Algorithmique, Complexité et Logique, Université Paris Est  
Créteil Val de Marne, 61, Av. Gén. de Gaulle, 94010 Créteil, France

Email: sergiu.ivanov@u-pec.fr

<sup>4</sup> School of Computer Sciences, Universiti Sains Malaysia, 11800 Penang, Malaysia  
Email: kgsmani1948@yahoo.com

In the string case, the *insertion* operation was first considered in [7, 8] and after that related insertion and *deletion* operations were investigated, e.g., in [9, 15]. In [10] *contextual* grammars were introduced, for an overview see [11].

Array insertion grammars have already been considered as *contextual array grammars* in [5], whereas the inverse interpretation of a contextual array rule as a deletion rule has newly been introduced in [2] and [3], which continued the research on P systems with left and right insertion and deletion of strings, see [6]. The results described in this note were elaborated in [3] for one-dimensional arrays and in [2] for two-dimensional arrays.

## Sequential Grammars

A (*sequential*) *grammar*  $G$  (see [4]) is a construct  $(O, O_T, w, P, \Longrightarrow_G)$  where  $O$  is a set of *objects*,  $O_T \subseteq O$  is a set of *terminal objects*,  $w \in O$  is the *axiom* (*start object*),  $P$  is a finite set of *rules*, and  $\Longrightarrow_G \subseteq O \times O$  is the *derivation relation* of  $G$  induced by the rules in  $P$ . We consider *grammars of type*  $X$ , e.g., *string grammars* with regular, etc. rules or with insertion and deletion rules, and *array grammars* with regular, etc. array rules as well as with array insertion and deletion rules. In [6], left and right insertions and deletions of strings were considered; the corresponding types of grammars using rules inserting strings of length at most  $k$  and deleting strings of length at most  $m$  at either side of a string are denoted by  $D^m I^k$ .

$L_*(G) = \{v \in O_T \mid w \xRightarrow{*}_G v\}$  is the *language generated by*  $G$  (in the  $*$ -mode);  $L_t(G) = \{v \in O_T \mid (w \xRightarrow{*}_G v) \wedge \nexists z (v \Longrightarrow_G z)\}$  is the *language generated by*  $G$  *in the*  $t$ -*mode*, i.e., the set of all terminal objects derivable from the axiom in a halting computation. The family of languages generated by grammars of type  $X$  in the derivation mode  $\delta$ ,  $\delta \in \{*, t\}$ , is denoted by  $\mathcal{L}_\delta(X)$ .

If for every  $G$  of type  $X$ ,  $G = (O, O_T, w, P, \Rightarrow_G)$ , we have  $O_T = O$ , then  $X$  is called a *pure* type, otherwise it is called *extended*.

The family of recursively enumerable string languages and of  $d$ -dimensional array languages is denoted by  $\mathcal{L}_*(ARB)$  and  $\mathcal{L}_*(d-ARBA)$ , respectively.

## Arrays and array grammars

Let  $d \in \mathbb{N}$ ; then a  $d$ -dimensional array  $\mathcal{A}$  over an alphabet  $V$  is a function  $\mathcal{A} : \mathbb{Z}^d \rightarrow V \cup \{\#\}$ , where  $\text{shape}(\mathcal{A}) = \{v \in \mathbb{Z}^d \mid \mathcal{A}(v) \neq \#\}$  is finite and  $\# \notin V$  is called the *background* or *blank symbol*. The set of all  $d$ -dimensional arrays over  $V$  is denoted by  $V^{*d}$ . For  $v \in \mathbb{Z}^d$ ,  $v = (v_1, \dots, v_d)$ , the norm of  $v$  is  $\|v\| = \max\{|v_i| \mid 1 \leq i \leq d\}$ . The *translation*  $\tau_v : \mathbb{Z}^d \rightarrow \mathbb{Z}^d$  is defined by  $\tau_v(w) = w + v$  for all  $w \in \mathbb{Z}^d$ . For any array  $\mathcal{A} \in V^{*d}$ ,  $\tau_v(\mathcal{A})$ , the corresponding  $d$ -dimensional array translated by  $v$ , is defined by  $(\tau_v(\mathcal{A}))(w) = \mathcal{A}(w - v)$  for all  $w \in \mathbb{Z}^d$ . For a (non-empty) finite set  $W \subset \mathbb{Z}^d$  the norm of  $W$  is defined as  $\|W\| = \max\{\|v - w\| \mid v, w \in W\}$ .

$[\mathcal{A}] = \{\mathcal{B} \in V^{*d} \mid \mathcal{B} = \tau_v(\mathcal{A}) \text{ for some } v \in \mathbb{Z}^d\}$  is the equivalence class of arrays with respect to linear translations containing  $\mathcal{A}$ . The set of all equivalence classes of  $d$ -dimensional arrays over  $V$  with respect to linear translations is denoted by  $[V^{*d}]$  etc.

$G_A = ((N \cup T)^{*d}, T^{*d}, \mathcal{A}_0, P, \Rightarrow_{G_A})$  is called a  $d$ -dimensional array grammar, where  $N$  is the alphabet of *non-terminal symbols*,  $T$  is the alphabet of *terminal symbols*,  $N \cap T = \emptyset$ ,  $\mathcal{A}_0 \in (N \cup T)^{*d}$  is the *start array*,  $P$  is a finite set of  $d$ -dimensional array rules over  $V$ ,  $V := N \cup T$ , and  $\Rightarrow_{G_A} \subseteq (N \cup T)^{*d} \times (N \cup T)^{*d}$  is the derivation relation induced by the array rules in  $P$ .

A  $d$ -dimensional contextual array rule (see [5]) over the alphabet  $V$  is a pair of finite  $d$ -dimensional arrays  $(\mathcal{A}_1, \mathcal{A}_2)$  with  $\text{dom}(\mathcal{A}_1) \cap \text{dom}(\mathcal{A}_2) = \emptyset$  and  $\text{shape}(\mathcal{A}_1) \cup \text{shape}(\mathcal{A}_2) \neq \emptyset$ ; we also call it an *array insertion rule*, as its effect is that in the context of  $\mathcal{A}_1$  we insert  $\mathcal{A}_2$ ; hence, we write  $I(\mathcal{A}_1, \mathcal{A}_2)$ . The pair  $(\mathcal{A}_1, \mathcal{A}_2)$  can also be interpreted as having the effect that in the context of  $\mathcal{A}_1$  we delete  $\mathcal{A}_2$ ; in this case, we speak of an *array deletion rule* and write  $D(\mathcal{A}_1, \mathcal{A}_2)$ . For any (contextual, insertion, deletion) array rule we define its norm by  $\|\text{dom}(\mathcal{A}_1) \cup \text{dom}(\mathcal{A}_2)\|$ .

The types of  $d$ -dimensional array grammars using array insertion rules of norm  $\leq k$  and array deletion rules of norm  $\leq m$  are denoted by  $d-D^m I^k A$ ; if only array insertion (i.e., contextual) rules are used, we have the case of pure grammars, and the type is denoted by  $d-CA$ .

## (Sequential) P Systems

For the the area of P systems, we refer the reader to [12] and the P page [14].

A (sequential) P system of type  $X$  with tree height  $n$  is a construct  $\Pi = (G, \mu, R, i_0)$  where  $G = (O, O_T, A, P, \Rightarrow_G)$  is a sequential grammar of type  $X$ ;  $\mu$  is the membrane (tree) structure of the system with the height of the tree being  $n$ , the membranes are uniquely labelled by labels from a set  $Lab$ ;  $R$  is a set of rules of the form  $(h, r, tar)$  where  $h \in Lab$ ,  $r \in P$ , and  $tar$ , called the *target indicator*, is taken from the set  $\{here, in, out\} \cup \{in_h \mid h \in Lab\}$ ;  $i_0$  is the initial membrane containing the axiom  $A$ .

A configuration of  $\Pi$  is a pair  $(w, h)$  where  $w$  is the current object (e.g., string or array) and  $h$  is the label of the membrane currently containing the object  $w$ . A sequence of transitions between configurations of  $\Pi$ , starting from the initial configuration  $(A, i_0)$ , is called a *computation* of  $\Pi$ . A *halting computation* is a computation ending with a configuration  $(w, h)$  such that no rule from  $R_h$  can be applied to  $w$  anymore;  $w$  is called the *result* of this halting computation if  $w \in O_T$ . The language generated by  $\Pi$ ,  $L_t(\Pi)$ , consists of all terminal objects from  $O_T$  being results of a halting computation in  $\Pi$ .

By  $\mathcal{L}_t(X-LP)$  ( $\mathcal{L}_t(X-LP^{(n)})$ ) we denote the family of languages generated by P systems (of tree height at most  $n$ ) using grammars of type  $X$ . If only the targets *here*, *in*, and *out* are used, then the P system is called *simple*, and the corresponding families of languages are denoted by  $\mathcal{L}_t(X-LsP)$  ( $\mathcal{L}_t(X-LsP^{(n)})$ ).

## Undecidability

An instance of the *Post Correspondence Problem* (PCP) is a pair of sequences of non-empty strings  $(u_1, \dots, u_n)$  and  $(v_1, \dots, v_n)$  over an alphabet  $T$ . A solution of this instance is a sequence of indices  $i_1, \dots, i_k$  such that  $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$ ; we call  $u_{i_1} \dots u_{i_k}$  the result of this solution. Let  $L((u_1, \dots, u_n), (v_1, \dots, v_n))$  denote the set of results of all solutions of the instance  $((u_1, \dots, u_n), (v_1, \dots, v_n))$  of the PCP, and let the homomorphism  $h_\Sigma$  be defined by  $h_\Sigma : \Sigma \rightarrow \Sigma^+$  with  $h_\Sigma(a) = aa'$  for all  $a \in \Sigma$ .

