Proceedings of the 17th International Conference on Membrane Computing (CMC17)

25-29 July, 2016

Milan (Italy)

Edited by Alberto Leporati and Claudio Zandron

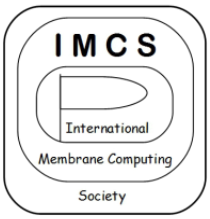
UNIVERSITÀ DEGLI STUDI DI MILANO BICOCCA

DISCO DIPARTIMENTO DI INFORMATICA SISTEMISTICA E COMUNICAZIONE

IMCS International Membrane Computing Society

Proceedings of the

Seventeenth International Conference on Membrane Computing (CMC17)



25-29 July, 2016

Milan, Italy

Alberto Leporati

Claudio Zandron

Editors

Proceedings of the 17th International Conference on Membrane Computing (CMC17)

Edited by: Alberto Leporati and Claudio Zandron

Published in July 2016

Preface

The present volume contains the invited contributions and a selection of papers presented at the 17th International Conference on Membrane Computing (CMC17), held in Milan, Italy, from July 25 to July 29, 2016. Further additional information on this conference can be found at the following website: *http://cmc17.disco.unimib.it*

The CMC series started with three workshops organized in Curtea de Argeş, Romania, in 2000, 2001 and 2002. The workshops were then held in Tarragona, Spain (2003), Milan, Italy (2004), Vienna, Austria (2005), Leiden, The Netherlands (2006), Thessaloniki, Greece (2007), and Edinburgh, UK (2008).

The 10th edition was organized again in Curtea de Argeş, in August 2009, where it was decided to continue the series as the Conference on Membrane Computing (CMC). The following editions were held in Jena, Germany (2010), Fontainebleau, France (2011), Budapest, Hungary (2012), Chişinău, Moldova (2013), Praha, Czech Republic (2014), and Valencia, Spain (2015).

CMC17 has been organized, under the auspices of the International Membrane Computing Society, and the European Molecular Computing Consortium (EMCC), by the Research Group On Molecular Computing of the Department of Informatics, Systems, and Communication, at the University of Milano-Bicocca.

CMC17 consisted of three different parts: the first day, representatives of research groups working on membrane computing presented recent research activities of the group, described the composition of the research team, and presented research networks and projects they are involved in. From Tuesday to Thursday the conference continued with standard sessions; invited lectures were given by Matteo Cavaliere (Edinburgh, UK), Thomas Hinze (Cottbus, Germany), Paolo Milazzo (Pisa, Italy), and Agustín Riscos-Núñez (Sevilla, Spain). The last day of the conference was devoted to interaction between participants, to discuss open problems and propose new research topics.

The editors express their gratitude to the Program Committee, the invited speakers, the authors of the papers, the reviewers, and all the participants for their contributions to the success of CMC17.

The support of the Department of Informatics, Systems, and Communication of the University of Milano-Bicocca and the Prize for the Best Student Paper awards granted by Springer-Verlag are gratefully acknowledged.

July 2016

Alberto Leporati
Claudio Zandron

The Steering Committee of the CMC/ACMC series consists of

Henry Adorna (Quezon City, Philippines)
Artiom Alhazov (Chişinău, Moldova)
Bogdan Aman (Iaşi, Romania)
Matteo Cavaliere (Edinburgh, Scotland)
Erzsébet Csuhaj-Varjú (Budapest, Hungary)
Rudolf Freund (Wien, Austria)
Marian Gheorghe (Bradford, UK) – Honorary Member
Thomas Hinze (Cottbus, Germany)
Florentin Ipate (Bucharest, Romania)
Shankara N. Krishna (Bombay, India)
Alberto Leporati (Milan, Italy)
Taishin Y. Nishida (Toyama, Japan)
Linqiang Pan (Wuhan, China) – Co-chair
Gheorghe Păun (Bucharest, Romania) – Honorary Member
Mario J. Pérez-Jiménez (Sevilla, Spain)
Agustín Riscos-Núñez (Sevilla, Spain)
Petr Sosík (Opava, Czech Republic)
Kumbakonam Govindarajan Subramanian (Penang, Malaysia)
György Vaszil (Debrecen, Hungary)
Sergey Verlan (Paris, France)
Claudio Zandron (Milan, Italy) – Co-chair
Gexiang Zhang (Chengdu, China)

The Organizing Committee of CMC17 consists of

Alberto Leporati (Milan, Italy) – Co-chair
Luca Manzoni (Milan, Italy)
Antonio Enrico Porreca (Milan, Italy)
Claudio Zandron (Milan, Italy) – Co-chair

The Programme Committee of CMC17 consists of

Artiom Alhazov (Chişinău, Moldova)
Bogdan Aman (Iaşi, Romania)
Lucie Ciencialová (Opava, Czech Republic)
Erzsébet Csuhaj-Varjú (Budapest, Hungary)
Giuditta Franco (Verona, Italy)
Rudolf Freund (Wien, Austria)
Marian Gheorghe (Bradford, UK)
Thomas Hinze (Cottbus, Germany)
Florentin Ipate (Bucharest, Romania)
Shankara Narayanan Krishna (Bombay, India)
Alberto Leporati (Milan, Italy) – Co-chair
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)
Mario de Jesús Pérez-Jiménez (Sevilla, Spain)
Antonio E. Porreca (Milan, Italy)
Agustín Riscos-Núñez (Sevilla, Spain)
José M. Sempere (Valencia, Spain)
Petr Sosík (Opava, Czech Republic)
György Vaszil (Debrecen, Hungary)
Sergey Verlan (Paris, France)
Claudio Zandron (Milan, Italy) – Co-chair
Gexiang Zhang (Chengdu, Sichuan, China)

Table of Contents

Invited Papers

Regular Papers

Extedended Abstracts

Invited Papers

Eco-Evo Dynamics and the Role of Cellular Computing

Matteo Cavaliere

University of Edinburgh, UK
mcavali2@staffmail.ed.ac.uk

In the commons, where the sustainability of communities depends on the presence of public goods, individuals often rely on (surprisingly) simple strategies to decide whether to contribute to the shared resource, i.e., public good (at risk of exploitation by free-riders, i.e., cheaters). This issue is observed in natural scenarios across different scales including cellular populations [5, 3]. As discussed in [2] simple cellular strategies, that exhibit limited information processing can be adjusted to exploit certain environmental constraints to attain sustainable growth - resilient to the invasion of cellular cheaters. This is obtained by coupling cellular decision-making to the core structural factors characterizing the cellular community, such as the size of its constituent groups or the resource characteristics, i.e., the public good efficiency. A very simple type of cellular decision-making is (trivially) the permanent production of public good. Such strategy only works when its creation efficiently induces growth and the commons is structured in relatively small groups. However, a simple strategy that stochastically alternates between contribution and non-contribution enlarges the range of commons where its adoption leads to sustainability. Moreover, two opposite conditional strategies –in which a simple sensing mechanism is at work – are effective in two contrasting environmental situations: positive plasticity (i.e., contribute only when most individuals are contributing) works for low pubic good efficiency and small groups, negative plasticity (contribute only when it is strictly necessary, i.e., contribute when few individuals are contributing) does it for high public good efficiency and large groups. Therefore, simple cellular computations associated to limited information processing [6] are effective against cheaters due to the specific ecological structure where they are applied, an issue originally raised in different contexts by H. Simon [7]. Results have been originally obtained by considering a stylized computational public-good model (in which a finite population of agents (cells) is organized in groups where they are involved in a public-good game, and the supply of public good determines population density). The ability of cells to monitor the environment and decide their contribution is then modeled by using finite state automata. Recently, the results have been experimentally shown in natural bacterial systems [1] and in synthetic ones [4].

References

1. R.C. Allen, L. McNally, R. Popat, S.P Brown. Quorum Sensing Protects Bacterial Co-Operation from Exploitation by Cheats. *ISME Journal*, 2016.

2. M. Cavaliere, J.F. Poyatos. Plasticity Facilitates Sustainable Growth in the Commons. *J R Soc Interface*, 10, 2013.
3. H. Celiker, J. Gore. Cellular Cooperation: Insights from Microbes. *Trends in Cell Biology*, **23**, 9-15, 2013.
4. C. Moreno-Fenoll, M. Cavaliere, E. Martinez-Garcia, J.F. Poyatos. Exploitation by Cheaters Facilitates the Preservation of Essential Public Goods in Microbial Communities.*bioRxiv*:040964, 2016.
5. S.A. Levin. *Fragile Dominion: Complexity and the Commons*. Perseus, Mass, 1999.
6. T.J. Perkins, P.S. Swain. Strategies for Cellular Decision-Making. *Mol. Systems Biology* **5**, 326, 2009.
7. H. Simon. *Rational Choice and the Structure of the Environment, in Models of Man*. J. Wiley Ed., New York, 1957.

Coping with Dynamical Structures for Useful Applications of Membrane Computing

Thomas Hinze

Brandenburg University of Technology, Institute of Computer Science
Postfach 10 13 44, D-03013 Cottbus

Friedrich Schiller University Jena, Ernst-Abbe-Platz 1–4, D-07743 Jena

thomas.hinze@b-tu.de

Extended Abstract

1 Biological Information Processing Primarily Utilises Dynamical Structures at Different Levels

Coping with *dynamical structures* turns out to be both, a challenging task but also a crucial clue in understanding, fine-grained modelling, and utilisation of biological and biologically inspired information processing [8]. Principles of molecular computing are mainly based on modifiable spatial and topological arrangements in different forms, contexts, and scales ranging from nanoscopic surface shapes up to complex macroscopic behavioural patterns. From a composition-oriented point of view, we identify at least *four levels* in which dynamical structures essentially occur:

1. The **molecular level** comprises spatial grouping of atoms by chemical bonds forming macromolecules. Corresponding intramolecular structures of DNA, RNA, and proteins constitute their functionality as data carrier and storage medium along with a co-ordinated set of biochemical reaction schemes. *Cell signalling* gives an illustrative example. Here, external stimuli like hormones or environmental factors reach receptors at the outer face of a cell membrane. At its inner part, signalling proteins and second messengers (ligands) are released. By passing a signalling cascade, signalling proteins become activated by a specific combination of ligands residing at protein binding sites. This results in composition of a dedicated molecular structure acting as transcription factor which in turn can enter the cell nucleus and afterwards initiate a specific gene expression producing a cell response. Even without modification of chemical bonds, we can observe dynamical molecular structures in a functional context. Let us consider an *ion channel*: It mainly consists of a transmembrane protein incorporating a controllable gate. Cations accumulate close to the channel entry outside the cell. As far as enough cations are present, the gate temporarily opens for a short moment, and an amount of cations can pass the channel into the cell inducing a spiking signal. The

function of the underlying gate is based on a dynamically movable side chain inside the transmembrane protein. Ion channels can for instance act as temperature sensors producing a frequency-encoded oscillatory spiking signal [6].

2. The **level of reaction network modules** opens the next stage of dynamical structures. Chemical reaction schemes, particularly those found in living organisms, apper to represent *invisible* networks. Nevertheless, they represent the driving force in information processing by conducting state transitions from one molecular configuration to another one. The topology of a chemical reaction network is commonly treated as a static structure due to its highly conserved genetic blueprint. A corresponding network of densely interwoven reactions forms a *module*, an elementary unit also called *motif*. Structural dynamics becomes visible when studying the interplay of reaction network modules over time. It turns out that several modules merge or couple to each other temporarily while modular compounds can also dissolve, re-arrange, or re-assembly. Initiated by trigger signals, by perturbances, or simply by random, network re-compositions arise. *Photosynthesis* is representative for that: Depending on presence or absence of light, different reaction schemes are active. Light intensity toggles between several underlying reaction network topologies composed of a set of modules. Another obvious example is given by *mutation* or *recombination* of genetic DNA which results in modified reaction network structures. Also *infection* of a cell by a virus or *bacterial gene transfer* can have similar effects. When aiming at understanding of maintenance of life, dynamics of reaction network structures is essential.
3. Increasing within the hierarchy, the **level of membranes** characterises a new quality of dynamical structures. Membranes enable a *compartmentalisation* of spatial structures in which chemical reactions and transportation processes appear. Having in mind that membranes form a physical *boundary* and they offer a selective or general *permeability* for molecules at the same time, it becomes obvious that dynamics at this level can imply powerful features [11]. *Cell division* is probably the most popular example of a dynamical membrane structure. Following this line, the formation of *tissues*, *organs*, and finally multi-cellular *organisms* exhibits an amazing capability of self-organisation [2] and self-coordination [1]. For instance, the spatial derivation of cytokines manages the progress in *cellular differentiation*. *Exocytosis* as well as *endocytosis* in conjunction with membrane creation and dissolution provides molecular containers for directed transportation. Often, membrane structures need to be assembled in an optimal way in order to achive a certain functionality at its best. *Optimal placement* of branches and junctions within capillar blood vessels for adequate supply of neighboured cells keeping low the overall need of molecular resources is a typical outcome. More notably, using *neural plasticity* the process of (re)mapping the brain structure emerges.
4. The **higher-order organism or population level** marks the topmost instance of dynamical structures found in complex biological systems. Un-

equivocally, underlying rules, principles, or laws responsible for control and development of the corresponding structures are often hard to identify and sometimes prone to errors, misinterpretations, or incompleteness. State-of-the-art approaches attempt to identify the rules by comprehensive statistical analyses of huge amounts of experimentally observed data resulting from macroscopic behavioural patterns [10]. A simple example is a *predator-prey system* for instance consisting of a population of rabbits and a population of foxes sharing the same living area. Dependent on relevant parameter values like rabbit's birth rate, fox's death rate and feeding activity, the system exhibits various behavioural patterns like stable oscillation with variable periodicity, extinction of foxes, or exponential growth of involved populations. An opposite example is *symbiosis* of organisms or populations. Moreover, *swarms*, *colonies*, and *societies* create highly complex behavioural patterns incorporating fascinating properties like cooperation, altruism, cognition, consciousness, or *intelligence*. Many of these properties still lack a formal definition based on substantiated understanding.

2 Membrane Systems for Explicit Formalisation of Structural Dynamics at Different Levels

Due to its discrete nature composed of algebraic elements, membrane systems appear to be an ideal candidate able to describe dynamical structures on adequate levels of abstraction [12]. Within research projects during the last years, we developed several P systems frameworks coping within dynamical structures at molecular level, at the level of reaction network modules, and at the level of membranes. Most of these descriptive frameworks come with simulation software tools employed for tackling a number of application case studies.

P systems for cell signalling modules (Π_{CSM}) act at the *molecular level* [5]. Here, each molecule is represented by a regular expression denoted as a string. The characters within the string reflect the underlying signalling protein name together with an arbitrary number of ligands which in turn can be individually present or absent in the protein structure. Unknown or irrelevant binding situations are allowed to be written by a placeholder symbol ($\star$). A multiset of strings constructed in this way defines the initial pool of molecules. The set of reaction rules is also allowed to utilise placeholder symbols when describing substrates or products. Hence, the number of reaction rules can be kept low. Execution of a reaction rule includes a matching process which identifies the affected substrate molecules. We equipped the system's specification with a discretised form of reaction kinetics estimating the selection of substrate molecules taken into account per time step for each available reaction. Based on the Π_{CSM} framework, the simulation software SRSim emerged in which SR stands for "structural rules" but also for "spatial rules" [3]. In addition to the string describing a molecule, three-dimensional cartesian coordinates together with bond length and angles can be assigned to each molecular component. In this way, a reactive calotte-like model of each molecule is obtained.