**Lemma 1.** *Let  $I = ((u_1, \dots, u_n), (v_1, \dots, v_n))$  be an instance of the PCP over  $T$ . Then we can effectively construct a one-dimensional array insertion P system  $\Pi$  such that  $[L(\Pi)] = \{LL'h_T(w)RR' \mid w \in L((u_1, \dots, u_n), (v_1, \dots, v_n))\}$ .*

As is well known (see [13]), the Post Correspondence Problem is undecidable, hence, the emptiness problem for  $\mathcal{L}_t(1-DIA-LP^{(k)})$  is undecidable:

**Corollary 1.** *For any  $k \geq 1$ , the emptiness problem for  $\mathcal{L}_t(1-DIA-LP^{(k)})$  is undecidable.*

For  $d \geq 2$ , even the emptiness problem for  $\mathcal{L}_t(d-CA)$  is undecidable, as in [1] it was shown that every recursively enumerable one-dimensional array language can be characterized as the projection of an array language generated by a two-dimensional contextual array grammar using rules of norm one only.

## Computational Completeness

In [6] it was shown that in the string case, for insertion and deletion rules applied on either side of strings, we have  $\mathcal{L}_t(D^1I^1-LsP^{(8)}) = \mathcal{L}_*(ARB)$ . As one-dimensional arrays can also be interpreted as strings, we immediately infer the corresponding result for one-dimensional arrays,  $\mathcal{L}_t(1-D^1I^1A-LsP^{(8)}) = \mathcal{L}_*(1-ARBA)$ . With respect to the tree height of the simple P systems, this result was improved considerably in [3]:

**Theorem 1.**  $\mathcal{L}_t(1-D^1I^1A-LsP^{(2)}) = \mathcal{L}_*(1-ARBA)$ .

Allowing norm two, we even do not need the regulating mechanism of membranes:

**Theorem 2.**  $\mathcal{L}_t(1-D^2I^2A) = \mathcal{L}_*(1-ARBA)$ .

It remains as an interesting question for future research whether this result for array grammars only using array insertion and deletion rules with norm at most two can also be achieved in higher dimensions, but at least for dimension two, as in [2], the corresponding computational completeness result has been shown for 2-dimensional array insertion and deletion P systems using rules with norm at most two.

**Theorem 3.**  $\mathcal{L}_t(2-D^2I^2A-LsP^{(2)}) = \mathcal{L}_*(2-ARBA)$ .

## References

1. H. Fernau, R. Freund, and M. Holzer, Representations of recursively enumerable array languages by contextual array grammars, *Fundamenta Informaticae* **64** (2005), pp. 159–170.
2. H. Fernau, R. Freund, S. Ivanov, M.L. Schmid, and K.G. Subramanian, Array insertion and deletion P systems, in G. Mauri, A. Dennunzio, L. Manzoni, and A. E. Porreca, Eds., *UCNC 2013*, Milan, Italy, July 1–5, 2013, LNCS **7956**, Springer 2013, pp. 67–78.
3. R. Freund, S. Ivanov, M. Oswald, and K. G. Subramanian, One-dimensional array grammars and P systems with array insertion and deletion rules, *accepted for MCU 2013*.
4. R. Freund, M. Kogler, and M. Oswald, A general framework for regulated rewriting based on the applicability of rules, in J. Kelemen and A. Kelemenová, Eds., *Computation, Cooperation, and Life - Essays Dedicated to Gheorghe Păun on the Occasion of His 60th Birthday*, LNCS **6610**, Springer, 2011, pp. 35–53.
5. R. Freund, Gh. Păun, and G. Rozenberg, Contextual array grammars, in K.G. Subramanian, K. Rangarajan, and M. Mukund, Eds., *Formal Models, Languages and Applications*, Series in Machine Perception and Artificial Intelligence **66**, World Scientific, 2007, pp. 112–136.
6. R. Freund, Yu. Rogozhin, and S. Verlan, P systems with minimal left and right insertion and deletion, in J. Durand-Lose and N. Jonoska, Eds., *UCNC 2012*, Orléans, France, September 3–7, 2012, LNCS **7445**, Springer, 2012, pp. 82–93.
7. B. Galiukschov: Semicontextual grammars. *Logica i Matem. Lingvistika*, 38–50. Tallin University (in Russian) (1981).
8. D. Haussler, *Insertion and Iterated Insertion as Operations on Formal Languages*. PhD thesis, Univ. of Colorado at Boulder, 1982.
9. L. Kari, *On Insertion and Deletion in Formal Languages*, PhD thesis, University of Turku, 1991.
10. S. Marcus, Contextual grammars, *Rev. Roum. Math. Pures Appl.* **14** (1969), pp. 1525–1534.
11. Gh. Păun, *Marcus Contextual Grammars*, Kluwer, Dordrecht, 1997.
12. Gh. Păun, G. Rozenberg, A. Salomaa, Eds., *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
13. E. L. Post, A variant of a recursively unsolvable problem, *Bull. Amer. Math. Soc.* **52** (1946), pp. 264–268.
14. The P systems Web page, <http://ppage.psystems.eu/>.
15. S. Verlan, *Study of Language-theoretic Computational Paradigms Inspired by Biology*, Habilitation thesis, University of Paris Est, 2010.



# Chemical Programming and Membrane Systems<sup>\*</sup>

Miklós Fésüs, György Vaszil

Faculty of Informatics, University of Debrecen  
P.O. Box 12, 4010 Debrecen, Hungary  
{fesus.miklos, vaszil.gyorgy}@inf.unideb.hu

In the following we would like to argue that chemical programs and membrane systems are closely related. Both models are highly parallel, non-deterministic, and distributed. A membrane system is structured set of regions working with multisets of objects which can travel between the regions. Each region have associated evolution rules that define how objects are produced. Chemical programs use the notion of a chemical solution which consists of molecules and reaction rules. Data can be seen as molecules and operations as chemical reactions. If some molecules satisfy a reaction condition, they are replaced by the result of the reaction. If no reaction is possible, the solution becomes inert, and the program terminates. Similarly to membrane computing, chemical solutions are represented by multisets. The chemical programming metaphor has been employed in different areas, see [3] for programming self-organizing systems, [4] for service orchestration, or [1] for a more general overview.

The Higher-Order Chemical Language (HOCL) is a higher-order extension of the Gamma-calculus, see [1]. A HOCL program consists of molecules, reaction rules, and sub-solutions. Reaction rules can be written as

**replace**  $P$  **by**  $M$  **if**  $C$

where  $P$  is a pattern which matches the required atoms,  $C$  is the reaction condition, and  $M$  is the result of the reaction. For example, the solution

$\{(\text{replace } x, y \text{ by } x \text{ if } x < y), 2, 7, 4, 3, 6, 8\}$

will result in a solution containing the minimum value of the molecules.

Reactions can be tagged using the syntax: **let** name = **replace** ... where we can also use a pattern  $\omega$  which matches any molecule, even the empty one. With this pattern we can extract elements from the solution. Molecules inside a solution cannot react with molecules inside a sub-solution, or outside the solution. For example, to find the least common multiple of 4 and 6, we need a nested sub-solution inside the main solution.

```
let multiplier = replace x, ω by ω if not(4 div x and 6 div x)
let min = replace x, y by x if x < y
{min, {multiplier, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
                                             22, 23, 24 ... } }
```

The { } border between sub-solutions adds sequentiality to this parallel model.

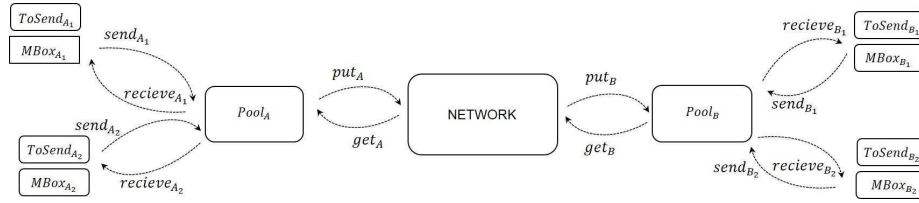
---

<sup>\*</sup> Supported in part by the Hungarian Scientific Research Fund, “OTKA”, grant no. K75952, and by the European Union through the TÁMOP-4.2.2.C-11/1/KONV-2012-0001 project which is co-financed by the European Social Fund.

### From chemical solutions to membrane systems

The border between sub-solutions can be seen as some kind of membrane, so our goal of this section is to show how to rewrite a chemical program to a membrane system. For this purpose, we use an example from [1].

**Mail system - Basic membrane system** The mail system (see Figure 1) is described by a solution. Clients send messages by adding them to the pool of messages of their domain. They receive messages from the pool of their domain and store them in their mailbox. The solution named NETWORK represents the global network interconnecting domains. Message stores are represented by sub-solutions, messages are represented by basic molecules. Solutions named  $ToSend_{d_i}$  and  $Mbox_{d_i}$  contain the messages to be sent and received by the client  $i$  of domain  $d$ , respectively, solutions named  $Pool_d$  contain the messages that the server of domain  $d$  must take care of. A client  $i$  in domain  $d$  is represented by two active molecules  $send_{d_i}$  and  $recv_{d_i}$ , a server of a domain  $d$  is represented by two active molecules  $put_d$  and  $get_d$ .



**Fig. 1.** Mail system

The movement of messages are performed by reaction rules of the form

**replace**  $A : \{msg, \omega_A\}, B : \{\omega_B\}$  **by**  $A : \{\omega_A\}, B : \{msg, \omega_B\}$  **if**  $Cond$ .