Within the *level of reaction network modules*, we introduced the P meta framework for polymorphic processes [4, 7]. Here, the main focus of attention is laid to dynamical composition and decomposition of modules towards formalisation of more complex system's behaviour. We permit modifications of the module connectivity at arbitrary points in time but also subject to conditional trigger signals. This feature offers a high flexibility in formalisation of measurable system's properties which can be helpful to bring in silico-simulations closer to experimental observations. Furthermore, a compact but expressive formalism is provided to manage dynamical topologies of reaction network structures. The underlying concept resembles an *event-based programming language*: A program is built of a final set of *instructions*. Each instruction contains a specific *condition* (a boolean term based on evaluation of elapsed model time and conditional trigger signals) followed by a corresponding *action*. An action could be the connection of two dedicated modules including coupling of shared species and supply of affected signal values. Other actions incorporate disconnection of modules, coupling/decoupling of additional species, module exchange, or module reset. The sequence of instructions defines individual priorities in order to prevent ambiguities.

Aimed at exploring abilities of self-organisation by dynamical structures within the *level of membranes*, we currently elaborate the idea of grid-exploring P systems assuming an initial grid of membranes. Each membrane on its own acts in terms of a module. It can be entered, passed, and left by molecules. In some dedicated modules called processing units, molecules can be processed by reactions of different types like composition ($a + b \to c$), incorporation ($a + b \to a$), and unification ($a + a \to b$). Molecules initially placed at different positions of the grid's boundary individually run through the grid visiting a sequence of designated membranes in which they become successively processed. Using artificial evolution, the arrangement of membranes within the grid becomes optimised for shortening the total time duration necessary for complete passage and processing of all molecules. We employ grid-exploring P systems for topological grid optimisation using artificial evolution which in turn cares for variation of grid elements following the metaphor of *walking membranes*.

3 Usefulness of Membrane Systems Managing Dynamical Structures

Convincing simulations and visualisations of biological and biologically inspired processes utilising dynamical structures can be seen as a first and essential step towards a beneficial toolbox of complementary membrane system instances. Beyond pure system's definition along with estimation of its computational capacity, our research is focused on identification and exploration of useful practical applications and application scenarios for membrane systems managing dynamical structures. Projects and case studies are motivated by finding hypotheses to *explain* phenomena and afterwards being able to *predict* a system's behaviour. Having this knowledge at hand, it is worth to become *adopted* and *adapted* for

suitable engineering tasks in terms of bionics like construction of a girder inspired by a bone structure. Another application objective is dedicated to *optimise* a system's behaviour like the best possible topological arrangement of processing units on a grid. We believe that bringing together the descriptive advantages of membrane systems with the existence of biological phenomena under study and capabilities of data mining could be a fruitful strategy. To this end, we closely collaborate with experts of life sciences, engineering, or natural sciences in an interdisciplinary manner.

References

1. F. Bernardini, M. Gheorghe, N. Krasnogor, J.L. Giavitto. On Self-assembly in Population P Systems. *Lecture Notes in Computer Science* **3699**:46-57, 2005
2. S. Camazine, J.L. Deneubourg, N.R. Franks, J. Sneyd, G. Theraulaz, E. Bonabeau. *Self-Organization in Biological Systems.* Princeton University Press, 2003
3. G. Gruenert, B. Ibrahim, T. Lenser, M. Lohel, T. Hinze, P. Dittrich. Rule-based spatial modeling with diffusing, geometrically constrained molecules. *BMC Bioinformatics* **11**:307, 2010
4. T. Hinze, B. Schell, M. Schumann, C. Bodenstein. Maintenance of Chronobiological Information by P System Mediated Assembly of Control Units for Oscillatory Waveforms and Frequency. *Lecture Notes in Computer Science* **7762**:208-227, 2013
5. T. Hinze, J. Behre, C. Bodenstein, G. Escuela, G. Grünert, P. Hofstedt, Pe. Sauer, S. Hayat, P. Dittrich. Membrane Systems and Tools Combining Dynamical Structures with Reaction Kinetics for Applications in Chronobiology. In P. Frisco, M. Gheorghe, M.J. Perez-Jimenez (Eds.), *Applications of Membrane Computing in Systems and Synthetic Biology.*, Series Emergence, Complexity, and Computation, Vol. 7, pp. 133-173, Springer Verlag, 2014
6. T. Hinze, K. Kirkici, Pa. Sauer, Pe. Sauer, J. Behre. Membrane Computing Meets Temperature: A Thermoreceptor Model as Molecular Slide Rule with Evolutionary Potential. *Lecture Notes in Computer Science* **9504**:215-235, 2015
7. T. Hinze, J. Behre, K. Kirkici, Pa. Sauer, Pe. Sauer, S. Hayat. Passion to P for Polymorphic Processes in Practice. In M. Gheorghe, I. Petre, M.J. Perez-Jimenez, G. Rozenberg, A. Salomaa (Eds.), *Multidisciplinary Creativity.* Spandugino, 2016
8. H. Kitano. Computational Systems Biology. *Nature* **420**:206-210, 2002
9. C. Martin-Vide, G. Paun, J. Pazos, A. Rodriguez-Paton. Tissue P Systems. *Theoretical Computer Science* **296(2)**:295-326, 2003
10. N. Matsumaru, T. Lenser, T. Hinze, P. Dittrich. Toward Organization-Oriented Chemical Programming: A Case Study with the Maximal Independent Set Problem. In F. Dressler, I. Carreras (Eds.), *Advances in Biologically Inspired Information Systems: Models, Methods, and Tools. Series Studies in Computational Intelligence*, Vol. 69, pp. 147-163, Springer Verlag, 2007
11. A.E. Porreca, A. Leporati, G. Mauri, C. Zandron. P systems with active membranes working in polynomial space. *International Journal of Foundations of Computer Science* **22(1)**:65-73, 2011
12. G. Păun. *Membrane Computing: An Introduction.* Springer Verlag, 2002

Applications of P systems in population biology and ecology

Paolo Milazzo

Dipartimento di Informatica
University of Pisa, Italy
milazzo@di.unipi.it

Abstract

Maximal parallelism, one of the distinguishing features of P systems, is particularly suitable for the modelling of populations that evolve by stages in which all of the individuals are involved in the same activity (e.g. reproduction, hibernation, natural selection). In order for maximal parallelism to be successfully exploited for the modelling of populations and ecosystems, it is necessary to cope it with a form of probabilistic choice of evolution rules. In addition, also rule promoters turn out to be useful for the modelling of stage-based population dynamics. Indeed, they are a simple mechanism that allows evolution rules describing activities of the different stages to be properly alternated.

Minimal probabilistic P systems (MPP systems) [2] are a variant of P systems that we recently proposed and that includes only the above mentioned features. Although simple, such a variant of P systems allows for the development of very concise models of populations and ecosystems. We applied MPP systems to study the stability of some kinds of European water frog populations. In particular, we faced the problem of understanding why in some of these populations (known as "L-E complexes") that are the results of hybridization of two different kinds of frog, potentially lethal DNA mutations are accumulated in the genotype of the hybrid individuals. Model analysis allowed us to formulate the hypothesis that such mutations are actually necessary, together with a form of sexual selection, to avoid extinction of the whole population [5]. Moreover, we used the model to predict the effect of the introduction in a L-E complex of few individuals of the original kinds of frog (denoted R) from which the hybrid population was obtained. The result was that in many cases, due to the particular reproductive patterns of these animals, the introduction led either to the replacement of the L-E population with a population of only R individuals, or to the extinction of the whole population.

As model analysis techniques for MPP systems we considered probabilistic simulation and statistical model checking. Probabilistic simulation gives the trace of an individual model execution in a rather short time. Statistical model checking can be seen as a way of performing statistical analysis of simulation results in a very organized way, by formulating queries denoted as temporal logic formulas. Such a form of model checking is actually an approximated form

of probabilistic model checking in which the exploration of the state space describing the behaviour of the modelled system is not exhaustive, but limited to a number of traces obtained by means of probabilistic simulation. In the case study of European water frog populations we used statistical model checking to precisely compute the probability of extinction of the population in the different considered scenarios, and also to compute the probabilities of some causality properties that allowed us to identify observable situations that are early predictors of the extinction of the whole population.

Obviously, not all the aspects of stage-based populations can be described concisely by MPP systems. For instance, spatiality aspects as well as age structures and social hierarchies require several objects and rules to be defined in the model in order to take into account all parameters of each individual (e.g. position, age, dominance level). In order to allow also these aspects of populations and ecosystems to be suitably dealt with, in the last few years we proposed other variants of P systems such as attributed probabilistic P systems (APP systems) [1] and spatial P systems [4, 3].

References

1. R. Barbuti, A. Bompadre, P. Bove, P. Milazzo, and G. Pardini. Attributed probabilistic p systems and their application to the modelling of social interactions in primates. In *International Conference on Software Engineering and Formal Methods*, pages 176–191. Springer, 2015.
2. R. Barbuti, P. Bove, P. Milazzo, and G. Pardini. Minimal probabilistic p systems for modelling ecological systems. *Theoretical Computer Science*, 608:36–56, 2015.
3. R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, and G. Pardini. Simulation of spatial p system models. *Theoretical Computer Science*, 529:11–45, 2014.
4. R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, G. Pardini, and L. Tesei. Spatial p systems. *Natural Computing*, 10(1):3–16, 2011.
5. P. Bove, P. Milazzo, and R. Barbuti. The role of deleterious mutations in the stability of hybridogenetic water frog complexes. *BMC evolutionary biology*, 14(1):1, 2014.

Borderlines of efficiency: what's up?

Agustín Riscos-Núñez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012, Sevilla, Spain
ariscosn@us.es

Abstract. Some of the theoretical open problems and current challenges concerning P systems from a theoretical perspective will be discussed, focusing on results related to searching bounds for computational complexity classes in general, and to the **P** vs **NP** problem and the Păun's conjecture in particular.
Membrane computing is already a mature field, and the research community around it is becoming more interdisciplinary and diverse. Although nowadays it seems that practical applications receive a rapidly increasing attention, there are still key theoretical issues that need to be addressed.

Since the dawn of membrane computing, a huge number of universality results have been obtained, showing how to reach a computing power equivalent to Turing machines, considering P systems as computing devices (both in accepting or generative mode). Another important research direction consists on studying ways to generate an exponential workspace in polynomial time (i.e. in a polynomial number of transition steps), enabling significant speed-ups in solutions to hard problems. Informally speaking, computational complexity classes are introduced in membrane computing as follows: for each P system model, its associated complexity class is the set of all problems solvable in polynomial time by using such type of P systems [1]. These complexity classes can be compared against the "classical" ones within Chomsky hierarchy.

The talk will overview theoretical results from the literature, illustrating the different techniques used in the proofs, as well as some applications of the results and the techniques. Open problems in these areas usually involve removing or restricting some of the ingredients of a P system model, in order to determine whether such an ingredient plays a relevant role or not, as far as the power of the model is concerned. This gives rise in a natural way to alternative formulations for borderlines of tractability, which provide, in turn, different approaches to tackle the **P** vs **NP** problem.

References

1. 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.

Regular Papers

Simulating R Systems by P Systems

Artiom Alhazov[1], Bogdan Aman[2], Rudolf Freund[3], and Sergiu Ivanov[3]

[1] Institute of Mathematics and Computer Science, Academy of Sciences of Moldova
Str. Academiei 5, Chişinău, MD 2028, Moldova
E-mail: artiom@math.md
[2] Romanian Academy, Institute of Computer Science, Iaşi, Romania
Blvd. Carol I no.8, 700505 Iaşi, Romania
E-mail: bogdan.aman@gmail.com
[3] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: rudi@emcc.at
[4] Université Paris Est, France
E-mail: sergiu.ivanov@u-pec.fr

Abstract. We show multiple ways to simulate R systems by non-cooperative P systems with atomic control by promoters and/or inhibitors, or with matter-antimatter annihilation rules, with a slowdown by a factor of constant. The descriptional complexity is also linear with respect to that of simulated R system. All these constants depend on how general the model of R systems is, as well as on the chosen control ingredients of P systems. Special attention is paid to the differences in the mode of rule application in these models.

1 Introduction. Differences between P and R

Membrane systems, also called P systems (non-distributed, with symbol objects) are a formal model of (possibly controlled) multiset rewriting [8]. Reaction systems, also called R systems, is also a formal rewriting-like model of set evolution introduced in [6], see also a recent survey [5]. Both P systems and R systems are inspired by the functioning of the living cells. It is a natural task to compare R systems, which was introduced later, to P systems, by simulation. A successful solution would allow us to use membrane computing tools for studying reaction systems. Some research comparing them was done in [10]. More exactly, the cited paper considers P systems with no-persistence aspect of R systems, from the viewpoint of the computational power. We, however, first focus on comparing standard R systems to standard P systems by simulating the former with latter, and then revisit the direction of bringing aspects of R systems to the P systems model, verifying how closer this can make the models.

We start the explanation of the simplest case – triples of single objects. Rules in R systems have form (a, b, c), which loosely correspond to $a \to c|_{\neg b}$ in P systems, i.e., the first element is the reactant (in this paper we may also call it the left side) the second element is the product (in this paper we may also

call it the right side), and the third element is the inhibitor, with the following differences in the mode of application.

The first difference is that the configuration is a set, not a multiset, and thus simultaneously producing the same symbol by multiple rules yields a single object. P systems with sets of objects instead of multisets of objects have been considered in [1], where they have been shown to be universal in the distributed P systems, both for the transitional model, and for the model with active membranes. However, in [1] the goal of showing universality was reached without actually using this first aspect (automatic reduction of multiple copies of identical object into one object), but rather by avoiding to ever need multiple copies of the same object (in the same region). This aspect, combined with the one below, are called the *threshold principle* in the literature. However, it is also meaningful to view these aspects individually.

The second difference is that, if multiple rules with the same a in the left side exists, (if a is present in the configuration, for all of these rules where the inhibitors are not present in the configuration) **all** these rules are applied, simultaneously producing the corresponding products. This comes from an inspiration that either the abundance of objects a is sufficient, or the replication and, possibly, proper control take place to guarantee the application of all such rules. This second aspect is standard, for instance, in *H systems* (together with the first one), e.g., see [11]. The second aspect has been already considered also in P systems area, see, for example, see [3].

The third difference is that the objects are not persistent. This means that, even if an object does not undergo any rule, it still disappears from the configuration of the next step, unless, of course, it is produced by some rule. This third aspect is standard in time-varying distributed H systems, for example see [9, 12], together with the first and second ones, and they relate especially naturally to *TVDH1* systems, see [7].

In the general case, the elements of the triples describing the rules of R systems are **sets** of objects. Hence, the meaning of the triple (A, B, C) is: the joint presence of objects in A, in the case when all objects in B are absent, leads to production of objects in C, and, moreover, the subsequent configuration is precisely equal to the union of the right sides of applicable rules (possibly united with the input context).

2 Preliminaries

The reader is assumed to be familiar with the basic notions of formal languages and membrane computing, see [13] for a comprehensive introduction and the webpage [15] of P systems.

The notation $(ncoo, pro_{k,l} + inh_{k',l'})$ describes the possible class of rules: non-cooperative evolution with at most k promoters of weight at most l and at most k' inhibitors of weight at most l', see [4, 14]; the sign "+" here means both promoters and inhibitors are allowed to be used in the same rule, if it is not the case, we write a comma instead of a plus sign.

The notation $(ncoo, antim/pri)$ stands for non-cooperative evolution rules and matter-antimatter annihilation rules, with weak priority of *all* annihilation rules assumed over all other rules (the most studied variant of P systems with antimatter), see [2].

3 Using promoters and inhibitors

In fact, in terms of intuition from P systems, A is more similar to a promoter than a reactant (and there is no difference between a set of distinct atomic promoters and a corresponding one higher-weight promoter), and B corresponds to a set of atomic inhibitors (if B were a single higher-weight inhibitor, it would disable the rule when all its elements are present, not just any of them, which would not correspond to the correct definition). However, within the traditional P systems mode, we would additionally need to restrict the rule application to only once per step.