The  $send$  molecule sends the messages from the client to the pool,  $recv$  gets the messages from the pool and places them inside the message box of the client,  $put$  forwards messages to the network,  $get$  receives messages from the network.

$$send_{d_i} = \text{replace } ToSend_{d_i} : \{msg, \omega_t\}, Pool_d : \{\omega_p\} \\ \text{by } ToSend_{d_i} : \{\omega_t\}, Pool_d : \{msg, \omega_p\}$$

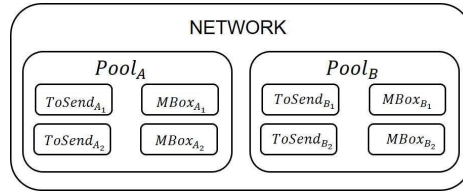
$$recv_{d_i} = \text{replace } Pool_d : \{msg, \omega_p\}, MBox_{d_i} : \{\omega_b\} \\ \text{by } Pool_d : \{\omega_p\}, MBox_{d_i} : \{msg, \omega_b\} \text{ if } recipient(msg) = i$$

$$put_d = \text{replace } Pool_d : \{msg, \omega_p\}, Network : \{\omega_n\} \\ \text{by } Pool_d : \{\omega_p\}, Network : \{msg, \omega_n\} \text{ if } recipientDomain(msg) \neq d$$

$$get_d = \text{replace } Network : \{msg, \omega_n\}, Pool_d : \{\omega_p\} \\ \text{by } Network : \{\omega_n\}, Pool_d : \{msg, \omega_p\} \text{ if } recipientDomain(msg) = d$$

MailSystem: {  $send_{A_1}, recv_{A_1}, ToSend_{A_1} : \{\dots\}, MBox_{A_1} : \{\dots\},$   
 $send_{A_2}, recv_{A_2}, ToSend_{A_2} : \{\dots\}, MBox_{A_2} : \{\dots\},$   
 $send_{A_3}, recv_{A_3}, ToSend_{A_3} : \{\dots\}, MBox_{A_3} : \{\dots\},$   
 $put_A, get_A, Pool_A, Network, put_B, get_B, Pool_B,$   
 $send_{B_1}, recv_{B_1}, ToSend_{B_1} : \{\dots\}, MBox_{B_1} : \{\dots\},$   
 $send_{B_2}, recv_{B_2}, ToSend_{B_2} : \{\dots\}, MBox_{B_2} : \{\dots\} \}$

This chemical solution can be represented by a membrane system (See Figure 2). In the corresponding membrane system message stores can be represented by membranes. The active molecules are represented by the following evolution rules:



**Fig. 2.** Membrane mail system

- $ToSend_{d_i} : msg_{d_i} \rightarrow (msg_{d_i}, out),$
- $Pool_d : msg_{d'_i} \rightarrow (msg_{d_i}, out), msg_{d_i} \rightarrow (msg_{d_i}, in_{MBox_i}),$
- $Network : msg_{d_i} \rightarrow (in_{Pool_d}).$

**Self healing - Basic extensions** We can extend the mail system to handle server breakdowns on its own (see Figure 3). by introducing backup servers for every original server in the system. A backup server automatically comes online, if the original server is down.

The molecule  $failure_d$  represents an error in the  $pool_d$ . If it is present, the system initiates a server repair with the  $repairserver_d$  rule. The  $DownOut_d$  and  $DownIn_d$  rules exchange the messages between the original and the backup server.  $Up_d$  runs after the original server is repaired: It transfers all the remaining messages from  $Pool_{d'}$  to  $Pool_d$ .

$crashserver_d = \text{replace } put_d, get_d, Up_d$   
 $\quad \text{by } put_{d'}, get_{d'}, DownIn_d, DownOut_d \text{ if } failure(d)$

$repairserver_d = \text{replace } put_{d'}, get_{d'}, DownIn_d, DownOut_d$   
 $\quad \text{by } put_d, get_d, Up_d \text{ if } recover(d)$

$DownOut_d = \text{replace } Pool_d : \{msg, \omega_p\}, Pool_{d'} : \{\omega_p\}$   
 $\quad \text{by } Pool_d : \{\omega_p\}, Pool_{d'} : \{msg, \omega_p\} \text{ if } domain(msg) \neq d$

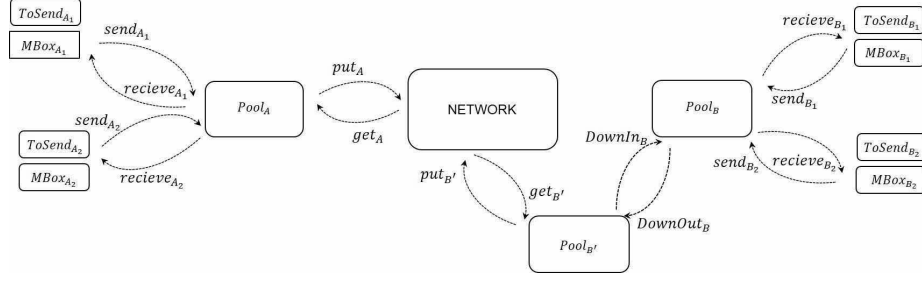


Fig. 3. Self-repairing mail system

$$\begin{aligned}
 \text{DownIn}_d &= \mathbf{replace} \ Pool_d : \{\omega_p, \}, Pool_{d'} : \{msg, \omega_p\} \\
 &\quad \mathbf{by} \ Pool_d : \{msg, \omega_p, \}, Pool_{d'} : \{\omega_p\} \text{ if } \text{domain}(msg) = d \\
 \text{Up}_d &= \mathbf{replace} \ Pool_{d'} : \{msg, \omega_p, \}, Pool_d : \{\omega_p\} \\
 &\quad \mathbf{by} \ Pool_{d'} : \{\omega_p, \}, Pool_d : \{msg, \omega_p\} \\
 \text{MailSystem} &: \{ \dots, \text{Up}_A, \text{Up}_B, Pool'_A, Pool'_B, \text{crashserver}_A, \\
 &\quad \text{repairserver}_A, \text{crashserver}_B, \text{repairserver}_B \}
 \end{aligned}$$

In the corresponding membrane system, the backup server comes online by dividing the  $Pool_d$  membrane. When the original server comes back online we simply destroy the  $Pool_{d'}$  membrane.

The evolution rules are the following:

- $Pool_d: \text{failure}_d \rightarrow [Pool_d], [Pool'_{d'}]$ ,
- $Pool_{d'}: msg_{d'_i} \rightarrow (msg_{d_i}, out), msg_{d'_i} \rightarrow (msg_{d_i}, in_{MBox_i}), Up_d \rightarrow \delta$ ,
- $Network: \text{failure}_d, msg_{d_i} \rightarrow (in_{Pool_{d'}})$ .

## References

1. J.P. Banâtre, P. Fradet, and Y. Radenac, Higher-order chemical programming style. In: [2], 84–95.
2. J.P. Banâtre, P. Fradet, J.L. Giavitto, and O. Michel, editors, *Unconventional Programming Paradigms*. Volume 3566 of *Lecture Notes in Computer Science*, Springer, Berlin Heidelberg, 2005.
3. J.P. Banâtre, P. Fradet, and Y. Radenac. Programming self-organizing systems with the higher-order chemical language. *International Journal of Unconventional Computing* 3(3):161–177 (2007)
4. J.P. Banâtre, T. Priol, and Y. Radenac, Service orchestration using the chemical metaphor. In: U. Brinkschulte, T. Givargis, S. Russo, editors, *Software Technologies for Embedded and Ubiquitous Systems*. Volume 5287 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, 2008, 79–89.
5. Gh. Păun, *Membrane Computing - An Introduction*. Springer-Verlag, Berlin, 2002.
6. G. Păun, G. Rozenberg and A. Salomaa. Eds.: *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.

# Catalytic and Purely Catalytic P Systems and P Automata: Control Mechanisms for Obtaining Computational Completeness

Rudolf Freund<sup>1</sup>, Marion Oswald<sup>1</sup>, and Gheorghe Păun<sup>2</sup>

<sup>1</sup> Faculty of Informatics, Vienna University of Technology  
Favoritenstr. 9, 1040 Vienna, Austria  
Email: {rudi,marion}@emcc.at

<sup>2</sup> Institute of Mathematics of the Romanian Academy  
PO Box 1-764, 014700 București, Romania, and  
Dep. of Computer Science and Artificial Intelligence, University of Sevilla  
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain  
gpaun@us.es, ghpaun@gmail.com

Already in the first papers introducing membrane systems (P systems), catalytic rules were considered, see [10]. Since then the question how many catalysts are needed for obtaining computational completeness in (purely) catalytic P systems, both in the generating as well as in the accepting case, has remained an interesting research topic. In [4], two (three) catalysts were shown to be sufficient for getting computational completeness with (purely) catalytic P systems in the generating case, whereas  $d + 2$  ( $d + 3$ ) are needed in the accepting case, with  $d$  being the dimension of the vectors of non-negative integers to be accepted. In all cases, just one membrane is enough.

It is still one of the most challenging open problems in the area of P systems, whether or not one (two) catalyst(s) might already be enough to obtain computational completeness with (purely) catalytic P systems. Using additional control mechanisms as, for example, priorities or promoters/inhibitors, P systems with only one catalyst can be shown to be computationally complete, e.g., see Chapter 4 of [11].

The idea of *P automata* was first published in [2] and considered at the same time under the notion of *analysing P systems* in [5]. For some more variants we refer to [9] and to Chapter 6 in [11]. We here consider both *P automata*, where the input is taken from the environment, using an additional target indication *come* as, for example, used in a special variant of communication P systems introduced by Petr Sosík (e.g., see [12]), and *accepting (analysing) P systems* where the input is given in an input membrane.

## (Purely) Catalytic P Systems

For the the area of P systems, we refer the reader to [11] and the P page [13].

A catalytic P system is a construct  $\Pi = (O, C, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$  where  $O$  is the alphabet of objects,  $C \subset O$  is the set of catalysts,  $\mu$  is the membrane structure (with  $m$  membranes),  $w_1, \dots, w_m$  are multisets of objects

present in the  $m$  regions of  $\mu$  at the beginning of a computation,  $R_1, \dots, R_m$  are finite sets of evolution rules, associated with the regions of  $\mu$ , and  $f$  is the label of the membrane region from which the outputs are taken (in the generative case) or where the inputs are put at the beginning of a computation (in the accepting case);  $f = 0$  indicates that the output/input is taken from the environment. The rules in the  $R_i$  either are *non-cooperative rules* of the form  $u \rightarrow v$ , where  $u \in O$  and  $v = (b_1, tar_1) \dots (b_k, tar_k)$  with  $b_i \in O$  and  $tar_i \in \{here, out, in\}$  or  $tar_i \in \{here, out\} \cup \{in_j \mid j \in Lab\}$ ,  $1 \leq i \leq k$ , or *catalytic rules* of the form  $ca \rightarrow cv$ , where  $c$  is a *catalyst*. In a *purely catalytic P system* we only allow catalytic rules. The evolution rules are used in the *non-deterministic maximally parallel* way. The objects present in the membrane regions of a system at a given time form a *configuration*; starting from a given *initial configuration*; a sequence of transitions using the rules in the maximally parallel way forms a *computation*. A computation is *halting* if it reaches a configuration where no rule can be applied.

In the generative case, with a halting computation we associate a *result*, in the form of the number of objects present in region  $f$  in the halting configuration (in the following, we assume  $f = 0$ ). In the accepting case, for  $f \neq 0$ , we accept all (vectors of) non-negative integers whose input given in membrane  $f$ , leads to a halting computation; the set of non-negative integers and the set of (Parikh) vectors of non-negative integers generated/accepted by halting computations in  $\Pi$  are denoted by  $N_{gen}(\Pi)/N_{acc}(\Pi)$  and  $Ps_{gen}(\Pi)/Ps_{acc}(\Pi)$ , respectively. For the input being taken from the environment, i.e., for  $f = 0$ , the multiset of all objects taken from the environment during a halting computation of  $\Pi$  is the multiset accepted by the so-called *P automaton*; the set of non-negative integers and the set of (Parikh) vectors of non-negative integers accepted by halting computations of the P automaton  $\Pi$  are denoted by  $N_{aut}(\Pi)$  and  $Ps_{aut}(\Pi)$ , respectively.

Let *RE* and *REG* denote the families of recursively enumerable and regular string languages. For a family of languages *FL*, by *PsFL* we denote the family of Parikh images of languages in *FL*, and by  $N^d FL$  the family of Parikh images of languages over an alphabet of  $d$  letters in *FL*; for  $N^1 FL$  we write *NFL*.

The family of sets  $Y_\delta(\Pi)$ ,  $Y \in \{N, Ps\}$ ,  $\delta \in \{gen, acc, aut\}$ , computed by (purely) catalytic P systems with at most  $m$  membranes and at most  $k$  catalysts is denoted by  $Y_\delta OP_m(cat_k)$  ( $Y_\delta OP_m(pcat_k)$ ). The following characterizations are known from the results proved in [4]:

1.  $Y_\delta OP_1(cat_k) = Y_\delta OP_1(pcat_{k+1}) = YRE$  for any  $k \geq 2$ ,  $Y \in \{N, Ps\}$ ,  $\delta \in \{gen, aut\}$ ;
2.  $Ps_{acc} OP_1(cat_{d+2}) = Ps_{acc} OP_1(pcat_{d+3}) = N^d RE$  for any  $d \geq 1$ .

## P Systems with Label Selection

A *P system with label selection* is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, H, W, f)$$

where  $\Pi' = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$  is a P system as defined above,  $H$  is a set of labels for the rules in the sets  $R_1, \dots, R_m$ , and  $W \subseteq 2^H$ . In any transition step in  $\Pi$  we first select a set of labels  $U \in W$  and then apply a non-empty multiset  $R$  of rules such that all the labels of these rules in  $R$  are in  $U$  and the set  $R$  cannot be extended by any further rule with a label from  $U$  so that the obtained multiset of rules would still be applicable to the existing objects in the membrane regions  $1, \dots, m$ . The family of sets  $N(\Pi)$  and  $Ps(\Pi)$  computed by P systems with label selection with at most  $m$  membranes and rules of type  $X$  is denoted by  $NOP_m(X, ls)$  and  $PsOP_m(X, ls)$ , respectively.

It is somehow surprising that with purely catalytic accepting P systems with only one catalyst, even with label selection, we obtain less than with purely catalytic P automata with only one catalyst; the following results were established in [6]:

**Lemma 1.** *For any  $m \geq 1$  and any  $Y \in \{N, Ps\}$ ,  $YREG = Y_{aut}OP_m(pcat_1)$ .*

**Lemma 2.** *Any set in  $Ps_{acc}OP_1(pcat_1)$  is the Parikh set of a regular language of the form  $\{a_1\}^* \dots \{a_n\}^*$ .*

**Theorem 1.** *For any  $m \geq 1$  and any  $Y \in \{N, Ps\}$ ,*

$$\begin{aligned} Y_{acc}OP_m(pcat_1) &= Y_{acc}OP_m(pcat_1, ls) \\ &\subset YREG = Y_{aut}OP_m(pcat_1) = Y_{aut}OP_m(pcat_1, ls). \end{aligned}$$

## Controlled P Systems and Time-Varying P Systems

Another method to control the application of the labeled rules is to use control languages (see [8] and [1]). A *controlled P system* is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, H, W, f)$$

where  $\Pi' = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$  is a P system as defined above,  $H$  is a set of labels for the rules in the sets  $R_1, \dots, R_m$ , and  $W$  is a string language over  $2^H$  from a family  $FL$ . Every successful computation in  $\Pi$  has to follow a control word  $U_1 \dots U_n \in W$ : in transition step  $i$ , only rules with labels in  $U_i$  are allowed to be applied, and after the  $n$ -th transition, the computation halts; we may relax this end condition, and then we speak of *weakly controlled P systems*. If  $W = (U_1 \dots U_p)^*$ ,  $\Pi$  is called a *(weakly) time-varying P system*: in the computation step  $pn + i$ ,  $n \geq 0$ , rules from the set  $U_i$  have to be applied;  $p$  is called the *period*. The family of sets  $Y(\Pi)$ ,  $Y \in \{N, Ps\}$ , computed by (weakly) controlled P systems and (weakly) time-varying P systems with period  $p$ , with at most  $m$  membranes and rules of type  $X$  as well as control languages in  $FL$  is denoted by  $YOP_m(X, C(FL))$  ( $YOP_m(X, wC(FL))$ ) and  $YOP_m(X, TV_p)$  ( $YOP_m(X, wTV_p)$ ), respectively.

## Computational Completeness Results

The computational completeness results for catalytic P systems were shown in [7], the results for purely catalytic P systems in [3], and the results for P automata and accepting P systems were established in [6].

**Theorem 2.** For  $Y \in \{N, Ps\}$ ,  $\delta \in \{acc, aut, gen\}$ ,

$$Y_{\delta}OP_1(pcat_2, ls) = Y_{\delta}OP_1(cat_1, ls) = YRE.$$

**Theorem 3.** For any  $Y \in \{N, Ps\}$  and  $\alpha \in \{\lambda, w\}$ ,  $\delta \in \{aut, gen\}$ ,

$$Y_{\delta}OP_1(cat_1, \alpha TV_6) = Y_{\delta}OP_1(pcat_2, \alpha TV_6) = YRE.$$

For time-varying accepting P systems, the number of catalysts depends on the dimension  $d$  of the input vectors as already observed for the case without any control mechanism in [4]:

**Theorem 4.** For any  $d \geq 1$  and any  $\alpha \in \{\lambda, w\}$ ,

$$Ps_{acc}OP_1(cat_1, \alpha TV_{3(d+2)}) = Ps_{acc}OP_1(pcat_2, \alpha TV_{3(d+2)}) = N^d RE.$$

## References

1. A. Alhazov, R. Freund, H. Heikenwälder, M. Oswald, Yu. Rogozhin, S. Verlan, Sequential P systems with regular control. In: E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (Eds.): *Membrane Computing - 13th International Conference, CMC 2012*, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, LNCS **7762**, Springer, 2013, 112–127.
2. E. Csuhaj-Varjú, Gy. Vaszil: P automata or purely communicating accepting P systems. In: Gh. Păun, G. Rozenberg, A. Salomaa, C. Zandron (Eds.): *Membrane Computing, International Workshop, WMC-CdsA 2002*, Curtea de Argeş, Romania, August 2002, LNCS **2597**, Springer, 2003, 219–233.
3. R. Freund: Purely catalytic P systems: two catalysts can be sufficient for computational completeness, in *Proc. CMC14*.
4. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Theor. Comp. Sci.* **330**, 2005, 251–266.
5. R. Freund, M. Oswald: A short note on analysing P systems. *Bulletin of the EATCS* **78** (October 2002), 231–236.
6. R. Freund, M. Oswald: Catalytic and purely catalytic P Automata: Control mechanisms for obtaining computational completeness, to appear in *Proc. NCMA 2013*.
7. R. Freund, Gh. Păun: How to obtain universality in P systems with one catalyst, to appear in *Proc. MCU 2013*.
8. K. Krithivasan, Gh. Păun, A. Ramanujan: On controlled P systems. *Fundamenta Informaticae*, to appear.
9. M. Oswald: P Automata. PhD dissertation. Vienna University of Technology, 2003.
10. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.* **61** (2000), 108–143 (see also TUCS Report 208, November 1998, [www.tucs.fi](http://www.tucs.fi)).
11. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
12. P. Sosík, J. Matyšek: Membrane computing: when communication is enough. In: C.S. Calude, M.J. Dinneen, F. Peper (Eds.): *Unconventional Models of Computation 2002*, LNCS **2509**, Springer, 2002, 264–275.
13. The P Systems Website: <http://ppage.psystems.eu>.