We recall that a rule with the set of atomic inhibitors $\{a, b\}$ will be disabled in configurations in which *either* a or b is present, while a rule equipped with the higher-weight inhibitor ab will only be disabled when *both* a and b are present in the configuration.

3.1 Using powerful rules

Hence, an arbitrary general R system with alphabet V of k symbols and rules (A_i, B_i, C_i), $1 \leq i \leq n$ could be written as the following P system (non-cooperative, but with powerful promoters and inhibitors), having additional objects I_1 and d_i for all $1 \leq i \leq n$:

$$\begin{aligned}
&\Pi_0 = (O, \mu = [\ ]_1, w_1, R_1) \text{ where} \\
&O = V \cup \{d_i \mid 1 \leq i \leq n\} \cup \{I_1\}, \\
&R_1 = \{d_i \to \prod\nolimits_{c \in C_i} c|_{A_i, \{\neg b \mid b \in B_i\}},\ d_i \to \lambda|_{\neg A_i},\ d_i \to \lambda|_b \ \mid b \in B_i,\ 1 \leq i \leq n\} \\
&\quad \cup \{a \to \lambda \mid a \in V\} \cup \{I_1 \to I_1 \prod\nolimits_{1 \leq i \leq n} d_i\}.
\end{aligned}$$

This combination of features only takes one step to simulate a step of R systems, $n + k + 1$ symbols and $2n + k + 1 + \sum_{1 \leq i \leq n} |B_i|$ rules. Note that the first rule in the description of R_1 uses a higher-weight promoter *together* with a *set* of atomic inhibitors. Also note that in a special case when the rules of the simulated R system are triples of *single* symbols, the control used becomes atomic promoters *together* with atomic inhibitors.

In the rest of the paper we show how to achieve the same goal with P systems having more restricted rules, also discussing how to produce only *one* copy of symbols present in the simulated R system. We use promoters and inhibitors, then consider only one kind of these features, then we replace them by matter-antimatter annihilation rules, and finally, we discuss how much the problem is simplified if some of the aspects of R systems are assumed by the P systems model.

3.2 Triples of symbols

We start with the simplest case - when the elements of triples describing the rules are single elements. Consider such an R system S with alphabet V and rules $\{(a_i, b_i, c_i) \mid 1 \leq i \leq n\}$. We construct a P system Π_1 simulating S, where the initial configuration w_1 matches the initial configuration of S, and the following rules, the simulation taking only 2 steps:

$$\begin{aligned}
\Pi_1 &= (O, \mu = [\ ]_1, w_1, R_1) \text{ where} \\
O &= V \cup \{a' \mid a \in V\} \cup \{d_i \mid 1 \leq i \leq n\}, \\
R_1 &= \{a \to a' \prod\nolimits_{1 \leq i \leq n, a_i = a} d_i \mid a \in V\} \\
&\quad \cup \{d_i \to c_i|_{\neg (b_i)'},\ d_i \to \lambda|_{(b_i)'} \mid 1 \leq i \leq n\} \cup \{a' \to \lambda \mid a \in V\}.
\end{aligned}$$

The simulation task here is simple for two reasons: we took the simpler model of R systems, and using promoters besides inhibitors makes it possible to remove unneeded objects easily. We also note that the number of objects and rules can be decreased by not producing a' when a participates in the left side of any rule, and using $d_{\min\{j|1 \leq j \leq n,\ a_j = b\}}$ instead of b' as promoter and inhibitor.

If $|V| = k$, then $|O| = 2k + n$ and $|R_1| = 2k + 2n$. Moreover, the optimization described in the previous paragraph decreases both $|O|$ and $|R_1|$ by the number of symbols appearing on the left side of some rule of S.

The multiplicities of symbols may grow. When the same symbol is produced simultaneously by multiple rules, the multiplicative effect happens. It is, however, fairly easy to reset the multiplicities of objects in V to one, at a cost of one more step, $2k + 3$ additional symbols in O and $3k + 3$ additional rules, also using an additional object I_1 in the initial configuration:

$$\begin{aligned}
\Pi_2 &= (O, \mu = [\ ]_1, w_1, R_1 = R_i \cup R_{ii} \cup R_{iii}) \text{ where} \\
O &= V \cup \{a', a'', \overline{a} \mid a \in V\} \cup \{d_i \mid 1 \leq i \leq n\}, \cup \{I_1, I_2, I_3\} \\
R_i &= \{a \to a' \prod\nolimits_{1 \leq i \leq n, a_i = a} d_i \mid a \in V\} \cup \{I_1 \to I_2\}, \\
R_{ii} &= \{d_i \to (c_i)''|_{\neg (b_i)'},\ d_i \to \lambda|_{(b_i)'} \mid 1 \leq i \leq n\} \\
&\quad \cup \{a' \to \lambda \mid a \in V\} \cup \{I_2 \to I_3 \prod\nolimits_{a \in V} \overline{a}\}, \\
R_{iii} &= \{\overline{a} \to a|_{a''},\ \overline{a} \to \lambda|_{\neg a''},\ a'' \to \lambda \mid a \in V\} \cup \{I_3 \to I_1\}.
\end{aligned}$$

3.3 Triples of sets

Now the task is more complicated. While generating a set C_i instead of symbol c_i is straightforward, instead of verifying that a_i is present and b_i is absent, rule applicability is defined as presence of **all** symbols from set A_i and absence of **all** symbols from set B_i. We recall that our task is a constant-time solution. Notice that the rule is not applicable if and only if some symbol from A_i is absent or some symbol from B_i is present.

Consider such an R system S with alphabet V and rules $\{(A_i, B_i, C_i) \mid 1 \leq i \leq n\}$. We construct a P system Π_3 simulating S, where the initial configuration matches the initial configuration of S, plus an additional object I_1, and the following rules, the simulation taking only 3 steps:

$$\begin{aligned}
\Pi_3 &= (O, \mu = [\ ]_1, w_1, R_1) \text{ where} \\
O &= V \cup \{d_i \mid 1 \leq i \leq n\} \cup \{I_1, I_2, I_3\}, \\
R_1 &= \{I_1 \to d_1 \cdots d_n I_2\} \\
&\quad \cup \{d_i \to \lambda|_{\neg a},\ d_i \to \lambda|_b \mid a \in A_i,\ b \in B_i,\ 1 \leq i \leq n\} \cup \{I_2 \to I_3\} \\
&\quad \cup \{d_i \to \prod_{c \in C_i} c|_{I_3} \mid 1 \leq i \leq n\} \cup \{a \to \lambda|_{I_3} \mid a \in V\} \cup \{I_3 \to I_1\}.
\end{aligned}$$

If $|V| = k$, then $|O| = k + n + 3$ and $|R_1| = k + n + 3 + \sum_{1 \leq i \leq k}(|A_i| + |B_i|)$. Notice also that, besides objects from V, no object ever appears in multiple copies. As for each object from V, its multiplicity represents the number of rules in S that has produced it in the last simulated step. Unlike the construction from the previous section, the multiplicative effect does not carry over to the next step of computation of S, since each object from V (except the instances in the starting configuration) is produced from some object d_i, produced in one copy, effectively resetting the multiplicities of the previous step. However, producing objects in V in a single copy requires additional overhead. Similarly to obtaining Π_2 from Π_1, we can obtain Π_4 from Π_3, at the price of one more step, $2k + 1$ additional symbols in O and $3k + 1$ additional rules. We skip the details.

3.4 Using only promoters

It should not be any surprise that (in the maximally parallel mode) the effect of inhibitors can be obtained by non-cooperative rules with promoters only. Informally, to verify that some object b is absent, we first check if b is present by some rule $a \to a'|_b$, and it suffices to check in the next step whether a is unchanged. The reverse, i.e. replacing promoters with inhibitors, is even easier to see, since promoting a rule by b can be modeled by inhibiting a rule by some immediately-erased object b', creation of which is inhibited by b. We still think it is interesting to consider the use of only promoters or only inhibitors, for two reasons. First, the reduction of promoters/inhibitors in the *general* case of P systems is too complicated, and second, we would like to explore how little overhead in terms of slowdown and descriptional complexity would suffice to achieve our task.

First, as an exercise, we construct a P system for an R system S with triples of symbols $\{(a_i, b_i, c_i)\}$ as rules. The initial configuration matches the initial configuration of S, plus an additional object I_1.

$$\Pi_5 = (O, \mu = [\ ]_1, w_1, R_1) \text{ where}$$

$$
\begin{aligned}
O &= V \cup \{a' \mid a \in V\} \cup \{d_i \mid 1 \le i \le n\} \cup \{I_1, I_2, I_3\},\\
R_1 &= \{I_1 \to I_2\} \cup \{a \to a' \prod\nolimits_{1 \le i \le n, a_i = a} d_i \mid a \in V\}\\
&\quad \cup \{I_2 \to I_3\} \cup \{d_i \to \lambda|_{(b_i)'} \mid 1 \le i \le n\} \cup \{a' \to \lambda \mid a \in V\}\\
&\quad \cup \{I_3 \to I_1\} \cup \{d_i \to c_i|_{I_3}, \mid 1 \le i \le n\}.
\end{aligned}
$$

This construction is obtained from the first one with promoters and inhibitors, implementing the group of rules with inhibitors (contrasted with existing rules with the same objects as promoters) in the next step, promoted by "timer" I_3. We also note, similarly to Π_1, that the number of objects and rules can be decreased by not producing a' when a participates in the left side of any rule, and using $d_{\min\{j \mid 1 \le j \le n,\ a_j = b\}}$ instead of b' as promoter. Once again, this simulation has multiplicative effect, and the multiplicities can be reset to one, at the price of one more step, $2k+1$ additional symbols in O and $3k+1$ additional rules. Let us call the obtained system Π_6. We omit the details, only mentioning that instead of rules $\overline{a} \to \lambda|_{\neg a''}$ as in Π_2, we can erase these symbols in the next step by rules $\overline{a} \to \lambda|_{I_1}$.

Now consider the general case of simulating an R system S with alphabet V and rules $\{(A_i, B_i, C_i) \mid 1 \le i \le n\}$. The simulating P system below has the initial configuration which matches the initial configuration of S, plus additional objects I_1 and a' for each $a \in V$.

$$
\begin{aligned}
\Pi_7 &= (O, \mu = [\ ]_1, w_1, R_1 = R_i \cup R_{ii} \cup R_{iii}) \text{ where}\\
O &= V \cup \{a' \mid a \in V\} \cup \{d_i \mid 1 \le i \le n\} \cup \{I_1, I_2, I_3\},\\
R_i &= \{I_1 \to d_1 \cdots d_n I_2\} \cup \{a' \to \lambda|_a \mid a \in V\},\\
R_{ii} &= \{d_i \to \lambda|_{a'},\ d_i \to \lambda|_b \mid a \in A_i,\ b \in B_i,\ 1 \le i \le n\}\\
&\quad \cup \{a \to \lambda|_{I_2},\ a' \to \lambda|_{I_2} \mid a \in V\} \cup \{I_2 \to I_3\},\\
R_{iii} &= \{d_i \to \prod\nolimits_{c \in C_i} c|_{I_3} \mid 1 \le i \le n\} \cup \{I_3 \to I_1 \prod\nolimits_{a \in V} a'\}.
\end{aligned}
$$

This construction is obtained from the second one with promoters and inhibitors, as follows. The role of objects a' is to survive for one step if and only if the corresponding object a is present, to be used as a promoter instead of inhibitor a; objects a' are recreated in the last step, for the next simulation cycle. Moreover, as now objects from V are no longer used as inhibitors, they can be removed one step earlier.

The system above needs only 3 steps to simulate a step of S, and if $|V| = k$, then $|O| = 2k + n + 3$ and $|R_1| = 3 + 3k + n + \sum_{1 \le i \le n}(|A_i| + |B_i|)$. Of course, alternatively, objects a' could be created from one additional initial object, at a price of an additional step and a few extra rules, but we currently focus on constructions that are efficient in time and descriptional complexity. We again comment that, although this construction has no multiplicative effect, the number of copies of a symbol in V produced in the end of the simulation equals the number of rules in S that have produced this symbol in the last step. Producing

exactly one copy needs one more step, $2k+1$ additional symbols in O and $3k+1$ additional rules. We call this system Π_8 and give no more details, since obtaining it from Π_7 is exactly like obtaining Π_6 from Π_5.

3.5 Using only inhibitors

First, as an exercise, we construct a P system for an R system S with triples of symbols $\{(a_i, b_i, c_i)\}$ as rules. The initial configuration matches the initial configuration of S, plus an additional object I_1.

$$\begin{aligned}
\Pi_9 &= (O, \mu = [\ ]_1, w_1, R_1) \text{ where} \\
O &= V \cup \{a' \mid a \in V\} \cup \{d_i \mid 1 \le i \le n\} \cup \{I_1, I_2\}, \\
R_1 &= \{I_1 \to I_2\} \cup \{a \to a' \prod\nolimits_{1 \le i \le n, a_i = a} d_i \mid a \in V\} \\
&\quad \cup \{I_2 \to I_1\} \cup \{d_i \to c_i|_{\neg (b_i)'} \mid 1 \le i \le n\} \\
&\quad \cup \{d_i \to \lambda|_{\neg I_2} \mid 1 \le i \le n\} \cup \{a' \to \lambda|_{\neg I_2} \mid a \in V\}.
\end{aligned}$$

This construction is obtained from the one with promoters and inhibitors, implementing the group of rules with promoters (contrasted with existing rules with the same objects as inhibitors) in the next step, inhibited by "timer" I_2. Moreover, removing objects a' is delayed for one step, to make sure that the rules inhibited by them in the second step are not applied in the third step. Notice also that the simulation of a computation step of S only takes two steps of computation in Π; the third step of computation cleaning objects d_i and a' overlaps with the first step of simulation of the next step in S. However, this produces no interference, since sub-alphabets $\{d_i \mid 1 \le i \le n\} \cup \{a' \mid a \in V\}$ and $\{I_1\} \cup V$ are disjoint. We also note, similarly to Π_1, that the number of objects and rules can be decreased by not producing a' when a participates in the left side of any rule, and using $d_{\min\{j \mid 1 \le j \le n,\ a_j = b\}}$ instead of b' as promoter.

The problem of multiplicative effect can be solved in the usual way, resetting multiplicities to one: produce one copy of each candidate-object, and erase the objects where the multiplicity is zero. However, with inhibitors it takes longer: one additional step to erase objects $\overline{a}$ when the corresponding object a'' is absent, and one further step to rewrite $\overline{a}$ into a.

$$\begin{aligned}
\Pi_{10} &= (O, \mu = [\ ]_1, w_1, R_1) \text{ where} \\
O &= V \cup \{a', a'', \overline{a} \mid a \in V\} \cup \{d_i \mid 1 \le i \le n\} \cup \{I_1, I_2, I_3, I_4\}, \\
R_1 &= \{I_1 \to I_2\} \cup \{a \to a' \prod\nolimits_{1 \le i \le n, a_i = a} d_i \mid a \in V\} \\
&\quad \cup \{I_2 \to I_3 \prod\nolimits_{a \in V} \overline{a}\} \cup \{d_i \to (c_i)''|_{\neg (b_i)'} \mid 1 \le i \le n\} \\
&\quad \cup \{I_3 \to I_4\} \cup \{d_i \to \lambda|_{\neg I_2} \mid 1 \le i \le n\} \cup \{a' \to \lambda|_{\neg I_2},\ \overline{a} \to \lambda|_{\neg a''} \mid a \in V\} \\
&\quad \cup \{I_4 \to I_1\} \cup \{\overline{a} \to a|_{\neg I_3},\ a'' \to \lambda|_{\neg I_3} \mid a \in V\}.
\end{aligned}$$

Hence, the total additional price for resetting the multiplicities of elements of V to one using only inhibitors is 2 more steps, $2k+2$ additional objects, and $3k+2$ rules.

Now consider the general case of simulating an R system S with alphabet V and rules $\{(A_i, B_i, C_i) \mid 1 \leq i \leq n\}$. The simulating P system below has the initial configuration which matches the initial configuration of S, plus additional objects I_1, J and b' for each $b \in V$.

$$
\begin{aligned}
\Pi_{11} &= (O, \mu = [\ ]_1, w_1, R_1 = R_i \cup R_{ii} \cup R_{iii}) \text{ where} \\
O &= V \cup \{b', b'' \mid b \in V\} \cup \{d_i \mid 1 \leq i \leq n\} \cup \{I_1, I_2, I_3, J\}, \\
R_i &= \{I_1 \to d_1 \cdots d_n I_2 J\} \cup \{b' \to b''|_{\neg b} \mid b \in V\} \cup \{J \to \lambda\}, \\
R_{ii} &= \{d_i \to \lambda|_{\neg a},\ d_i \to \lambda|_{\neg b''} \mid a \in A_i,\ b \in B_i,\ 1 \leq i \leq n\} \\
&\quad \cup \{b' \to \lambda|_{\neg I_1} \mid b \in V\} \cup \{I_2 \to I_3,\ J \to \lambda\}, \\
R_{iii} &= \{d_i \to \prod\nolimits_{c \in C_i} c|_{\neg I_2} \mid 1 \leq i \leq n\} \\
&\quad \cup \{a \to \lambda|_{\neg J} \mid a \in V\} \cup \{b'' \to \lambda|_{\neg I_2} \mid b \in V\} \cup \{I_3 \to I_1 J \prod\nolimits_{b \in V} b'\}.
\end{aligned}
$$

This construction is obtained from the second one with promoters and inhibitors, as follows. The role of objects b' is to change into b'' if and only if the corresponding object b is present, so b'' can be used as an inhibitor instead of promoter b; objects b' are recreated in the last step, for the next simulation cycle. Moreover, to make sure the rules erasing d_i in the absence of a are not applied in the third step, objects a can only be removed in the third step. This is why an additional object J is present in each of the first two steps of the simulation, inhibiting premature removal of objects a. The rule erasing J is written both in R_i and R_{ii} only to highlight that it is applied both in the first and in the second step.

The system above needs only 3 steps to simulate a step of S, and if $|V| = k$, then $|O| = 3k + n + 4$ and $|R_1| = 4 + 4k + n + \sum_{1 \leq i \leq n}(|A_i| + |B_i|)$. Of course, alternatively, objects b' could be created from one additional initial object, at a price of an additional step and a few extra rules, but we currently focus on constructions that are efficient in time and descriptional complexity. Resetting to one the multiplicities of objects in V can be done exactly how Π_{10} was constructed from Π_9. Hence, the new system Π_{12} will have, compared to Π_{11}, 2 more steps, $2k+2$ additional objects, and $3k+2$ rules.

4 Using antimatter

This section is devoted to a different control mechanism: matter-antimatter annihilation rules are used instead of promoters and/or inhibitors. The weak priority of annihilation rules over non-cooperative rules is assumed, which is the most common variant of the antimatter model. First, we notice that erasing with a promoter, say, $d \to \lambda|_b$, in the case the promoting object b is erased without

being used anywhere else, and when the number of copies of d is bounded, can be modeled by antimatter as follows:

- replace the promoting object b by anti-object d^- of the promoted object, in sufficient copies to erase all possible copies of promoted object d,
- add erasing rules for this anti-object d^- to remove the copies of the anti-objects which did not annihilate.

We now construct the P system equivalent to Π_1 using antimatter.

$$\begin{aligned}\Pi_{13} &= (O = V \cup \{d_i, d_i^- \mid 1 \le i \le n\}, \mu = [\ ]_1, w_1, R_1) \text{ where}\\ R_1 &= \{a \to \prod\nolimits_{1\le i\le n, b_i=a} d_i^- \prod\nolimits_{1\le i\le n, a_i=a} d_i \mid a \in V\}\\ &\quad \cup \{d_i d_i^- \to \lambda,\ d_i \to c_i,\ d_i^- \to \lambda \mid 1 \le i \le n\}.\end{aligned}$$

In each rule of the first group of R_1, it is enough to produce a single copy of d_i^-, because at most one d_i may be generated by the system in the same step, since the rule uniquely determines its left side. The simulation only takes two steps, and uses $k + 2n$ objects and $k + 3n$ rules.

This construction, too, has multiplicative effect. Resetting multiplicities to one can be done by two-step annihilation. Say, we got some number (possibly zero) of objects c'', and we only want to know whether this number is positive. Then we produce one copy of $(c'')^-$ and rewrite it to c' if it does not immediately annihilate. One step later, we produce one copy of $(c')^-$, and rewrite it to c if it does not immediately annihilate. As a result, c will appear if and only if $(c')^-$ did not annihilate, i.e., c' did not appear one step before. But this happened if and only if $(c'')^-$ was annihilated, i.e., there was at least one copy of c'' two steps before. Performing this routine to objects in V of Π_{13}, we obtain the following system, using an additional starting object I_1:

$$\begin{aligned}\Pi_{14} &= (O, \mu = [\ ]_1, w_1, R_1) \text{ where}\\ O &= V \cup \{a', a'', (a')^-, (a'')^- \mid a \in V\} \cup \{d_i, d_i^- \mid 1 \le i \le n\} \cup \{I_1, I_2, I_3, I_4\},\\ R_1 &= \{a \to \prod\nolimits_{1\le i\le n, b_i=a} d_i^- \prod\nolimits_{1\le i\le n, a_i=a} d_i \mid a \in V\} \cup \{I_1 \to I_2\}\\ &\quad \cup \{d_i d_i^- \to \lambda,\ d_i \to (c_i)'',\ d_i^- \to \lambda \mid 1 \le i \le n\} \cup \{I_2 \to I_3 \prod\nolimits_{a\in V} (a'')^-\}\\ &\quad \cup \{a''(a'')^- \to \lambda,\ (a'')^- \to a' \mid a \in V\} \cup \{I_3 \to I_4\}\\ &\quad \cup \{a'(a')^- \to \lambda,\ (a')^- \to a \mid a \in V\} \cup \{I_4 \to I_1\}.\end{aligned}$$

As you can see, resetting multiplicities with antimatter has a price of two more steps, $4k + 4$ additional objects and $4k + 4$ additional rules.

Now consider the general case of simulating an R system S with alphabet V and rules $\{(A_i, B_i, C_i) \mid 1 \le i \le n\}$. The simulating P system below has the

initial configuration which matches the initial configuration of S, plus additional object I_1.

$$\begin{aligned}
\Pi_{15} &= (O, \mu = [\ ]_1, w_1, R_1 = R_i \cup R_{ii} \cup R_{iii} \text{ where} \\
O &= V \cup \{a', (a')^-, a'', \mid a \in V\} \cup \{d_i, d_i^- \mid 1 \le i \le n\} \cup \{I_1, I_2, I_3\}, \\
R_i &= \{I_1 \to I_2 d_1 \cdots d_n \prod\nolimits_{a \in V} a'\} \cup \{a \to (a')^- a'' \mid a \in V\}, \\
R_{ii} &= \{a'(a')^- \to \lambda,\ a' \to d_i^-,\ b'' \to d_i^-,\ (a')^- \to \lambda \\
&\qquad \mid a \in A_i,\ b \in B_i,\ 1 \le i \le n\} \cup \{I_2 \to I_3\}, \\
R_{iii} &= \{d_i d_i^- \to \lambda,\ d_i \to \prod\nolimits_{c \in C_i} c,\ d_i^- \to \lambda \mid 1 \le i \le n\} \cup \{I_3 \to I_1\}.
\end{aligned}$$

Symbols from C_i are produced from d_i if and only if it is not annihilated, i.e., neither a' nor b'' should produce d_i^- for any $a \in A_i$, $b \in B_i$. Since a' is annihilated if and only if a is present, and b'' is not produced if and only if b is absent, the simulation of an application of rule i of the R system happens if and only if all symbols from the first set are present and all symbols from the second set are absent. The simulation takes 3 steps, using the alphabet of $4k + 2n + 3$ symbols and the set of $3k + 3n + 3 + \sum_{1 \le i \le n}(|A_i| + |B_i|)$ rules.

This construction produces each symbol in multiplicity equal to the number of rules of S that produced it, not carrying the multiplicative effect to the next step. If needed, resetting multiplicities can also be done, which costs two more steps, $4k + 2$ additional objects and $4k + 2$ additional rules. We call this system Π_{16}, and provide no more details since it is obtained from Π_{15} exactly as Π_{14} is obtained from Π_{13}.

5 Non-standard P systems

Some difficulty of simulation of R systems by P systems arises from the differences in their standard derivation modes. We would like to discuss how varying this may affect the problem.

First, if we consider P systems **with sets** instead of multisets, where production of a symbol multiple times still yields a single copy of the result, then *all* constructions in this paper still hold literally, i.e., no changes in the description of these P systems are needed. However, some things may become simpler, e.g., in this case resetting multiplicities to one is done by the model, and does not require additional time, symbol and rule complexity.

We note that, in a non-distributed case, P systems with sets of objects are no longer universal, since the number of possible configuration is bounded by two to the power of the cardinality of the alphabet. However, universality is not needed to simulate R systems (which has also been shown in the case of deterministic P systems with promoters and/or inhibitors).

Second, if we consider P systems which deterministically apply *all* individually applicable rules, even with overlapping left sides (i.e., competing for resources), then of course the existing solutions still literally hold, but in some

cases there are much easier ways: we would have no need to explicitly produce multiple objects from one. For instance, the constructions in this paper usually involve production of rule labels d_i, either from the corresponding reactant a_i, or from some "timer" object I_j, and then have different rules processing these label objects. In this "auto-replication" mode, these various processing rules could be applied directly to the corresponding original object a_i or I_j, the replication being done by the model itself. This would definitely simplify the simulation. Let us refer to this aspect as **auto-replication**. For example, the set of rules of system Π_1 can be simplified to the following:

$$\{a_i \to c_i|_{\neg b_i} \mid 1 \leq i \leq n\} \cup \{a \to \lambda \mid a \in V\},$$

i.e., just one step, no additional objects and k additional rules. The problem with resetting the multiplicities is also simpler:

$$\begin{aligned}
\Pi &= (O, \mu = [\ ]_1, w_1, R_1 \text{ where} \\
O &= V \cup \{a' \mid a \in V\} \cup \{I_1, I_2\}, \\
R_1 &= \{I_1 \to I_2\} \cup \{a_i \to (c_i)'|_{\neg b_i} \mid 1 \leq i \leq n\} \cup \{a \to \lambda \mid a \in V\} \\
&\quad \cup \{I_2 \to I_1\} \cup \{I_2 \to c|_{c'}\} \cup \{c' \to \lambda \mid c \in V\},
\end{aligned}$$

i.e., requiring only one more step, $k + 2$ additional symbols and $2k + 2$ additional rules (compared to increase of complexity of Π_2 over Π_1 by one step, $2k + 3$ symbols and by $3k + 3$ rules).

Third, if we consider P systems where idle objects (i.e., those not consumed by applied rules) do not contribute to the next configuration, we call this aspect "**no persistence**", then many erasing rules (in particular, all erasing promoted or inhibited by some "timer" I_j) would no longer be needed, while occasionally some renaming rules should be added when object was designed to be used later than in the next step after its production. For instance, in case of no-persistence, all $n + k'$ erasing rules of Π_1 may be removed, leaving just $n + k$ rules. Similarly, all erasing rules of Π_2 may be removed; hence, the subtask of resetting the multiplicities to one in this case only needs $k + 3$ additional rules instead of $3k + 3$.

However, testing for presence of some object b by "failing to apply a rule with inhibitor b and finding the reactant unchanged in the next step" would not work. The working solution is to use b as an inhibitor in a rule producing some object b', and to use b' as an inhibitor in the next step. Testing for absence by "failing to apply a rule with a promoter and finding the reactant unchanged in the next step" would be no longer possible, so the model with promoters only seems to be considerably weaker in the case without persistence of idle objects.

We would like to note that, in case of P systems with sets and auto-replication, the aspect of no-persistence can be simulated as follows: add rules $a \to \lambda$ for each $a \in V$; they will make sure that such objects are not carried over to the next step, in the same time not adding anything to the result (as

for productive objects, erasing them is just another option, which in the auto-replication case neither grows nor shrinks the set of objects obtained from them). This simulation takes one step, k objects and $n + k$ rules.

And, of course, if we consider P systems with all these differences, i.e., with sets, auto-replication, and without object persistence, then rule (a, b, c) of R systems becomes *identical* to rule $a \to c|_{\neg b}$ of P systems, while rule (A, B, C) of R systems becomes *identical* to rule $\prod_{a \in A} a \to \prod_{c \in C} c|_{\{\neg b | b \in B\}}$, so the simulation is trivial, requiring one step, k objects and n rules of type $(ncoo, pro_{1,*} + inh_{*,1})$.

6 Conclusions

We recall that although deterministic P systems with promoters and/or inhibitors are not universal and have subregular characterizations, their power is sufficient to simulate R systems.

All constructions presented in this paper (except those in previous section) simulate R systems (in their standard derivation mode) by P systems (in *their* standard derivation mode), with the slowdown by a factor of constant, where the descriptional complexity of the simulating P system is linear with respect to the descriptional complexity of the simulated R system. The proportionality constants vary depending on whether R systems are defined as triples of symbols or as triples of sets of symbols, and on whether promoters, inhibitors or both are used in P systems. All constructions are deterministic: while the multiset of rules to be applied to a given configuration may not be unique, the next configuration is unique. Indeed, in all these constructions, if two rules have the same left side, then either their applicability is mutually exclusive (one is being promoted and the other one is being inhibited by the same symbol), or also the right side is the same (and thus, if there are multiple choices which object would promote or inhibit the rule, such choice would not influence the result).