# Computational Completeness with Generating and Accepting P Systems Using Minimal Left and Right Insertion and Deletion

Rudolf Freund<sup>1</sup>, Yurii Rogozhin<sup>2</sup>, and Sergey Verlan<sup>3</sup>

<sup>1</sup> Faculty of Informatics, Vienna University of Technology  
Favoritenstr. 9, 1040 Vienna, Austria  
Email: [rudi@emcc.at](mailto:rudi@emcc.at)

<sup>2</sup> Institute of Mathematics and Computer Science  
Academy of Sciences of Moldova  
Str. Academiei 5, Chişinău, MD-2028, Moldova  
Email: [rogozhin@math.md](mailto:rogozhin@math.md)

<sup>3</sup> LACL, Département Informatique, Université Paris Est  
61, av. Général de Gaulle, 94010 Créteil, France  
Email: [verlan@univ-paris12.fr](mailto:verlan@univ-paris12.fr)

The *insertion* operation was first considered in [4, 5] and after that related insertion and *deletion* operations were investigated, e.g., in [6, 11]. If the length of the contexts and/or of the inserted and deleted strings are big enough, then the insertion-deletion closure of a finite language leads to computational completeness, e.g., see [11] for an overview of this area. For specific references of using biologically motivated point mutations – insertions, deletions, and substitutions – in the area of molecular computing we refer to [2] and [3]; the results described in this note were established in these papers.

## Grammars, Post Rewriting Rules, Point Mutations

A (*string rewriting*) grammar  $G$  of type  $X$  is a construct  $(V, T, A, P)$  where  $V$  is a (finite) set of *symbols*,  $T \subseteq V$  is a set of *terminal symbols*,  $A \in V^*$  is the *axiom*, and  $P$  is a finite set of *rules* of type  $X$ .

$L(G) = \{v \in T^* \mid A \xRightarrow{*}_G v\}$  is the *language generated by  $G$* , and  $L_a(G) = \{v \in T^* \mid v \xRightarrow{*}_G A\}$  is the *language accepted by  $G$* . The family of languages generated (accepted) by grammars of type  $X$  is denoted by  $\mathcal{L}(X)$  ( $\mathcal{L}_a(X)$ ). If instead of a single axiom  $A$  we allow a finite set of axioms, we put an  $A$  in front of the type  $X$  for this variant of grammars, thus obtaining the family of languages generated (accepted) by grammars of type  $X$  denoted by  $\mathcal{L}(A-X)$  ( $\mathcal{L}_a(A-X)$ ).

We here consider string rewriting rules only working at the ends of a string (they can be seen as restricted variants of Post rewriting rules as already introduced by Emil Post in [9]):

**Simple Post rewriting rule**  $P[u\$x/y\$v]$  with  $u, x, y, v \in V^*$ :

$$P[u\$x/y\$v](uwv) = ywv \text{ for } w \in V^*.$$

**Normal Post rewriting rule**  $P[x/y]$  with  $x, y \in V^*$ :  $P[x/y](wx) = yw$  for  $w \in V^*$ .

**Left substitution**  $S_L[u/y]$  with  $u, y \in V^*$ :  $S_L[u/y](uw) = yw$  for  $w \in V^*$ .

**Right substitution**  $S_R[x/v]$  with  $x, v \in V^*$ :  $S_R[x/v](wx) = wxv$  for  $w \in V^*$ .

For a type of grammars using only substitution rules  $S_L[x/y]$  with  $|x| \leq k$  and  $|y| \leq m$ , we write  $S_L^{k,m}$ ; in the same way, we define the types  $S_R^{k,m}$ ,  $I_L^m$ ,  $I_R^m$ ,  $D_L^k$ , and  $D_R^k$ . The type  $D^k I^m S^{k'm'}$  allows for the deletion of strings with length  $\leq k$ , for the insertion of strings with length  $\leq m$ , and for the substitution of strings of length  $\leq k'$  by strings of length  $\leq m'$  on either side of a string.

*Example 1.* Let  $G = (V, T, A, P)$  be a regular grammar, i.e., the rules in  $P$  are of the form  $A \rightarrow bC$  and  $A \rightarrow \lambda$  with  $A, C \in V \setminus T$  and  $b \in T$ . Then the grammar  $G' = (V, T, A, \{S_R[A/y] \mid A \rightarrow y \in P\})$  with substitution rules generates the same language as  $G$ , i.e.,  $L(G') = L(G)$ . Hence, with  $REG$  denoting the family of regular languages, we obviously have got  $REG \subseteq \mathcal{L}(S_R^{1,2})$ .

**Theorem 1.** Every language  $L \subseteq T^*$  in  $\mathcal{L}(D^1 I^1 S^{1,1})$  can be written in the form  $T_l^* S T_r^*$  where  $T_l, T_r \subseteq T$  and  $S$  is a finite subset of  $T^*$ .

**Corollary 1.**  $\mathcal{L}(A-D^1 I^1 S^{1,1}) = \mathcal{L}(A-I^1) = \mathcal{L}_a(A-D^1 I^1 S^{1,1}) = \mathcal{L}_a(A-D^1) \subseteq REG$ .

## Post Systems

A *Post system*  $(V, T, A, P)$  is a grammar of *type SPS (NPS)* using only simple (normal) Post rewriting rules; it is said to be in *normal form* (of *type PSNF*) if and only if the Post rewriting rules  $P[x/y]$  in  $P$  are only of the forms  $P[ab/c]$ ,  $P[a/bc]$ ,  $P[a/b]$ , and  $P[a/\lambda]$ , with  $a, b, c \in V$ , and to be in *Z-normal form* (of *type PSZNF*) if and only if it is in normal form,  $A \in V \setminus T$ , and, moreover, there exists a special symbol  $Z \in V \setminus T$  such that

- $Z$  appears only once in the string  $x$  of a Post rewriting rule  $P[x/y]$ , and this rule is  $P[Z/\lambda]$ ;
- if the rule  $P[Z/\lambda]$  is applied, the derivation in the Post system stops yielding a terminal string;
- a terminal string can only be obtained by applying the rule  $P[Z/\lambda]$ .

The results established in [2] and [3] and presented in this note are based on results for Post systems being folklore since many years, e.g., see [7] and [1]:

**Theorem 2.** For every recursively enumerable language  $L \subseteq T^*$  there exists

- a Post rewriting system  $G$  in *Z-normal form* generating  $L$ , i.e.,  $\mathcal{L}(SPS) = \mathcal{L}(PSNF) = \mathcal{L}(PSZNF) = RE$ ;
- a Post rewriting system  $G'$  in *normal form* accepting  $L \setminus \{Z\}$ ,  $Z \notin T$ .

## (Sequential) P Systems

For the the area of P systems, we refer the reader to [8] and the P page [10].

A *(sequential) P system of type X with tree height n* is a construct  $\Pi = (G, \mu, R, i_0)$  where  $G$  is a sequential string grammar of type  $X$ ;  $\mu$  is the membrane (tree) structure of the system with the height of the tree being  $n$ , the membranes are uniquely labelled by labels from a set  $Lab$ ;  $R$  is a set of rules of the form  $(h, r, tar)$  where  $h \in Lab$ ,  $r \in P$ , and  $tar$ , called the *target indicator*, is taken from the set  $\{here, in, out\} \cup \{in_h \mid h \in Lab\}$ ;  $i_0$  is the initial membrane containing the axiom  $A$ .

A configuration of  $\Pi$  is a pair  $(w, h)$  where  $w$  is the current object (e.g., string or array) and  $h$  is the label of the membrane currently containing the object  $w$ . A sequence of transitions between configurations of  $\Pi$ , starting from the initial configuration  $(A, i_0)$ , is called a *computation* of  $\Pi$ . A *halting computation* is a computation ending with a configuration  $(w, h)$  such that no rule from  $R_h$  can be applied to  $w$  anymore;  $w$  is called the *result* of this halting computation if  $w \in O_T$ . The language generated by  $\Pi$ ,  $L_t(\Pi)$ , consists of all terminal objects from  $O_T$  being results of a halting computation in  $\Pi$ .

By  $\mathcal{L}(X-LP)$  ( $\mathcal{L}(X-LP^{(n)})$ ) we denote the family of languages generated by P systems (of tree height at most  $n$ ) using rules of type  $X$ . If only the targets *here*, *in*, *out* are used, then the P system is called *simple*, and the families of languages are denoted by  $\mathcal{L}(X-LsP)$  ( $\mathcal{L}(X-LsP^{(n)})$ ). If even only the targets *in* and *out* are used, then the P system is called a *channel type P system*, as any change taking place in such a P system can be interpreted as only happening when an object (a string) passes through a membrane; the corresponding families of languages are denoted by  $\mathcal{L}(X-LcP)$  ( $\mathcal{L}(X-LcP^{(n)})$ ).

For the accepting case,  $i_0$  is the initial membrane where the axiom  $A$  together with the input  $w \in T^*$  is put as  $wA$  at the beginning of a computation, and the input  $w$  is accepted if and only if there exists a halting computation from the initial configuration  $(wA, i_0)$ . By  $\mathcal{L}_a(X-LP)$ ,  $\mathcal{L}_a(X-LsP)$ , and  $\mathcal{L}_a(X-LcP)$  ( $\mathcal{L}_a(X-LP^{(n)})$ ,  $\mathcal{L}_a(X-LsP^{(n)})$ , and  $\mathcal{L}_a(X-LcP^{(n)})$ ) we then denote the families of languages accepted by P systems, simple P systems, and channel type P systems (of tree height at most  $n$ ) using rules of type  $X$ .

*Example 2.* Let  $\Pi = (G, [_1 [_2 ]_2 [_3 ]_3]_1, R, 1)$  be a P system of type  $I_R^1 I_L^1$  with

$$\begin{aligned} G &= (\{a, b\}, \{a, b\}, a, \{I_R[b], I_L[a]\}), \\ R &= \{(1, I_R[b], in), (2, I_L[a], out)\} \end{aligned}$$

Then  $\Pi$  generates the non-regular language  $\{a^{n+1}b^{n+1} \mid n \geq 0\}$ , which is also accepted by the P system  $\Pi' = (G, [_1 [_2 ]_2 [_3 [_4 [_5 ]_4]_3]_1, R, 2)$  of type  $D_R^1 D_L^1 I_R^1$  with

$$\begin{aligned} G &= (\{a, b\}, \{a, b\}, \lambda, \{D_L[a], D_R[b], I_R[\#]\}), \\ R &= \{(2, D_L[a], out), (1, D_R[b], in), (3, D_R[a], in), (3, D_R[b], in)\} \\ &\quad \cup \{(1, I_R[\#], in), (2, I_R[\#], out), (3, D_R[\#], in)\} \\ &\quad \cup \{(4, I_R[\#], in), (5, D_R[\#], out)\}. \end{aligned}$$

## Computational Completeness of P Systems with Minimal Insertion, Deletion, and Substitution Rules

The results  $RE = \mathcal{L}(D_R^1 I_L^1 S_R^{1,1} - LP^{(1)}) = \mathcal{L}(D^1 I^1 - LsP^{(8)})$  were already established in [2], the remaining results were obtained in [3].