Seventeen constructions are presented, see Table 1: 0)(general and simple in particular) P systems using single higher-weight promoters together with multiple atomic inhibitors, 1)simple P systems using promoters and inhibitors, 2)simple P systems using promoters and inhibitors and resetting multiplicities to one, 3)general P systems using promoters and inhibitors, 4)general P systems using promoters and inhibitors and resetting multiplicities to one, 5)simple P systems using promoters, 6)simple P systems using promoters and resetting multiplicities to one, 7)general P systems using promoters, 8)general P systems using promoters and resetting multiplicities to one, 9)simple P systems using inhibitors, 10)simple P systems using inhibitors and resetting multiplicities to one, 11)general P systems using inhibitors, 12)general P systems using inhibitors and resetting multiplicities to one, 13)simple P systems using antimatter, 14)simple P systems using antimatter and resetting multiplicities to one, 15)general P systems using antimatter, 16)general P systems using antimatter and resetting multiplicities to one. The table below shows the number of steps of simulating P system to simulate one step of R system, alphabet size and the number of rules in these simulations (n is the number of rules in S, k is the number of symbols

P	R	mult	features	steps	$\lvert O\rvert$	$\lvert R_1\rvert$
Π_0	s	L	$(ncoo, pro_{1,1} + inh_{1,1})$	1	$n+k+1$	$3n+k+1$
Π_0	G	L	$(ncoo, pro_{1,*} + inh_{*,1})$	1	$n+k+1$	$2n+k+1+T'$
Π_1	s	M	$(ncoo, pro_{1,1}, inh_{1,1})$	2	$n+k+k'$	$2n+k+k'$
Π_2	s	1	$(ncoo, pro_{1,1}, inh_{1,1})$	3	$n+3k+k'+3$	$2n+4k+k'+3$
Π_3	G	L	$(ncoo, pro_{1,1}, inh_{1,1})$	3	$n+k+3$	$n+k+3+T$
Π_4	G	1	$(ncoo, pro_{1,1}, inh_{1,1})$	4	$n+3k+4$	$n+4k+4+T$
Π_5	s	M	$(ncoo, pro_{1,1})$	3	$n+k+k'+3$	$2n+k+k'+3$
Π_6	s	1	$(ncoo, pro_{1,1})$	4	$n+3k+k'+4$	$2n+4k+k'+4$
Π_7	G	L	$(ncoo, pro_{1,1})$	3	$n+2k+3$	$n+3k+3+T$
Π_8	G	1	$(ncoo, pro_{1,1})$	4	$n+4k+4$	$n+6k+4+T$
Π_9	s	M	$(ncoo, inh_{1,1})$	2	$n+k+k'+2$	$2n+k+k'+2$
Π_{10}	s	1	$(ncoo, inh_{1,1})$	4	$n+3k+k'+4$	$2n+4k+k'+4$
Π_{11}	G	L	$(ncoo, inh_{1,1})$	3	$n+3k+4$	$n+4k+4+T$
Π_{12}	G	1	$(ncoo, inh_{1,1})$	5	$n+5k+6$	$n+6k+6+T$
Π_{13}	s	M	$(ncoo, antim/pri)$	2	$2n+k$	$3n+k$
Π_{14}	s	1	$(ncoo, antim/pri)$	4	$2n+5k+4$	$3n+5k+4$
Π_{15}	G	L	$(ncoo, antim/pri)$	3	$2n+4k+3$	$3n+3k+3+T$
Π_{16}	G	1	$(ncoo, antim/pri)$	5	$2n+8k+5$	$3n+7k+5+T$

Table 1. Comparative table of simulation of R systems by P systems

in S, k' is the number of symbols that do not appear in the left side of any rule of the simulated system; by T we denote $\sum_{1\leq i\leq k}(|A_i|+|B_i|)$ and by T' we denote $\sum_{1\leq i\leq k}|B_i|$). Column R describes the type of simulated system, where s stands for simple (rules with triples of symbols) and G stands for general (rules with triples of sets). Column mult describes the multiplicities of symbols in the simulating P system, where M stands for multiplicative effect, L stands for last multiplicity, and 1 stands for multiplicities 0 and 1. Column features describes the kinds of rules used.

Note: in Π_6, Π_8 and Π_9, intermediate objects are removed one step later, in parallel with the first step of simulation of the next step of evolution of the simulated R system, but not interfering with it.

Finally, in the previous section we discussed how (qualitatively and quantitatively) adopting some aspects of R systems (such as sets instead of multisets, auto-replication or no-persistence) into the working model of P systems simplifies simulation of R systems.

References

1. A. Alhazov. P systems without multiplicities of symbol-objects. *Information Processing Letters*, 100(3):124–129, 2006.
2. A. Alhazov, B. Aman, and R. Freund. P systems with anti-matter. In M. Gheorghe, G. Rozenberg, A. Salomaa, P. Sosík, and C. Zandron, editors, *Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August*

20-22, 2014, Revised Selected Papers, volume 8961 of *Lecture Notes in Computer Science*, pages 66–85. Springer, 2014.

3. A. Alhazov, S. Cojocaru, A. Colesnicov, L. Malahova, and M. Petic. A P system for annotation of Romanian affixes. In A. Alhazov, S. Cojocaru, M. Gheorghe, Yu. Rogozhin, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing - 14th International Conference, CMC 2013, Chişinău, Republic of Moldova, August 20-23, 2013, Revised Selected Papers*, volume 8340 of *Lecture Notes in Computer Science*, pages 80–87. Springer, 2013.
4. A. Alhazov and R. Freund. Asynchronous and maximally parallel deterministic controlled non-cooperative P systems characterize NFIN and coNFIN. In E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, and Gy. Vaszil, editors, *Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers*, volume 7762 of *Lecture Notes in Computer Science*, pages 101–111. Springer, 2012.
5. R. Brijder, A. Ehrenfeucht, M. G. Main, and G. Rozenberg. A tour of reaction systems. *International Journal of Foundations of Computer Science*, 22(7):1499–1517, 2011.
6. A. Ehrenfeucht and G. Rozenberg. Reaction systems. *Fundamenta Informaticae*, 75(1):263–280, 2007.
7. M. Margenstern, Yu. Rogozhin, and S. Verlan. Time-varying distributed H systems with parallel computations: The problem is solved. In J. Chen and J. H. Reif, editors, *DNA Computing, 9th International Workshop on DNA Based Computers, DNA9, Madison, WI, USA, June 1-3, 2003, revised Papers*, volume 2943 of *Lecture Notes in Computer Science*, pages 48–53. Springer, 2003.
8. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.
9. Gh. Păun. DNA computing based on splicing: universality results. *Theoretical Computer Science*, 231(2):275–296, 2000.
10. Gh. Păun and M. J. Pérez-Jiménez. Towards bridging two cell-inspired models: P systems and R systems. *Theoretical Computer Science*, 429:258–264, 2012.
11. Gh. Păun, G. Rozenberg, and A. Salomaa. Computing by splicing. *Theoretical Computer Science*, 168(2):321–336, 1996.
12. Gh. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing - New Computing Paradigms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 1998.
13. Gh. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.
14. D. Sburlan. Further results on P systems with promoters/inhibitors. *International Journal of Foundations of Computer Science*, 17(1):205–221, 2006.
15. The P Systems Website. `http://ppage.psystems.eu`.

Purely Catalytic P Systems over Integers and Their Generative Power

Artiom Alhazov[1], Omar Belingheri[2], Rudolf Freund[3],
Sergiu Ivanov[4], Antonio E. Porreca[2], and Claudio Zandron[2]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Str. Academiei 5, Chişinău, MD 2028, Moldova
E-mail: artiom@math.md
[2] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
E-mail: {o.belingheri@campus,porreca@disco,zandron@disco}.unimib.it
[3] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: rudi@emcc.at
[4] Université Paris Est, France
E-mail: sergiu.ivanov@u-pec.fr

Abstract. We further investigate the computing power of the recently introduced P systems with $\mathbb{Z}$-multisets (also known as hybrid sets) as generative devices. These systems apply catalytic rules in the maximally parallel way, even consuming absent non-catalysts, thus effectively generating vectors of arbitrary (not just non-negative) integers. The rules may only be made inapplicable by dissolution rules. However, this releases the catalysts into the immediately outer region, where new rules might become applicable to them. We discuss the generative power of this model. Finally, we consider the variant with mobile catalysts.

1 Introduction

Membrane systems (cell-like, with symbol objects) have traditionally been viewed as collections of hierarchically arranged multiset processors, e.g., see [12]. In the list of open problems disseminated in 2015, see [11], Gheorghe Păun suggested to go beyond the traditional setting where symbol multiplicities in multisets are restricted to non-negative integers. In [6] generalized multisets are defined as taking multiplicities from arbitrary finitely generated, totally ordered commutative groups.

In [3], a different approach is taken: only catalytic rules are allowed, and the applicability of a rule only depends on the presence of the corresponding catalyst in the given region. Consuming an absent non-catalyst makes its multiplicity negative. While in [3] it was already established that such a model is not universal, we found it interesting to investigate its generative power more precisely.

Since the number of catalysts remains finite and does not change throughout the computation, this induces a finite set of "rule teams" which can be applied in parallel in one step. The virtual absence of applicability conditions and the finiteness of the "teams" hints at the possibility of seeing them as integer vectors; in this case the P system itself can be seen as evolving by sequentially adding such vectors (possibly having negative components) to the contents of its membranes. In [2] this general model is compared with vector addition systems (see [5, 9] for standard definitions; adapted to allow negative vector components in [8]) and blind register machines, e.g., see [7].

Here we return to the particular model from [3], discussing the lower bound of its generative power and giving some results on the variant with target indications.

2 Preliminaries

The reader is assumed to be familiar with the basic notions of formal languages and membrane computing; see [13] for a comprehensive introduction to both. We only recall that multisets over the set of objects O can be seen as functions from O into $\mathbb{N}$; thus, the set of all multisets over O is $\mathbb{N}^O$ (the set of all functions from O to $\mathbb{N}$). In membrane computing, the set of all multisets over O is commonly denoted by $O^\circ = \mathbb{N}^O$, while the multisets themselves are represented by strings in O^*, keeping in mind that the order of symbols is not relevant.

2.1 Extending Multisets

To represent also negative multiplicities, multisets over a set of objects O have to be extended to $\mathbb{Z}$. A $\mathbb{Z}$-multiset, allowing integer multiplicities (called a *hybrid set* in [4]) then is from $\mathbb{Z}^O$; it can be represented by a string in $(O \cup O^-)^*$, where $O^- = \{a^- \mid a \in O\}$ is a set of symbols that represents objects in multiplicity "negative one". Note that, as opposed to P systems with matter-antimatter [1], symbol a^- here is not an actual object, but simply a convenient way to represent a deficit of a, and the actual multiplicity of a represented by a string w is $|w|_a - |w|_{a^-}$. We also do not distinguish between notations a^{-k} and $(a^-)^k$. The superscript $^-$ can be used as a morphism, producing a multiset with opposite multiplicities, e.g., $(a^k)^-$ represents the same $\mathbb{Z}$-multiset as a^{-k}. As the strings here are only used to represent $\mathbb{Z}$-multisets, we may write an equality sign between the strings representing the same $\mathbb{Z}$-multiset. For conciseness, let us use the notation $O^\bullet = (O \cup O^-)^*$. Finally, since it will be always clear from the context, we may call an element of $O^\bullet$ "multiset", omitting the word "representing". Assuming an order is fixed on O, for $u \in O^\bullet$, vector $(|u|_a - |u|_{a^-})_{a \in O}$ is denoted by $\psi_O(u)$; the subscript O may be omitted when it is clear from the context. This vector is called the *Parikh image* of u.

2.2 Linear Sets

The *linear* set generated by a set of vectors $A = \{\mathbf{a}_i \mid 1 \leq i \leq d\} \subset_{fin} \mathbb{Z}^n$ (here $A \subset_{fin} B$ indicates that A is a finite subset of B) and an offset $\mathbf{a}_0 \in \mathbb{Z}^n$ is defined as follows:

$$\langle A, \mathbf{a}_0 \rangle_{\mathbb{N}} = \left\{ \mathbf{a}_0 + \sum\nolimits_{i=1}^{d} k_i \mathbf{a}_i \mid k_i \in \mathbb{N},\ 1 \leq i \leq d \right\}.$$

If the offset $\mathbf{a}_0$ is the zero vector, we will call the corresponding linear set *homogeneous*; we also will use a short notation $\langle A \rangle_{\mathbb{N}} = \langle A, \mathbf{0} \rangle_{\mathbb{N}}$.

We use the notation $\mathbb{Z}^n LIN_{\mathbb{N}} = \{\langle A, \mathbf{a}_0 \rangle_{\mathbb{N}} \mid A \subset_{fin} \mathbb{Z}^n,\ \mathbf{a}_0 \in \mathbb{Z}\}$, to refer to the class of all linear sets. Semilinear sets are defined as finite unions of linear sets. We use the notations $\mathbb{Z}^n SLIN_{\mathbb{N}}$ to refer to the classes of semilinear sets of n-dimensional vectors. In case no restriction is imposed on the dimension, n is replaced by $*$. We may omit n if $n = 1$. A finite union of linear sets which only differ in the starting vectors is called *uniform* semilinear:

$$\mathbb{Z}^n SLIN_{\mathbb{N}}^{U} = \left\{ \bigcup\nolimits_{\mathbf{b} \in B} \langle A, \mathbf{b} \rangle_{\mathbb{N}} \mid A \subset_{fin} \mathbb{Z}^n,\ B \subset_{fin} \mathbb{Z}^n \right\}$$

Let us denote such a set by $\langle A, B \rangle_{\mathbb{N}}$.

Note that the uniform semilinear set can be $\langle A, B \rangle_{\mathbb{N}}$ seen as a pairwise sum of the finite set B and the homogeneous linear set $\langle A \rangle_{\mathbb{N}}$:

$$\langle A, B \rangle_{\mathbb{N}} = \{\mathbf{a} + \mathbf{b} \mid \mathbf{a} \in \langle A \rangle_{\mathbb{N}}, \mathbf{b} \in B\}.$$

This observation immediately yields the conclusion that the sum of two uniform semilinear sets $\langle A_1, B_1 \rangle_{\mathbb{N}}$ and $\langle A_2, B_2 \rangle_{\mathbb{N}}$ is uniform semilinear as well and can be computed in the following way:

$$\langle A_1, B_1 \rangle_{\mathbb{N}} + \langle A_2, B_2 \rangle_{\mathbb{N}} = \{\mathbf{a} + \mathbf{b}_1 + \mathbf{b}_2 \mid \mathbf{a} \in \langle A_1 \cup A_2 \rangle_{\mathbb{N}}, \mathbf{b}_1 \in B_1, \mathbf{b}_2 \in B_2\}.$$

3 Purely Catalytic P Systems over Integers

In purely catalytic P systems over integers the set of objects is a disjoint union of catalysts C and the regular objects O. The regular objects are allowed to have any integer multiplicity, while the catalysts are only allowed to appear in a non-negative number of copies.

The rules can be of the two following types:

- *catalytic rules*: $cu \to cv$, where $c \in C$ and $u, v \in O^*$;
- *catalytic rules with dissolution*: $cu \to cv\delta$, where $c \in C$, $u, v \in O^*$, and $\delta \notin C \cup O$ is the symbol indicating membrane dissolution.

The rules applied in parallel cannot involve more catalysts than available in the system; the multiplicities of regular objects, on the other hand, do not influence the applicability of rules. An application of a rule $cu \to cv$ in a region containing cw ($c \in C$, $u, v \in O^*$, $w \in O^\bullet$ produces $cw(cu)^- cv = cwv(u^-)$, or, in

terms of vectors, ignoring the catalyst, vector $\psi(w)+\psi(v)-\psi(u)$ represents the contents of that region after the rule has been applied. An application of a rule $cu \to cv\delta$ produces the same effect, and then dissolves the enclosing membrane, moving the contents of the dissolved membrane into the parent membrane.

Purely catalytic P systems over integers evolve under the maximally parallel semantics, so each catalyst induces the application of exactly one rule (non-deterministically chosen), unless the given region has no rules associated with this catalyst. By $\mathbb{Z}^d O_{\mathbb{Z}} P_m(pcat_k, \delta)$ we denote the family of sets of d-dimensional vectors of integers generated by purely catalytic P systems over integers with dissolution, at most m membranes and at most k catalysts. If any of parameters d, m, k is unbounded, it is replaced by $*$ in this notation.

We also use notations for extended features (listed in parentheses in the notation of the sets of $\mathbb{Z}$-vectors generated by the corresponding families of P systems). Target indications, denoted by tar, allow the non-catalysts to be sent to a different membrane. In the right side of the rules, sending object a is written by (a, tar), where $tar \in \{out\} \cup \{in_j \mid 1 \leq j \leq m\}$; j here is the label of an immediately inner membrane. In this paper, we may write $tar_{\mathbb{Z}}$ in the notation of a set of $\mathbb{Z}$-vectors generated by a family of P systems; this generalization reflects the possibility to assign targets even to negative multiplicites of objects.

Another feature is *mobile catalysts* [10], i.e., targets may also be associated to the catalysts, and thus the catalysts move across the membrane structure; we denote this feature by $mpcat_k$ since the systems we consider are purely catalytic. We use the plus sign between the features of catalytic mobility and dissolution when it is allowed for the *same* rule to move a catalyst and to dissolve the membrane currently containing it.

4 Results

4.1 Simplifications and Observations

First, we would like to explicitly allow rules of the form $c \to cx$, $(c \in C, x \in O^\bullet)$, i.e., the multiset of regular objects in the left side being empty. This does not change the model, since any $\mathbb{Z}$-multiset x can be written as $u(v^-)$, $u, v \in O^*$, and, fixing some $a \in O$, $c \to cx$ is equivalent to $cau \to av$. Moreover, any rule $cu \to cv$ is equivalent to $c \to cu(v^-)$, so it suffices to only consider rules of types $c \to cx$ and $c \to cx\delta$ $(c \in C, x \in O^\bullet)$.

Second, we claim that it is enough to start with a single catalyst in every region. To show that, we will consider that membrane i of the P system contains the catalysts $c_{i,k}$, $1 \leq k \leq n_i$, in the *initial* configuration. Now we will define the sets $X_{i,k,j}$ of right-hand sides of catalytic rules of membrane j involving the catalysts initially located in membrane i:

$$X_{i,k,j} = \{x \in O^\bullet \cup O^\bullet\delta \mid (c_{i,k} \to c_{i,k}x) \in R_j\},$$

where R_j is the set of rules associated with membrane j. To simplify subsequent explanations, we will adopt the following convention: If, for a given i and j, there exists such a k that $X_{i,k,j} \neq \emptyset$, then we will replace all empty $X_{i,k',j}$ by $\{\lambda\}$.

We remark now that the catalysts $c_{i,k}$ initially present in membrane i will always stay together, because dissolution cannot separate them. We will replace each such group ("band") by a new catalyst having the combined effect of the group. More formally, we will replace all the catalysts $c_{i,k}$ initially present in membrane i by one new catalyst c_i, and the rules associated with membrane j by the following set:

$$R'_j = \{c_i \to c_i x_1 \cdots x_{n_i} \mid x_k \in X_{i,k,j}, 1 \leq k \leq n_i, 1 \leq i \leq m\},\ 1 \leq j \leq m,$$

where m is the number of membranes of the membrane system. Note that R'_j contains rules for every catalyst c_i representing the original "band" from membrane i which may have an effect in region j.

The argument in the previous paragraph shows that we can replace multiple catalysts in a region by a single one. On the other hand, having no catalyst in some region is equivalent to having one catalyst with no associated rules. Therefore, without restricting the generality, in the following we assume that in the initial configuration of an arbitrary purely catalytic P system over integers, each membrane region i, $1 \leq i \leq m$, contains precisely one catalyst, and we can call it c_i.

Third, notice that the symbols may only travel from the inner membranes to the outer ones, so if the output region i_0 is not the skin, only the contents of the membrane substructure inside i_0 (including i_0) is relevant for the result. The only way in which a membrane i not contained within i_0 could influence the evolution of the system is by preventing it from halting. If i halts after i_0, but in a finite number of steps, then this only influences the moment when we are allowed to retrieve the result, but not the result itself. If i never halts, then the result of the system is always empty. If i may choose between halting in a finite number of steps or never halting, then we can only consider those computations in which it halts, and, as we have just shown, in this case it does not have any influence on the result of the system.

According to this reasoning, the membranes not contained within the output membrane i_0 may influence the power of the system only trivially (by reducing its result to the empty set). We will therefore assume that the output region is always the skin.

Fourth, every elementary membrane having no rules associated to the catalysts available there may be removed from the system without affecting the result (unless it is the output membrane, in which case a singleton is generated, which is a degenerate case), so in the following we assume that each elementary membrane has some applicable rules. Clearly, the P system will not reach the halting until this membrane is dissolved.

Consider this reasoning starting from the elementary membranes, by induction. Take any non-elementary membrane i which becomes elementary during a computation. Assume i is not dissolved (i.e., it has no rules associated to any of the catalysts that were placed within the membrane substructure inside i, including i), but it is not the output membrane. Then all the computations in the membrane substructure inside i, including i, do not contribute to the result, and can be removed from the system without affecting the result.

As a summary of the fourth observation, without restricting the generality (except, possibly the degenerate cases generating the empty set or some singleton), we may assume that any purely catalytic P system over integers has applicable rules associated to all elementary membranes, and all membranes except the skin must be dissolved at some moment during the computation.

Finally, for every region except the skin, a catalyst c_i without associated rules is equivalent to a catalyst with a rule $c_i \to c_i$. Hence, without restricting the generality, we may assume that the catalysts are *never* idle before the halting is reached. Clearly, (excluding the degenerate case generating the empty set), the skin should have no rules associated to any catalyst of the system.

We would like to note that even without pruning the membrane structure by removing membrane substructures not contributing to the result, the membrane structure obtained at halting (if at all reachable) is unique.

We recall that in [2], the following generalization approach is taken: There is a finite number of reachable membrane structures. These could be used as states of a sequential P system, which may be obtained, separately for each membrane structure, by combining the behavior of all catalysts in all regions of the P system. Indeed, having fixed a reachable membrane structure, we know which membranes have been dissolved, and thus the resulting location of each catalyst. Then, for each catalyst, associated rules in its current location are considered and combined, similarly to the second observation above, but globally. Having obtained a sequential system, the catalyst is no longer needed. Then, in [2] it was shown that such a generalization is nothing else but a sequential blind vector addition system with states, and it was claimed that it characterizes precisely the family of all semilinear vectors of integers.

Indeed, in this way any purely catalytic P system over integers can be substituted by a sequential blind vector addition system with states, so the upper bound of the family of all semilinear sets of vectors of integers, or, equivalently, the family of all integer vector sets, generated by blind register machines, holds. However, the reverse is not necessarily true, i.e., it does not follow that for any sequential blind vector addition system with states there would exist an equivalent purely catalytic P system over integers.

In the present paper we investigate the particularities of how dissolution affects the computation, and the lower bounds.

4.2 Generative Power

We recall that we discuss the family of integer vector sets generated by purely catalytic P systems over integers, with the usual halting condition.

Since the output region cannot be dissolved by definition and any other applicable rule can never be stopped, single-membrane purely catalytic P systems over integers are degenerate:

$$Z^d O_{\mathbb{Z}} P_1(pcat_*, \delta) = \{\emptyset\} \cup \{\{\mathbf{a}\} \mid \mathbf{a} \in \mathbb{Z}^d\}.$$

For simplicity, we will not mention these degenerate cases while considering multiple membranes.

With two membranes, a characterization is still straightforward:

$$Z^dO_{\mathbb{Z}}P_2(pcat_*, \delta) = \mathbb{Z}^dSLIN^U_{\mathbb{N}}.$$

Indeed, let A be the finite set of vectors corresponding to the non-dissolving rules in the elementary membranes, and let B be the finite set of sums of two vectors: the one corresponding to the initial configuration and vectors corresponding to the dissolving rules in the elementary membrane; the skin should have no rules. If the catalyst in the elementary membrane is c_2, then the correspondence mentioned above is $c_2 \to c_2x \leftrightarrow \psi(x)$, and similarly with dissolution. An arbitrary computation of a P system consists of an arbitrary number of applications of non-dissolving rules and one application of a dissolving rule. Hence, the resulting vector sums up from the "initial" vector, one arbitrary "dissolving" vector, and an arbitrary linear combination of "non-dissolving" vectors.

It is worth noting that, by a similar reasoning, for a P system with multiple membranes, if the chronological order of dissolving membranes is fixed, the result is still $\mathbb{Z}^dSLIN^U_{\mathbb{N}}$. Indeed, each combination of rules (one for each catalyst) yields one vector, so all such possible combinations of non-dissolving rules yield a finite set of vectors, and multiple non-dissolving steps yield a linear set generated by these vectors. Thus, over the whole computation the result sums up from the initial configuration, a finite number of dissolution vectors, and a finite number of linear sets corresponding to the membrane structures reached during that computation. Since the total number of chronological orders of dissolving membranes is bounded, the known result already follows:

$$Z^dO_{\mathbb{Z}}P_*(pcat_*, \delta) \subseteq \mathbb{Z}^dSLIN_{\mathbb{N}}.$$

Even with three membranes, in case two of them are elementary, the power of such purely catalytic P systems over integers is still $\mathbb{Z}^dSLIN^U_{\mathbb{N}}$, but for a different reason: each elementary membrane contributes with its uniform semilinear set, and a sum of two uniform semilinear sets is still uniform semilinear.

Let us now examine a P system with three nested membranes – the minimal number to obtain a set which is not in $\mathbb{Z}^dSLIN^U_{\mathbb{N}}$. Let the vector obtained by joining the initial contents of all membranes be $\mathbf{a}$, the set of non-dissolving vectors of the elementary membrane be A_3, the set of dissolving vectors of the elementary membrane be B_3, the sets of non-dissolving and dissolving vectors in the middle membrane associated to catalyst c_2 are A_2 and B_2, and the similar sets associated to catalyst c_3 (which will arrive from the elementary membrane) are A and B. Let us see what the resulting vector set is built from, besides $\mathbf{a}$.

A non-dissolving computation in three membranes adds at each step (an element of) A_3 to the elementary membrane and (an element of) A_2 to the middle membrane. Eventually all objects will arrive to the skin, so the three-membrane phase of the computation will contribute by (an arbitrary element of) $\langle A_2 + A_3 \rangle_{\mathbb{N}}$.

Then there are two possibilities. If membrane 2 is dissolved first, then the system continues computing by only applying the rules in membrane 3, and eventually dissolving membrane 3, yielding $B_2 + \langle A_3 \rangle_{\mathbb{N}} + B_3$. However, if membrane 3 is dissolved first, then both catalysts are active in membrane 2, eventually dissolving it, yielding $B_3 + \langle A_2 + A \rangle_{\mathbb{N}} + (B_2 + A \cup A_2 + B \cup A + B)$. The expression in parentheses corresponds to applying at least one dissolving rule. Therefore, the set of integer vectors generated by such a purely catalytic P system over integers with three nested membranes is

$$M = \mathbf{a} + B_3 + \langle A_2 + A_3 \rangle_{\mathbb{N}} + \Big(B_2 + \langle A_3 \rangle_{\mathbb{N}} \cup \langle A_2 + A \rangle_{\mathbb{N}} + \big(B_2 + A \cup A_2 + B \cup B_2 + B \big) \Big),$$

and the power of all three-membrane purely catalytic P systems over integers, noting that the power of the nested case subsumes the power of the case with two elementary membranes, is

$$Z^d O_{\mathbb{Z}} P_3(pcat_*, \delta) = \{ M \mid \mathbf{a} \in \mathbb{Z}^d,\ A_2, A_3, B_2, B_3, A, B \in FIN(\mathbb{Z}^d) \},$$

where M is the expression above. Unfortunately, it is not obvious what can be simplified in it, except B_3 can subsume $\mathbf{a}$. So we try to analyze it in details, possibly going into particular cases.

All terms in the expression M are bounded except three: $\langle A_3 + A_2 \rangle_{\mathbb{N}}$, $\langle A_3 \rangle_{\mathbb{N}}$ and $\langle A + A_2 \rangle_{\mathbb{N}}$. These terms are not independent, even though A_2, A_3 and A are three independent finite sets of vectors. It is, however, possible to separate them in a particular case when $|A_3| = 1$, choosing $A_2 = -A_3$ and $A = C - A_2$. Since A_3 is a singleton, the identity $A_3 - A_3 = \{\mathbf{0}\}$ holds, so the three unbounded terms become $\langle \{\mathbf{0}\} \rangle_{\mathbb{N}}$, $\langle A_3 \rangle_{\mathbb{N}}$ and $\langle C \rangle_{\mathbb{N}}$, so we are getting close to obtaining a union of two particular linear (or even uniform semilinear) sets with different base vectors.

Indeed, if we choose $\mathbf{a} = \mathbf{0}$, $B_3 = \{\mathbf{0}\}$, $B_2 = \{\mathbf{0}\}$, $B = \{\mathbf{0}\}$ and $A_3 = \{\mathbf{e}\}$, expression M simplifies to $\langle \{\mathbf{e}\} \rangle_{\mathbb{N}} \cup \langle C \rangle_{\mathbb{N}} + (C + \{\mathbf{e}\} \cup \{\mathbf{0}\})$, which can be rewritten as $\langle \{\mathbf{e}\} \rangle_{\mathbb{N}} \cup \langle C \rangle_{\mathbb{N}} \cup \{\mathbf{e}\} \langle C \rangle_{\mathbb{N}}$.

Alternatively, to avoid dealing with the union of three cases when membrane 2 is divided last, if we choose $B_2 = A_2$ and $B = A$, then the last parenthesis in the general expression of set M becomes simply $A_2 + A = C$. Choosing $\mathbf{a} = \mathbf{0}$, $B_3 = \{\mathbf{0}\}$, and $A_3 = \{\mathbf{e}\}$, expression M simplifies to $\langle \{\mathbf{e}\} \rangle_{\mathbb{N}} - \{\mathbf{e}\} \cup \langle C \rangle_{\mathbb{N}} + C$. Since $\mathbf{0} \in \langle \{\mathbf{e}\} \rangle_{\mathbb{N}} - \{\mathbf{e}\}$ and $\langle C \rangle_{\mathbb{N}} + C \cup \{\mathbf{0}\} = \langle C \rangle_{\mathbb{N}}$, in this case we can rewrite M to

$$-\{\mathbf{e}\} \cup \langle \{\mathbf{e}\} \rangle_{\mathbb{N}} \cup \langle C \rangle_{\mathbb{N}},$$

which is a union of any two homogeneous linear sets, such that the first one has only one generator, united with the opposite vector of that generator. Hence,

$$Z^d O_{\mathbb{Z}} P_n(cat, \delta) \supsetneq Z^d SLIN^U_{\mathbb{N}},\ n \geq 3.$$

What if $B = \emptyset$, i.e., catalyst c_3 has no associated dissolution rules in region 2? Then the general expression of set M is immediately simplified to

$$M = \mathbf{a} + B_2 + B_3 + \langle A_2 + A_3 \rangle_{\mathbb{N}} + (\langle A_3 \rangle_{\mathbb{N}} \cup \langle A_2 + A \rangle_{\mathbb{N}} + A),$$

and in our case of $A_3 = \{\mathbf{e}\}$, $A_2 = -\{\mathbf{e}\}$ and $A = C + \{\mathbf{e}\}$, M becomes

$$\mathbf{a} + B_2 + B_3 + (\langle\{\mathbf{e}\}\rangle_{\mathbb{N}} \cup \langle C\rangle_{\mathbb{N}} + C + \{\mathbf{e}\}),$$

and choosing $\mathbf{a} + B_2 + B_3 = \{-\mathbf{e}\}$, and noticing that C 0 times is covered by $\mathbf{e}$ 0 times and $\langle C\rangle_{\mathbb{N}} + C \cup \{\mathbf{0}\} = \langle C\rangle_{\mathbb{N}}$, we simplify M to $\{-\mathbf{e}\} \cup \langle\{\mathbf{e}\}\rangle_{\mathbb{N}} \cup \langle C\rangle_{\mathbb{N}}$, i.e., an "almost clean union" we already obtained before. Finally, we notice that we can equivalently write it as

$$\langle\{\mathbf{e}\}, -\mathbf{e}\rangle_{\mathbb{N}} \cup \langle C\rangle_{\mathbb{N}}\,.$$

Continuing the current approach with more membranes would only result in more cases.