**Theorem 3.**  $RE = \mathcal{L}(D_R^1 I_L^1 S_R^{1,1} - LP^{(1)}) = \mathcal{L}_a(D_R^1 I_L^1 S_R^{1,1} - LP^{(1)})$ .

If we want to restrict ourselves to the simple targets *here, in, out* or even to the targets *in* and *out* as well to use only minimal insertions and deletions, then we need P systems with larger tree height (and also much more difficult proof techniques):

**Theorem 4.**  $\mathcal{L}(D^1 I^1 - LsP^{(8)}) = \mathcal{L}_a(D^1 I^1 - LsP^{(8)}) = RE$ .

**Theorem 5.**  $\mathcal{L}(D^1 I^1 - LcP^{(8)}) = \mathcal{L}_a(D^1 I^1 - LcP^{(8)}) = RE$ .

The P systems constructed in the proofs for the theorems stated above have rather large tree height; it remains an open question to reduce this complexity parameter.

## References

1. R. Freund, M. Oswald, A. Păun: Gemmating P systems are computationally complete with four membranes. In: L. Ilie, D. Wotschke (eds.): *Pre-proceedings DCFs 2004*. The University of Western Ontario, Rep. No. 619, 191–203 (2004).
2. R. Freund, Yu. Rogozhin, S. Verlan: P systems with minimal left and right insertion and deletion. In: J. Durand-Lose, N. Jonoska (eds.): *Unconventional Computation and Natural Computation, 11th International Conference, UCNC 2012*. Orleans, France, September 3–7, 2012. Lecture Notes in Computer Science **7445**, 82–93, Springer (2012).
3. R. Freund, Yu. Rogozhin, S. Verlan: Generating and accepting P systems with minimal left and right insertion and deletion. To appear in *Natural Computing*.
4. B. Galiukschov: Semicontextual grammars. *Logica i Matem. Lingvistika*, 38–50. Tallin University (in Russian) (1981).
5. D. Haussler: *Insertion and Iterated Insertion as Operations on Formal Languages*. PhD thesis, Univ. of Colorado at Boulder, 1982.
6. L. Kari: *On Insertion and Deletion in Formal Languages*. PhD thesis, University of Turku, 1991.
7. M.L. Minsky: *Computation: Finite and Infinite Machines*. Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
8. Gh. Păun, G. Rozenberg, A. Salomaa: *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
9. E.L. Post: Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics* **65** (2), 197–215 (1943).
10. The P systems Web page: <http://ppage.psyste.ms.eu/>
11. S. Verlan: *Study of Language-Theoretic Computational Paradigms Inspired by Biology*. Habilitation thesis, University of Paris Est, 2010.

# A Quantum Inspired UREM P System for Solving a Linguistic Problem

Alberto Leporati<sup>1</sup> and Lyudmila Burtseva<sup>2</sup>

<sup>1</sup> Dipartimento di Informatica, Sistemistica e Comunicazione  
Università degli Studi di Milano-Bicocca  
Viale Sarca 336/14, 20126 Milano, Italy  
`alberto.leporati@unimib.it`

<sup>2</sup> Institute of Mathematics and Computer Science  
5, Academiei street, Chisinau, Republic of Moldova, MD 2028  
`luburtseva@gmail.com`

**Abstract.** We illustrate a quantum-inspired solution to the following dictionary search problem: given as input a word from a predefined natural language (for example, English), produce its translation to another predefined natural language (in our example, Romanian). In our solution, words are represented as vectors in an appropriate finite-dimensional Hilbert space. Such a simple representation opens up the possibility to solve several classes of problems from computational linguistics by quantum computations.

**Keywords:** unconventional computation, quantum computation, computational linguistics, quantum-inspired UREM P system, quantum linguistic string.

## 1 Introduction

Problems of Natural Language Processing (NLP) are known to consume many resources. Search for solutions of such classes of problems usually inspire the employment of new techniques, methods and even computational devices. Hence, during the last decade, NLP problems have become widespread practical applications for unconventional computation. In the late 2000s, our groups of researchers, dealing with quantum and bio-inspired (more specifically, with **P systems**) computational models, joined their efforts. After proposing to use a hybrid bio-quantum model for NLP [1], we now concentrate on selecting problems whose solutions look promising for the new formalism. In particular, narrowing the domain of NLP to computational linguistics, we chose the problem of dictionary search, which simply consists of producing a string which is the translation (to a predefined human language) of the string given as input.

The solution to this problem by **P systems** was proposed in [2], and works as follows. The P system operates on strings of symbols, where each symbol is a lowercase letter from the English alphabet. The P system consists of a skin, containing 26 subsystems — one for each letter of the alphabet. The string given as

input is put into the region enclosed by the skin membrane. The system removes the first character of the string and, based on its value, sends the remaining substring to the subsystem labeled by the corresponding character. The same process then occurs into such subsystem, which has the same structure as the skin membrane: the second character of the string is removed and the remaining substring is sent into one of the subsystems at the next level down the hierarchy. In the last step, when the last character of the string has been analyzed, the system produces a new string — which is the translation of the word represented by the string given as input — and sends it to the environment as output.

In this extended abstract we propose a quantum version of this process, based on quantum-inspired UREM P systems [3]. Despite the fact that quantum computations have a wide range of features suitable for solving NLP tasks, the application of such features are still under consideration by researchers. The main stumbling block is the problem of giving a quantum representation of linguistics elements, especially the main one — orthographic strings. Several approaches to this problem have been proposed since 2009, but even more recent solutions assume strong restrictions on the processed subsets of the natural language. In this work we propose to represent **linguistic strings** as vectors of an appropriate finite-dimensional Hilbert space. Such a simple representation opens up the possibility to solve several problems of computational linguistics by quantum computations.

## 2 The proposed approach

Our quantum version of the dictionary search process is based on quantum-inspired UREM P systems, which have been introduced in [3]. Due to the lack of space, we refer the reader to [3] for their precise definition as well as the details on how they work. Here we just recall that each symbol is a quantum system which assumes its states in a finite-dimensional Hilbert space, whereas each membrane has a truncated harmonic oscillator associated to it, representing its status. Interactions between objects and membranes occur by means of linear (in general, non-unitary) operators, realizing projectors between quantum base states. Transformations between superpositions of base states occur by linearity.

Let  $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}, \dots, \mathbf{z}, sp\}$  be the alphabet containing the 26 lowercase latin letters plus the symbol  $sp$ , indicating a space. Let  $L_d = \left\{0, \frac{1}{d-1}, \frac{2}{d-1}, \dots, \frac{d-2}{d-1}, 1\right\}$  be the set of base states of the Hilbert space  $\mathbb{C}^d$ . We can represent each letter of a string by a quantum system that is able to assume one of the 27 values contained in the basis  $\mathcal{B}_{27} = \{|x\rangle \mid x \in L_{27}\}$  of the Hilbert space  $\mathbb{C}^{27}$ ; so, for example, letter  $\mathbf{e}$  is represented as  $\left|\frac{4}{26}\right\rangle$ , also written as  $|\mathbf{e}\rangle$  in the following. If we denote by  $n$  the maximum length of the strings we are dealing with, then each string that the system will process can be represented as a vector  $|c_1, \dots, c_n\rangle$ , where each  $c_i$ , with  $1 \leq i \leq n$ , is a quantum system representing a character. Such a vector is an element of the Hilbert space  $\otimes^n \mathbb{C}^{27}$ , whose basis is the set  $\mathcal{B}_{27}^n = \{|x_1, \dots, x_n\rangle \mid x_i \in L_{27} \text{ for all } 1 \leq i \leq n\}$ . If the given string has a length

less than  $n$ , we pad it with spaces; so, for example, the string **hello** will be represented as the vector  $|h, e, l, l, o, sp, \dots, sp\rangle$  of length  $n$ .

Now we can use projection operators to transform letters to letters. Precisely, the operator  $E_{x,y} = |y\rangle\langle x|$ , where  $x, y \in L_{27}$ , works as follows:  $E_{x,y}|x\rangle = |y\rangle$ , whereas  $E_{x,y}|z\rangle = \mathbf{0}$  for all  $z \in L_{27} \setminus \{x\}$ . It is known that linear (and non-unitary) operators  $E_{x,y}$ , for any choice of  $x, y \in L_{27}$  can be realized as a composition of *creation* and *annihilation* operators working on harmonic oscillators, as described in [3]. If we apply  $E_{x,y}$  to an argument  $|z\rangle$  which is a superposition of vectors from the basis  $\mathcal{B}_{27}$  then the result is obtained by linearity. For example, let  $|z\rangle = \alpha|x\rangle + \beta|t\rangle$ , with  $x \neq t$  and  $x, t \in L_{27}$ ; then  $E_{x,y}|z\rangle = E_{x,y}(\alpha|x\rangle + \beta|t\rangle) = \alpha E_{x,y}|x\rangle + \beta E_{x,y}|t\rangle = \alpha|y\rangle + \beta\mathbf{0} = \alpha|y\rangle$ .