4.3 Communication

We would like to remark that adding target indications to the regular objects should not increase the power of purely catalytic P systems over integers. Indeed, looking at a purely catalytic P system over integers, it is easily decidable which membranes will eventually be dissolved. Hence, the only question is whether the contents of a region specified by target, after possible dissolutions, will be in the output. There is no need to examine the future of a moved regular object, since the resources in purely catalytic P systems over integers are unbounded, and we can view this copy of a moved object as staying in that region until the end of the computation.

However, if also the catalysts are allowed to have target indications associated, it does make a difference. We claim the following characterizations.

$$Z^dO_{\mathbb{Z}}P_*(mpcat_k, tar_{\mathbb{Z}}) = \mathbb{Z}^dSLIN_{\mathbb{N}},\ k \geq 1,$$
$$Z^dO_{\mathbb{Z}}P_*(mpcat_k + \delta) = \mathbb{Z}^dSLIN_{\mathbb{N}},\ k \geq 1,$$
$$Z^dO_{\mathbb{Z}}P_*(mpcat_*, \delta) = \mathbb{Z}^dSLIN_{\mathbb{N}},$$

The upper bound in either case is easy to see because the number of possible arrangements of catalysts across the given membrane structure (and any possible structures obtained from it by membrane dissolutions) is bounded. Hence, purely catalytic P systems over integers with mobile catalysts are still not more powerful than blind vector-addition systems with states, which characterize $\mathbb{Z}^*SLIN_{\mathbb{N}}$, see [2]. We now proceed to $\supseteq$ inclusions.

Consider an arbitrary semilinear set $\bigcup_{1\leq i\leq m}\langle A_i, \mathbf{b}_i\rangle_{\mathbb{N}}$, where for each i, $1 \leq i \leq m$, A_i is a finite set, $A_i \cup \{\mathbf{b}_i\} \subseteq \mathbb{Z}^d$. We construct the following purely

catalytic P system over integers

$$\begin{aligned}
\Pi_1 &= (O, C, \mu, w_1, \cdots, w_{2m+1}, R_1, \cdots, R_{2m+1}, i_0 = 1) \text{ where} \\
O &= \{a_i \mid 1 \le i \le d\},\ C = \{c\}, \\
\mu &= [\,[\,[\,\,]_{m+2}\,]_2 \cdots [\,[\,\,]_{2m+1}\,]_{m+1}\,]_1, \\
w_1 &= c,\ w_{i+1} = \lambda,\ 1 \ge i \ge 2m, \\
R_1 &= \{c \to (c, in_{i+1})v_i \mid 1 \le i \le m,\ \psi(v_i) = \mathbf{b}_i\}, \\
R_{i+1} &= \{c \to c(v, out) \mid \psi(v) \in A_i\} \cup \{c \to (c, in_{m+i+1})\},\ 1 \le i \le m, \\
R_{m+i+1} &= \emptyset,\ 1 \le i \le m.
\end{aligned}$$

The work of Π_1 consists of a non-deterministic choice of i-th linear set to generate, by moving catalyst c into membrane $i+1$ and producing $\mathbf{b}_i$. After sending to the skin an arbitrary combination of vectors from A_i, the catalyst enters membrane $m+i+1$ and the system halts.

The system Π_2 is obtained from Π_1 by replacing the sets R_{i+1} of rules, $1 \le i \le m$, by

$$\{c \to cv \mid \psi(v) \in A_i\} \cup \{c \to (c, in_{m+i+1})\delta\}.$$

It works just as Π_1, with one difference, Here, instead of sending v out (possibly containing negative multiplicities), the linear combination of vectors from A_i is generated directly in membrane $i+1$, and is released into the skin upon dissolution of membrane $i+1$, simultaneously with sending the catalyst into the elementary membrane $m+i+1$. Now consider the following purely catalytic P system over integers.

$$\begin{aligned}
\Pi_3 &= (O, C, \mu, w_1, \cdots, w_{3m+1}, R_1, \cdots, R_{3m+1}, i_0 = 1) \text{ where} \\
O &= \{a_i \mid 1 \le i \le d\},\ C = \{c_i \mid 1 \le i \le m+1\}, \\
\mu &= [\,[\,[\,[\,\,]_{2m+2}\,]_{m+2}\,]_2 \cdots [\,[\,[\,\,]_{3m+1}\,]_{2m+1}\,]_{m+1}\,]_1, \\
w_1 &= c_1,\ w_{i+1} = \lambda,\ 1 \le i \le 2m, \\
w_{2m+1+i} &= c_{1+i},\ 1 \ge i \ge m, \\
R_1 &= \{c_1 \to (c_1, in_{i+1})v_i \mid 1 \le i \le m,\ \psi(v_i) = \mathbf{b}_i\}, \\
R_{i+1} &= \{c_1 \to c_1 v \mid \psi(v) \in A_i\} \\
&\quad \cup \{c_1 \to (c_1, in_{m+i+1}),\ c_i \to c_i\delta\},\ 1 \le i \le m, \\
R_{m+i+1} &= \{c_1 \to (c_1, in_{2m+i+1}),\ c_i \to (c_i, out)\},\ 1 \le i \le m, \\
R_{2m+i+1} &= \{c_1 \to c_1\delta\},\ 1 \le i \le m.
\end{aligned}$$

The basic idea is the same, but the implementation is a little longer. To each linear set i, $1 \le i \le n$, three nested membranes are associated ($i+1$, $m+i+1$ and $2m+i+1$). The beginning is just like in the case of Π_2, until catalyst c_1 is sent into membrane $m+i+1$, but membrane $i+1$ is not dissolved yet.

Then, c_1 enters the elementary membrane $2m+i+1$ and dissolves it, releasing catalyst c_{i+1} into the surrounding membrane $m+i+1$. Clearly, c_1 cannot reenter membrane $2m+i+1$, which no longer exists, so it has no applicable associated rules. Catalyst c_i, however, is sent out to membrane $i+1$, and dissolves it, which releases all generated regular objects to the skin and halts the computation. This proves the characterizations.

5 Conclusions

We have reproved that the power of purely catalytic P systems over integers is contained in the family of all semilinear sets of vectors of integers. We then have shown that with one membrane purely catalytic P systems over integers give degenerate results, and with two membranes they are characterized exactly by the family of all uniform semilinear sets of vectors of integers. With more membranes, this equality becomes a strict inclusion, and a specific union of linear sets with different base vectors have been obtained. More specifically, for any vector $\mathbf{e} \in \mathbb{Z}^d$ and any finite set $C \subseteq \mathbb{Z}^d$, purely catalytic P systems over integers can generate

$$\langle \{\mathbf{e}\}, -\mathbf{e} \rangle \cup \langle C \rangle_{\mathbb{N}} .$$

The most interesting open question remaining is whether $Z^*O_{\mathbb{Z}}P_*(pcat_*, \delta)$ is closed under union. While in almost all cases in membrane computing closure under union is trivial, e.g., by making a non-deterministic choice in the first step of the computation, the current situation is rather surprising.

Finally, we have considered the variants with mobile catalysts, and showed a few combinations of features leading to characterizations of semilinear sets of $\mathbb{Z}$-vectors.

References

1. A. Alhazov, B. Aman, R. Freund, and Gh. Păun. Matter and anti-matter in membrane systems. In H. Jürgensen, J. Karhumäki, and A. Okhotin, editors, *Descriptional Complexity of Formal Systems: 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, pages 65–76. Springer, 2014.
2. A. Alhazov, O. Belingheri, R. Freund, S. Ivanov, A. E. Porreca, and C. Zandron. Semilinear sets, register machines, and integer vector addition (P) systems. In *17th International Conference on Membrane Computing (this volume)*, 2016.
3. O. Belingheri, A. E. Porreca, and C. Zandron. P systems with hybrid sets, 2016. Workshop on Membrane Computing *(UCNC 2016)*, submitted.
4. J. Carette, A. P. Sexton, V. Sorge, and S. M. Watt. Symbolic domain decomposition. In S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton, editors, *Intelligent Computer Mathematics, 10th International Conference, AISC 2010, 17th Symposium, Calculemus 2010, and 9th International Conference, MKM 2010, Paris, France, July 5-10, 2010. Proceedings*, volume 6167 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2010.

5. R. Freund, O. Ibarra, Gh. Păun, and H.-C. Yen. Matrix languages, register machines, vector addition systems. *Third Brainstorming Week on Membrane Computing*, pages 155–167, 2005.
6. R. Freund, S. Ivanov, and S. Verlan. P systems with generalized multisets over totally ordered abelian groups. In *Int. Conf. on Membrane Computing*, volume 9504 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2015.
7. S. A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7(3):311–324, 1978.
8. C. Haase and S. Halfon. Integer vector addition systems with states. In J. Ouaknine, I. Potapov, and J. Worrell, editors, *Reachability Problems: 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, pages 112–124. Springer, 2014.
9. J. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.
10. S. N. Krishna and A. Păun. Results on catalytic and evolution-communication P systems. *New Generation Computing*, 22(4):377–394, 2004.
11. Gh. Păun. Some quick research topics. `http://www.gcn.us.es/files/OpenProblems_bwmc15.pdf`.
12. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.
13. Gh. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.

Semilinear Sets, Register Machines, and Integer Vector Addition (P) Systems

Artiom Alhazov[1], Omar Belingheri[2], Rudolf Freund[3],
Sergiu Ivanov[4], Antonio E. Porreca[2], and Claudio Zandron[2]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Str. Academiei 5, Chişinău, MD 2028, Moldova
E-mail: artiom@math.md
[2] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
E-mail: {o.belingheri@campus,porreca@disco,zandron@disco}.unimib.it
[3] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Vienna, Austria
E-mail: rudi@emcc.at
[4] Université Paris Est, France
E-mail: sergiu.ivanov@u-pec.fr

Abstract. In this paper we consider P systems working with multisets with integer multiplicities. We focus on a model in which rule applicability is not influenced by the contents of the membrane. We show that this variant is closely related to blind register machines and integer vector addition systems. Furthermore, we describe the computational power of these models in terms of linear and semilinear sets of integer vectors.

1 Introduction

P systems have traditionally been viewed as hierarchical processors of multisets, for instance, see [11]. In the list of open problems disseminated in 2015, see [10], Gheorghe Păun suggested to go beyond the traditional setting and to consider multisets in which objects are not restricted to have non-negative multiplicities. Several possible approaches have been suggested since then, including the one from [3], which defines generalised multisets as taking multiplicities from finitely generated, totally ordered Abelian groups.

In [1], a different approach is taken. The objects of the P system are partitioned into two classes: regular objects, which may have any integer multiplicity, and "catalysts", which may only appear in a bounded number of copies and cannot be consumed without being immediately reproduced. Thus, the regular objects cannot influence the applicability of rules, while the always bounded catalysts induce a finite set of "rule teams" which can be applied in parallel in one step. The virtual absence of applicability conditions and the finiteness of the "teams" hints at the possibility of seeing them as integer vectors; in this case

the P system itself can be seen as evolving by sequentially adding such vectors to the contents of its membranes.

Even though this vision is quite reminiscent of the folklore vector addition systems (VAS), this latter model is actually limited to having natural vectors as configurations [2, 8]. On the other hand, P systems manipulating integer multisets allow symbols with negative multiplicities to appear. It turns out that vector addition systems *without* the limitation of having natural configurations (integer VAS) have received relatively little attention in the literature, for example, see [7].

Another related model which has received notoriously little attention are the blind register machines, whose registers are allowed to range over the whole set of integers. Blind counter automata have been introduced and studied as string recogniser devices by Sheila Greibach in [6]; their adaptation to recognising vectors of integer numbers seems quite relevant to the study of multisets with integer multiplicities.

In the present work we bring together the three models – P systems over integer multisets as defined in [1], integer vector addition systems, and blind register machines – and formally show the connections between their different variants. We also give detailed characterisations of their computing power in terms of linear and semilinear sets of natural and integer vectors.

The article is structured as follows. Section 2 recalls some notions used throughout the paper, in particular semilinear sets and vector addition systems. Section 3 gives a general definition of a register machine over a set A, and then defines blind, partially blind, and conventional register machines within this general framework. Section 4 defines the model of integer vector addition P systems and gives some details as to their semantics. Section 5 investigates the power of blind register machines and gives characterisations in terms of semilinear sets of vectors. Finally, Section 6 studies the power of integer vector addition systems with and without membranes, and compares different variants of the models between themselves and with blind register machines.

2 Preliminaries

The reader is assumed to be familiar with the basic notions of formal languages and membrane computing; see [12] for a comprehensive introduction to both.

We only recall the definition of multisets over a commutative monoid M, i.e., a commutative semi-group with unit element e:

Definition 1. *An* $\mathbb{M}$-multiset *over the (finite) alphabet* O *is a mapping* $w : O \to \mathbb{M}$. *The value* $w(a)$ *is called the* multiplicity *of* a *in* w. *An object* $a \in O$ *is said to* appear in w *if* $w(a) \neq e$. *A multiset* w *is said to be* empty *if no objects appear in it, i.e., if* $w(a) = e$ *for all* $a \in O$.

For example, $\mathbb{Z}$-multisets can be seen as *vectors* of integers, indexed by elements of O. For the empty $\mathbb{Z}$-multiset over O all objects have multiplicity 0. We will use the notation $\mathbb{Z}^O$ to refer to the set of all $\mathbb{Z}$-multisets over O.

2.1 Linear Sets

The $\mathbb{N}$-*linear* set of $\mathbb{Z}$-vectors (or just *linear set* of $\mathbb{Z}$-vectors) generated by a set of vectors $A = \{\mathbf{a}_1, \cdots, \mathbf{a}_d\} \subset \mathbb{Z}^n$ and an offset $\mathbf{a}_0 \in \mathbb{Z}^n$ is defined as follows:

$$\langle A, \mathbf{a}_0 \rangle_{\mathbb{N}} = \left\{ \mathbf{a}_0 + \sum_{i=1}^{d} k_i \mathbf{a}_i \;\middle|\; k_i \in \mathbb{N},\ 1 \leq i \leq d \right\}.$$

We underline that the vectors are over $\mathbb{Z}$, but the coefficients are from $\mathbb{N}$ (we will also consider the special case when $A \cup \{\mathbf{a}_0\} \subset \mathbb{N}^n$; this then is an $\mathbb{N}$-*linear* set of $\mathbb{N}$-vectors, or just a linear set, a well-known concept from Formal Language Theory).

A $\mathbb{Z}$-linear set of $\mathbb{Z}$-vectors

$$\langle A, \mathbf{a}_0 \rangle_{\mathbb{Z}} = \left\{ \mathbf{a}_0 + \sum_{i=1}^{d} k_i \mathbf{a}_i \mid k_i \in \mathbb{Z},\ 1 \leq i \leq d \right\}$$

can be considered, too. It corresponds precisely to the *linear vector space* notion from the classic course of Linear Algebra. However, it is also a particular case. Indeed, it is easy to see that $\langle A, \mathbf{a}_0 \rangle_{\mathbb{Z}} = \langle B, \mathbf{a}_0 \rangle_{\mathbb{N}}$ for

$$B = \{\mathbf{a}_1, \cdots, \mathbf{a}_d, -\mathbf{a}_1, \cdots, -\mathbf{a}_d\}.$$

If the offset $\mathbf{a}_0$ is the zero vector, we call the corresponding linear set *homogeneous*.