Working on tensor products of the Hilbert space  $\mathbb{C}^{27}$ , we can also transform strings into strings. So, given two vectors  $|v\rangle, |w\rangle \in \mathcal{B}_{27}^n$  representing two words as described above, we can build the projection operator  $E_{v,w} = |w\rangle\langle v|$  that transforms  $|v\rangle$  into  $|w\rangle$  as follows. Let  $|v\rangle = |v_1, \dots, v_n\rangle$  and  $|w\rangle = |w_1, \dots, w_n\rangle$ . The operator  $E_{v,w}$  can be decomposed as  $E_{v,w} = E_{v_1,w_1} \otimes \dots \otimes E_{v_n,w_n}$ , so that given any  $|z\rangle = |z_1, \dots, z_n\rangle = |z_1\rangle \otimes \dots \otimes |z_n\rangle \in \mathcal{B}_{27}^n$  we have:

$$\begin{aligned} E_{v,w}|z\rangle &= (E_{v_1,w_1} \otimes \dots \otimes E_{v_n,w_n})(|z_1\rangle \otimes \dots \otimes |z_n\rangle) \\ &= E_{v_1,w_1}|z_1\rangle \otimes \dots \otimes E_{v_n,w_n}|z_n\rangle \end{aligned} \quad (1)$$

From this decomposition it is clear that vector  $|z\rangle$  is transformed to vector  $|w\rangle$  if and only if  $|z\rangle$  coincides with  $|v\rangle$ , character by character:  $z_i = v_i$  for all  $1 \leq i \leq n$ . In fact, let  $z_i \neq v_i$  for some  $1 \leq i \leq n$ . Then, applying the operator  $E_{v_i,w_i}$  on  $|z_i\rangle$  produces the null vector of  $\mathbb{C}^{27}$ , and hence the null vector of  $\otimes^n \mathbb{C}^{27}$  is obtained as a result in (1). We can thus conclude that  $E_{v,w}$  operates as follows on the vectors of  $\mathcal{B}_{27}^n$ :  $E_{v,w}|v\rangle = |w\rangle$ , whereas  $E_{v,w}|z\rangle = \mathbf{0}$  for all  $|z\rangle \in \mathcal{B}_{27}^n \setminus \{v\}$ . Once again,  $E_{v,w}$  operates by linearity on vectors  $|z\rangle$  which are superpositions of base vectors from  $\mathcal{B}_{27}^n$ . Each operator  $E_{v,w}$ , for any  $|v\rangle, |w\rangle \in \mathcal{B}_{27}^n$ , can be realized as an appropriate composition of creation, annihilation operators and tensor products, as described in [3].

A *dictionary* can be implemented by summing appropriate operators  $E_{v,w}$ . For example, assume that we want to translate the word **bat** to **liliac** and the word **milk** to **lapte**. Moreover, for simplicity assume that  $n = 8$ . Then, the operator  $\mathcal{D}$  corresponding to our dictionary is:

$$\begin{aligned} \mathcal{D} &= E_{bat,liliac} + E_{milk,lapte} \\ &= |1, i, l, i, a, c, sp, sp\rangle\langle b, a, t, sp, sp, sp, sp, sp| + \\ &\quad |1, a, p, t, e, sp, sp, sp\rangle\langle m, i, l, k, sp, sp, sp, sp| \end{aligned}$$

Once again, this operator can be realized using only creation and annihilation operators, together with tensor products and sums, as discussed in [3].

### 3 Discussion

Although being theoretically perfect, the idea of using the tensor product to represent words brings with it the so-called “curse of dimensionality”. In particular,

considering the approach closest to our one [4], that implements linguistics retrieval through quantum vector spaces and projectors, we found orderly quantum schemes supplied by simplified examples.

In our case, the problem lies with the size of the linear matrix describing the operator  $\mathcal{D}$ . Assuming  $n = 100$  for a given natural language, and an alphabet of 27 symbols, the order of the square linear matrix describing  $\mathcal{D}$  would be  $27^{100}$ , a huge number that makes impractical the physical realization of  $\mathcal{D}$ .

At the moment, we see two possibilities to escape from this situation. The first possibility uses a pure quantum approach and consists of identifying each word of the language we want to translate by a natural number, and encoding the translation as a mapping between natural numbers. So doing, given for example the word **bat**, we should first calculate the natural number associated to it, then apply the linear operator describing the mapping, and finally find the word (**lilic**) that corresponds to the resulting number. This, of course, seems to be more complicated than the original translation problem; anyway, in this case, the order of the linear matrix implementing the mapping between natural numbers would be equal to the number of words contained in our languages (usually no more than 500 thousands).

Another possibility lies in attempts to decompose the linear matrix as a tensor product of several smaller matrices. This can be done if and only if the action of  $\mathcal{D}$  can be decomposed as two or more independent actions on different (and separated) parts of the vectors given as input. Albeit we prefer this second solution, since the preparation and separation of inputs can be implemented both by pure quantum and hybrid bio-quantum approaches, we recognize that looking for the possibility of separating the action of  $\mathcal{D}$  into independent parts is in general a difficult task, requiring an impressive amount of pre-computation.

## References

1. A. Alhazov, L. Burtseva, S. Cojocaru, A. Colesnicov, L. Malahov, Yu. Rogozhin: Joining bio-inspired and quantum approaches in computer algebra and computer linguistics. In M. Gheorghe et al. (Eds.), *Pre-proceedings of the Twelfth Conference On Membrane Computing, CMC 12*, Fontainebleau, August 19-27, 2011, pp. 471-478. Available at <http://cmc12.lacl.ft/cmc12proceedings.pdf>
2. A. Alhazov, S. Cojocaru, L. Malahova, Y. Rogozhin: Dictionary Search and Update by P Systems with String-Objects and Active Membranes. *International Journal of Computers Communications & Control* 4(3):206-213, 2009.
3. A. Leporati, G. Mauri, C. Zandron: Quantum Sequential P Systems with Unit Rules and Energy Assigned to Membranes. In R. Freund et al. (Eds.), *Membrane Computing: 6<sup>th</sup> International Workshop, WMC 2005*, Revised Selected and Invited Papers, LNCS 3850, Springer, 2006, pp. 310-325.
4. P. Wittek, S. Darányi: Introducing scalable quantum approaches in language representation. In D. Song et al. (Eds.), *Quantum Interaction: 5<sup>th</sup> International Symposium, QI '11*, Revised Selected Papers, LNCS 7052, Springer, 2011, pp. 2-12.



# A Catalytic P System with Two Catalysts Generating a Non-Semilinear Set

Petr Sosík<sup>1,2</sup>

<sup>1</sup> Departamento de Inteligencia Artificial, Facultad de Informática,  
Universidad Politécnica de Madrid, Campus de Montegancedo s/n,  
Boadilla del Monte, 28660 Madrid, Spain,

<sup>2</sup> Research Institute of the IT4Innovations Centre of Excellence,  
Faculty of Philosophy and Science, Silesian University in Opava  
74601 Opava, Czech Republic, [petr.sosik@fpf.slu.cz](mailto:petr.sosik@fpf.slu.cz)

**Abstract.** Although it has been proven already in 2005 [1] that catalytic P systems with two catalysts are computationally universal, no simple example of such a P system generating a non-semilinear set was known. The present paper intends to fill this gap and provides such an example with 54 rules. It is expected, however, that this number of rules can be dramatically reduced and the minimal number of rules to generate a non-semilinear set in a catalytic P system with two catalysts remains open.

## 1 Main result

In this paper we use extended catalytic P systems as defined at the beginning of Chapter 4 in [2], where the terminal set of output objects is formed by all non-catalytic objects. No cooperative rules, bistable catalysts, priorities or similar enhancements are considered.

**Theorem 1.** *Extended catalytic P systems with a single membrane, two catalysts and 54 rules can generate non-semilinear sets of numbers.*

*Proof.* We construct an example of a catalytic P systems with two catalysts and 54 rules generating the set  $\{2^n - 1 \mid n \geq 1\}$ . This set is generated by a non-deterministic register machine with the following program, starting with both registers empty and with the result in register 1:

1: (ADD(1), 2, 7)  
2: (SUB(1), 3, 5)  
3: (ADD(2), 4, 4)  
4: (ADD(2), 2, 2)  
5: (SUB(2), 6, 1)  
6: (ADD(1), 5, 5)  
7: HALT

We construct a catalytic P system  $\Pi$  following precisely the construction in the proof of Corollary 8 in [1]. This construction uses a specific group of rules of a P system to implement each instruction of the register machine. However, we use the terminal set of objects  $O \setminus C$  to define the output as in [2], instead of another membrane used in [1]. Therefore we omit the rules in [1] which transport objects to membrane 2. The resulting catalytic P system simulates the non-deterministic register machine described above and it generates a representation of contents of its registers by the corresponding number of symbols  $o_1$  and  $o_2$ , respectively:

$$\begin{aligned}
\Pi &= (O, \{c_1, c_2\}, [1]_1, w, R, 1), \\
O &= \{\#\} \cup \{c_1, c'_1, c''_1, c_2, c'_2, c''_2\} \cup \{o_1, o_2\} \\
&\cup \{p_j, \tilde{p}_j, p'_j, p''_j, \bar{p}_j, \bar{p}'_j, \bar{p}''_j, \hat{p}_j, \hat{p}'_j, \hat{p}''_j \mid j = 2, 5\} \\
&\cup \{p_j, \tilde{p}_j \mid j = 1, 3, 4, 6, 7\}, \\
R &= \{x \rightarrow \# \mid x \in \{p_j, \tilde{p}_j, p'_j, p''_j, \bar{p}_j, \bar{p}'_j, \bar{p}''_j, \hat{p}_j, \hat{p}'_j, \hat{p}''_j \mid j = 2, 5\}\} \\
&\cup \{x \rightarrow \# \mid x \in \{c'_1, c''_1, c'_2, c''_2\}\} \cup \{\# \rightarrow \#\} \\
&\cup \{c_1 p_7 \rightarrow c_1, c_2 \tilde{p}_7 \rightarrow c_2\} \\
&\cup \{c_1 \tilde{p}_j \rightarrow c_1 \mid j = 1, 3, 4, 6\} \\
&\cup \{c_2 p_1 \rightarrow c_2 p_2 \tilde{p}_2 o_1, c_2 p_1 \rightarrow c_2 p_7 \tilde{p}_7 o_1, \\
&\quad c_2 p_3 \rightarrow c_2 p_4 \tilde{p}_4 o_2, c_2 p_4 \rightarrow c_2 p_2 \tilde{p}_2 o_2, c_2 p_6 \rightarrow c_2 p_5 \tilde{p}_5 o_1\} \\
&\cup \{c_r p_j \rightarrow c_r \hat{p}_j \hat{p}'_j, c_r p_j \rightarrow c_r \bar{p}_j \bar{p}'_j \bar{p}''_j, c_r o_r \rightarrow c_r c'_r, c_r c'_r \rightarrow c_r c''_r, \\
&\quad c_{3-r} c''_r \rightarrow c_{3-r}, c_r \hat{p}'_j \rightarrow c_r \#, c_{3-r} \hat{p}'_j \rightarrow c_{3-r} \hat{p}''_j, c_r \hat{p}''_j \rightarrow c_r p_k \tilde{p}_k, \\
&\quad c_r \bar{p}_j \rightarrow c_r, c_{3-r} \bar{p}''_j \rightarrow c_{3-r} p''_j, c_{3-r} p''_j \rightarrow c_{3-r} p'_j, \\
&\quad c_r p'_j \rightarrow c_r p_l \tilde{p}_l \mid (j, r, k, l) = (2, 1, 3, 5), (5, 2, 6, 1)\} \\
&\cup \{c_2 y \rightarrow c_2 \mid y \in \{\tilde{p}_2, \hat{p}_2, \bar{p}'_2\}\} \\
&\cup \{c_1 y \rightarrow c_1 \mid y \in \{\tilde{p}_5, \hat{p}_5, \bar{p}'_5\}\}, \\
w &= c_1 c_2 p_1 \tilde{p}_1.
\end{aligned}$$

The above construction is correct (provided that the proof of Corollary 8 in [1] is correct). The total number of rules is 62. However, the above construction is general and can be simplified in specific cases. For example, when a SUB instruction is followed in the program by one or more instructions ADD, the rules implementing ADD can be eliminated and substituted by an extended last rule of SUB which would add the necessary objects. Similarly, the rules implementing HALT can be substituted by a modification of rules of the previous instruction. Particularly, in our case we can:

1. (a) remove the rules  $c_1 \tilde{p}_3 \rightarrow c_1$ ,  $c_1 \tilde{p}_4 \rightarrow c_1$ ,  $c_2 p_3 \rightarrow c_2 p_4 \tilde{p}_4 o_2$ ,  $c_2 p_4 \rightarrow c_2 p_2 \tilde{p}_2 o_2$ , implementing instructions 3: (ADD(2), 4, 4), 4: (ADD(2), 2, 2), and
- (b) modify the rule  $c_1 \hat{p}_2'' \rightarrow c_1 p_3 \tilde{p}_3$  which is a part of implementation of 2: (SUB(1), 3, 5) to the form  $c_1 \hat{p}_2'' \rightarrow c_1 p_2 \tilde{p}_2 o_2 o_2$ ;

2. (a) remove the rules  $c_1\tilde{p}_6 \rightarrow c_1$ ,  $c_2p_6 \rightarrow c_2p_5\tilde{p}_5o_1$ , implementing instruction 6: (ADD(1), 5, 5), and
  - (b) modify the rule  $c_2\hat{p}_5'' \rightarrow c_2p_6\tilde{p}_6$  which is a part of implementation of 2: (SUB(2), 6, 1) to the form  $c_2\hat{p}_5'' \rightarrow c_2p_5\tilde{p}_5o_1$ ;
3. (a) remove the rules  $c_1p_7 \rightarrow c_1$ ,  $c_2\tilde{p}_7 \rightarrow c_2$ , implementing 7: HALT, and
  - (b) modify the rule  $c_2p_1 \rightarrow c_2p_7\tilde{p}_7o_1$  which is a part of implementation of 1: (ADD(1), 2, 7) to the form  $c_2p_1 \rightarrow c_2o_1$ .

These modifications allow to save eight rules (and some objects), getting us to the final number of 54 rules. The resulting P systems will have the form:

$$\begin{aligned}
 \Pi &= (O, \{c_1, c_2\}, [1]_1, w, R, 1), \\
 O &= \{\#\} \cup \{c_1, c_1', c_1'', c_2, c_2', c_2''\} \cup \{o_1, o_2\} \cup \{p_1, \tilde{p}_1\} \\
 &\quad \cup \{p_j, \tilde{p}_j, p_j', p_j'', \bar{p}_j, \bar{p}_j', \bar{p}_j'', \hat{p}_j, \hat{p}_j', \hat{p}_j'' \mid j = 2, 5\} \\
 R &= \{x \rightarrow \# \mid x \in \{p_j, \tilde{p}_j, p_j', p_j'', \bar{p}_j, \bar{p}_j', \bar{p}_j'', \hat{p}_j, \hat{p}_j', \hat{p}_j'' \mid j = 2, 5\}\} \\
 &\quad \cup \{x \rightarrow \# \mid x \in \{c_1', c_1'', c_2', c_2''\}\} \cup \{\# \rightarrow \#\} \\
 &\quad \cup \{c_1\tilde{p}_1 \rightarrow c_1, c_2p_1 \rightarrow c_2p_2\tilde{p}_2o_1, c_2p_1 \rightarrow c_2o_1\} \\
 &\quad \cup \{c_rp_j \rightarrow c_r\hat{p}_j\hat{p}_j', c_rp_j \rightarrow c_r\bar{p}_j\bar{p}_j'\bar{p}_j'', c_ro_r \rightarrow c_rc_r', c_rc_r' \rightarrow c_rc_r'', \\
 &\quad \quad c_{3-r}c_r'' \rightarrow c_{3-r}, c_r\hat{p}_j' \rightarrow c_r\#, c_{3-r}\hat{p}_j' \rightarrow c_{3-r}\hat{p}_j'', \\
 &\quad \quad c_r\bar{p}_j \rightarrow c_r, c_{3-r}\bar{p}_j'' \rightarrow c_{3-r}p_j'', c_{3-r}p_j'' \rightarrow c_{3-r}p_j', \\
 &\quad \quad c_rp_j' \rightarrow c_r p_l \tilde{p}_l \mid (j, r, l) = (2, 1, 5), (5, 2, 1)\} \\
 &\quad \cup \{c_1\hat{p}_2'' \rightarrow c_1p_2\tilde{p}_2o_2o_2, c_2\hat{p}_5'' \rightarrow c_2p_5\tilde{p}_5o_1\} \\
 &\quad \cup \{c_2y \rightarrow c_2 \mid y \in \{\tilde{p}_2, \hat{p}_2, \bar{p}_2'\}\} \\
 &\quad \cup \{c_1y \rightarrow c_1 \mid y \in \{\tilde{p}_5, \hat{p}_5, \bar{p}_5'\}\}, \\
 w &= c_1c_2p_1\tilde{p}_1.
 \end{aligned}$$

## 2 Conclusion

We have shown that an extended catalytic P systems with a single membrane, two catalysts and 54 rules can generate non-semilinear sets of numbers. However, this result is barely optimal in terms of simplicity of the P system, particularly the minimal number of necessary rules.

For example, we conjecture that a significant simplification of implementation of instructions SUB can be achieved using the fact that for each register there exists only one instruction SUB decrementing it. Another promising way is to abandon the constructions used in [1, 2] and to construct a catalytic P system generating a non-semilinear set directly “from the scratch.”

## Acknowledgements

This work was supported by the Ministerio de Ciencia e Innovación, Spain, under project TIN2012-36992, by the European Regional Development Fund in

the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), and by the Silesian University in Opava under the Student Funding Scheme, project SGS/7/2011.

## References

1. Freund, R., Kari, L., Oswald, M., Sosík, P.: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science* 330, 251–266 (2005)
2. Păun, G., Rozenberg, G., Salomaa, A. (eds.): *The Oxford Handbook of Membrane Computing*. Oxford University Press, Oxford (2010)

# Author Index

- Adorna H. N., 181  
Alhazov A., 41  
Aman B., 49, 299  
  
Banu-Demergian I.T., 63  
Burtseva L., 325  
  
Cienciala L., 81  
Ciencialová L., 81  
Ciobanu A., 95  
Ciobanu G., 49, 299, 303  
Cojocaru S., 41  
Colesnicov A., 41  
Csajbók Z.E., 241  
Csuhaj-Varjú E., 117  
  
Dragomir C., 131  
  
Fésüs M., 313  
Fernau H., 309  
Freund R., 153, 309, 317, 321  
  
Gazdag Zs., 167  
Gheorghe M., 9  
Gruska J., 11  
  
Hernandez N.H. S., 181  
  
Ipate F., 9, 95, 131, 255  
Ivanov S., 199, 309  
  
Juayong R.A. B., 181  
  
Konur S., 131  
  
Langer M., 81  
Lefticaru R., 131  
Leporati A., 15, 213, 225, 325  
  
Malahov L., 41  
Manzoni L., 213  
Mauri G., 225  
Mierla L., 131  
Mihálydeák T., 241  
  
Nicolescu R., 255  
  
Obtułowicz A., 277  
Oswald M., 309, 317  
  
Pérez-Jiménez M.J., 283  
Păun Gh., 25, 317  
Petic M., 41  
Porreca A.E., 213, 225  
  
Riscos-Núñez A., 283  
Rius-Font M., 283  
Rogozhin Yu., 321  
  
Sburlan D., 303  
Schmid M.L., 309  
Sosík P., 35, 329  
Stefanescu G., 63  
Subramanian K.G., 309  
  
Takács P., 241  
  
Valencia-Cabrera L., 283  
Vaszil G., 117, 313  
Verlan S., 37, 199, 321  
  
Wu H., 255  
  
Zandron C., 225