A *positive-restricted* $\mathbb{N}$-linear set of $\mathbb{Z}$-vectors generated by A and an offset $\mathbf{a}_0$ is as the $\mathbb{N}$-linear set of $\mathbb{Z}$-vectors generated by A, restricted to non-negative vectors only:

$$\langle A, a_0 \rangle_{\mathbb{Z}}^{+} = \{\mathbf{x} \in \langle A, a_0 \rangle_{\mathbb{N}} \mid \mathbf{x} \geq 0\},$$

where $\mathbf{x} \geq 0$ means that every component of $\mathbf{x}$ is non-negative.

We will use the notations $\mathbb{Z}^n LIN_{\mathbb{N}}$, $\mathbb{N}^n LIN_{\mathbb{N}}$, $\mathbb{Z}^n LIN_{\mathbb{Z}}$, and $\mathbb{Z}_+^n LIN_{\mathbb{N}}$ to refer to the classes of all $\mathbb{N}$-linear sets of $\mathbb{Z}$-vectors, $\mathbb{N}$-linear sets of $\mathbb{N}$-vectors, $\mathbb{Z}$-linear sets of $\mathbb{Z}$-vectors, and positive restricted $\mathbb{N}$-linear sets of $\mathbb{Z}$-vectors of dimension n, correspondingly. Semilinear sets are defined as finite unions of the corresponding types of linear sets. We will use the notations $\mathbb{Z}^n SLIN_{\mathbb{N}}$, $\mathbb{N}^n SLIN_{\mathbb{N}}$, $\mathbb{Z}^n SLIN_{\mathbb{Z}}$, and $\mathbb{Z}_+^n SLIN_{\mathbb{N}}$ to refer to the families of $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors, $\mathbb{N}$-semilinear sets of $\mathbb{N}$-vectors, $\mathbb{Z}$-semilinear sets of $\mathbb{Z}$-vectors, and positive-restricted $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors of dimension n, respectively. In case no particular restriction is imposed on the dimension, n will be replaced by $*$. We may omit n if $n = 1$.

We recall the following general result from number theory known as *Bézout's identity*. Given a set of integers $A = \{a_1, \cdots, a_n\} \subset \mathbb{Z}$, there exist integers $x_1, \cdots, x_n \in \mathbb{Z}$ such that the following holds:

$$\sum_{i=1}^{n} x_i a_i = \gcd(a_1, \cdots, a_n),$$

where $\gcd(a_1, \ldots, a_n)$ is the greatest common divisor of the integers from A. Furthermore, the greatest common divisor is the smallest positive integer which can be obtained as a ($\mathbb{Z}$-)linear combination of the elements of A.

2.2 Vector Addition Systems

A *vector addition system* (VAS) of dimension $n \in \mathbb{N}$ is defined to be the pair $(\mathbf{w}_0, W)$, where $\mathbf{w}_0 \in \mathbb{N}^n$ is the start vector, and W is a finite set of vectors from $\mathbb{Z}^n$, called addition vectors. An addition vector $\mathbf{w} \in W$ is said to be applicable to a vector $\mathbf{x} \in \mathbb{N}^n$ if $\mathbf{x} + \mathbf{w} \in \mathbb{N}^n$, i.e., if all the components of the vector $\mathbf{x} + \mathbf{w}$ are non-negative. A VAS evolves from the start vector $\mathbf{w}_0$ by sequentially adding an applicable addition vector from W in each step.

A *vector addition system with states* (VASS) is a VAS equipped with a finite state control. Essentially, state labels are assigned to addition vectors and a graph of states is given which defines the possible sequences of application of addition vectors.

We will use the notation *VAS* and *VASS* to refer to the families of sets of natural vectors which can be generated by VAS and VASS, respectively.

It was shown in [8] that VASS are equivalent in expressive power to VAS (without states): any n-dimensional VASS can be simulated by an $(n + 3)$-dimensional VAS.

A variation of the model of vector addition systems consists in lifting the restriction that the valid vectors must have non-negative components. This model has recently been defined in [7].

An *integer vector addition system* ($\mathbb{Z}$-VAS) of dimension $n \in \mathbb{N}$ is the pair $(\mathbf{w}_0, W)$, where $\mathbf{w}_0 \in \mathbb{Z}^n$ is the start vector and $W \subseteq \mathbb{Z}^n$ is finite set of addition vectors. A $\mathbb{Z}$-VAS evolves from $\mathbf{w}_0$ by sequentially applying the addition vectors from W. The set of vectors generated by a $\mathbb{Z}$-VAS is defined to be the set of reachable vectors.

An *integer vector addition system with states* ($\mathbb{Z}$-VASS) is a $\mathbb{Z}$-VAS equipped with a state control and is defined as a tuple $(\mathbf{w}_0, Q, q_0, q_h, p, \delta)$, where $\mathbf{w}_0 \in \mathbb{Z}^n$ is the start vector, Q is a finite set of state labels, $q_0 \in Q$ is the starting state, $q_h \in Q$ is the halting state, $p : Q \setminus \{q_h\} \to \mathbb{Z}^n$ is a function assigning a vector to every state from $Q \setminus \{q_h\}$, and $\delta : Q \to 2^Q$ is a state transition function assigning to each state the set of possible successor states.

A $\mathbb{Z}$-VASS starts in $\mathbf{w}_0$ and in state q_0, applies the addition vector $p(q_0)$, and non-deterministically moves into one of the states from $\delta(q_0)$. This process is iteratively repeated, until the halting state q_h is reached. The vector language generated by a $\mathbb{Z}$-VASS is defined as the set of all vectors which are reachable in the halting state q_h.

We will use the notations $\mathbb{Z}$-*VAS* and $\mathbb{Z}$-*VASS* to refer to the sets of integer vectors generated by $\mathbb{Z}$-VAS or $\mathbb{Z}$-VASS.

3 Register Machines

Definition 2. *A* register machine over the set A *is the tuple* $M_A = (n, A, Q, q_0, q_h, P)$*, where* $n \in \mathbb{N}$*,* A *is a register alphabet,* Q *is a finite set of state labels,* q_0 *is the initial state,* $q_h \in Q$ *is the halting state, and* P *is a mapping associating an instruction to every state of* M_A*. An* instruction *is a function* $p : A^n \to A^n \times 2^Q$ *associating to every* n*-tuple of values from* A *another* n*-tuple of such values and a set of states from* Q*. A configuration* $C \in Q \times A^n$ *of* M_A *is a tuple combining a state and* n *values from* A*.*

M_A can be seen as storing values of type A in its n registers. A configuration of M_A therefore defines its current state and the values of its n registers. When in state $q \in Q$, M_A can execute the instruction $P(q)$, which will compute (1) new values for *all* registers of M_A and (2) a set of possible new states; M_A can non-deterministically make a transition into one of these states.

Definition 3. *A* k*-step (finite)* computation *of the register machine* $M_A = (n, A, Q, q_0, q_h, P)$ *is a finite sequence of configurations* $(C_i)_{0 \leq i \leq k}$ *such that,*

1. $C_0 = (q_0, \mathbf{a}_0)$*, where some of the components of* $\mathbf{a}_0$ *(registers) may contain input values;*
2. $C_k = (q_h, \mathbf{a}_k)$*, where some of the components of* $\mathbf{a}_k$ *(registers) may contain output values;*
3. *for every* $0 \leq i < k$*,* $C_i = (q_i, \mathbf{a}_i)$*,* $C_j = (q_j, \mathbf{a}_j)$*,* $P(q_i)(a_i) = (\mathbf{a}_j, H)$*, and* $q_j \in H$*.*

M_A therefore makes a transition from a configuration to another one by sequentially applying its instructions. Whenever M_A is in state q_i, it retrieves the corresponding instruction $P(q_i)$ and applies it to the tuple describing the values of the registers. The result, $P(q_i)(a_i)$, gives the new values for the registers and a set of states H from which M_A picks q_j and moves into it. The last configuration C_h is habitually referred to as the *halting configuration.*

Often, in order to be able to express the instructions in a sensible way, some kind of structure over the set A is considered; one example of such a structure may be a finitely generated Abelian group. Classical definitions of register machines rely on (sub)sets of integers and on the associated structure of a linearly ordered finitely generated Abelian group.

In what follows, we describe the existing models of register machines using the abstract language we have just introduced, and we show that blind register machines actually represent the *least* restricted variant.

Definition 4. *A* blind register machine *is a register machine* B *over the finitely generated Abelian group* $(\mathbb{Z}, +)$*. The instructions of blind register machines can be of the following two types:*

1. $(ADD(i), q, s)(a_1, \ldots, a_i, \ldots, a_n) = \big((a_1, \ldots, a_i + 1, \ldots, a_n), \{q, s\}\big)$*, and*
2. $(SUB^*(i), q)(a_1, \ldots, a_i, \ldots, a_n) = \big((a_1, \ldots, a_i - 1, \ldots, a_n), \{q\}\big)$.

A computation of a blind register machine is defined as a computation of the corresponding register machine over $(\mathbb{Z}, +)$.

Definition 5. *A blind register machine* accepts *an input vector by resetting all registers to zero in the halting configuration. A blind register machine* generates *(or* computes *from an input) a vector of numbers by resetting all registers not containing the output to zero.*

We will use the notation $Ps_{\mathbb{Z}}BRM$ (resp., $Ps_{\mathbb{N}}BRM$) to refer to the class of sets of vectors of integer (resp., natural) numbers *accepted* by blind register machines.

All other well known types of register machines can be defined as subtypes of blind register machines.

Definition 6. *A* partially blind register machine *is a blind register machine whose registers are only allowed to contain non-negative numbers: for any computation $(C_i)_{1\leq i\leq k}$ of a partially-blind register machine and for any $C_i = (q_i, \mathbf{a}_i)$, $1 \leq i \leq k$, every component of $\mathbf{a}_i$ is non-negative.*

Thus, if the partially blind register machine B' decides at some point to decrement a register whose value is already zero, it will produce an illegal configuration which will render the whole computation invalid. This means that B' still cannot check its registers for zero, but it knows that all of them are non-negative at any given time. The computations of partially blind machines therefore satisfy a condition which renders them strictly stronger than blind register machines [6]: the registers may never go below zero.

Definition 7. *A partially blind register machine* accepts *an input vector by resetting all registers to zero in the halting configuration. A partially blind register machine* generates *(or* computes *from an input) a vector of numbers by resetting all registers not containing the output to zero in the halting configuration.*

We will use the notation $PsPBRM$ to refer to the class of sets of vectors of natural numbers accepted by partially blind register machines.

We can now also define conventional register machines in our general framework.

Definition 8. *A (conventional)* register machine *is a register machine over $(\mathbb{Z}, +)$ with the following two types of instructions:*

1. $(ADD(i), q, s)(a_1, \ldots, a_i, \ldots, a_n) = \big((a_1, \ldots, a_i + 1, \ldots, a_n), \{q, s\}\big)$, *and*
2. $(SUB(i), q, z)(a_1, \ldots, a_i, \ldots, a_n) =$
$$\begin{cases} \big((a_1, \ldots, a_i - 1, \ldots, a_n), \{q\}\big), & \text{if } a_i > 0, \\ \big((a_1, \ldots, a_i, \ldots, a_n), \{z\}\big), & \text{if } a_i = 0. \end{cases}$$

Computations of conventional register machines are defined as computations of the corresponding register machines over $(\mathbb{Z}, +)$ with the restriction that, in the initial configuration, all registers must contain non-negative values.

It follows from the form of instructions allowed in conventional register machines that their registers contain non-negative values at any time. Therefore, one can see such register machines as an even more powerful form of partially blind register machines (and thus a particular case of blind register machines), in which the machine is allowed to check whether any given register is zero.

We would like to remark that by considering other types of instructions or restrictions on the class of valid computations, one can characterise many other variants of register machines. For example, reversal-bounded counter automata are register machines in which one can only switch from incrementing to decrementing a register (and conversely) a bounded number of times, for example, see [9].

4 Integer Vector Addition P Systems

In [10], Gheorghe Păun suggested to explore multisets with negative multiplicities. Several possible answers were suggested. In [3], the authors defined generalised multisets as having multiplicities from totally ordered Abelian groups. In [1], a different approach is taken. The alphabet of objects is partitioned into two categories: the regular objects, which may have any integer multiplicity, and the so-called "catalysts", which are only allowed to appear in a bounded number of copies. Like in purely catalytic P systems, the "catalysts" in this model are used to guide the applicability of rules.

In this work, we generalise this model to the concept of integer vector addition P systems.

Definition 9. *An* integer vector addition P system *($\mathbb{Z}$-VAPS) is a construct*

$$\Pi = (O, T, \mu, w_1, \ldots, w_n, R, h_i, h_o),$$

where O is a finite alphabet of objects, $T \subseteq O$ is the set of terminal objects, μ is the membrane structure injectively labelled by the numbers from $\{1, \ldots, n\}$ and usually given by a sequence of correctly nested brackets, w_i are the $\mathbb{Z}$-multisets giving the initial contents of every membrane i, $1 \leq i \leq n$, R is a finite set of rules of the form $r : \{1, \ldots, n\} \to \mathbb{Z}^O$, and h_i and h_o are the labels of the input and the output membranes, respectively ($1 \leq h_i \leq n$, $1 \leq h_o \leq n$).

Thus, integer vector addition P systems manipulate vectors of integers, indexed by the objects from O ($\mathbb{Z}$-multisets). A rule $r \in R$ assigns such a vector to every membrane of Π; applying r means adding (by componentwise addition) the vector $r(i)$ to the vector representing the contents of the membrane i, for every $1 \leq i \leq n$. Therefore, one may see r as only having a right-hand side and as being unconditionally applicable. Such a form comes in naturally, since, as also pointed out in [3, 1], considering multiplicities over $\mathbb{Z}$ renders the usual rule applicability conditions irrelevant. We also remark that this way of defining the rules generalises naturally to a tissue-like membrane structure, i.e., a membrane structure which is not required to be a tree, but can be an arbitrary graph (for instance, see [5]).

We will use the tuple notation to describe rules of vector addition P systems – a rule r will be given by the set $\big\{(i, r(i)) \mid 1 \leq i \leq n\big\}$. We will often omit those tuples $(i, r(i))$ in which $r(i)$ is empty.

In [1], the authors use a special symbol δ to command the dissolution of the membrane in which it is produced. To allow for the same possibility in vector addition P systems, we will define the rules as functions of the form $\{1, \ldots, n\} \to \mathbb{Z}^{O \cup \{\delta\}}$, i.e., as functions assigning $\mathbb{Z}$-multisets over $O \cup \{\delta\}$ to each membrane. If $r(i)(\delta) = 1$ for the elementary membrane i, the application of r will dissolve this membrane after having added the multiplicities of symbols different from δ to its contents. We may even allow $r(i)(\delta) = k > 1$, in which case k successive membranes in the hierarchy will be dissolved. There are tow possible semantics for what happens the contents of the intermediary membranes: either only the contents of the innermost dissolved membrane will be moved into the corresponding parent membrane and the contents of the intermediary membranes will be lost, or else the contents of the innermost dissolved membrane as well as the contents of the intermediary membranes will be moved into the corresponding parent membrane.

Allowing dissolution makes it possible to introduce a rule *applicability condition*: r **is applicable if every membrane** i**, for which** $r(i)$ **is not empty, is still present in the system**.

The integer vector addition P system Π evolves by *sequentially* applying rules from R until a halting configuration is reached. Remark that, because of the use of $\mathbb{Z}$-multisets, the only way to use the classical halting condition is to dissolve all the membranes to which the rules of Π may contribute. This corresponds to the approach proposed in [1] which consists in dissolving all the working membranes until the result reaches a membrane without any rules. Thus, the classical halting condition becomes somewhat degenerate; it is therefore only natural to discuss other halting conditions, for example:

- *unconditional halting* – the system may halt at any moment, independently of rule applicability or of the contents of the membranes;
- *halting by zero* – the system halts when it reaches a configuration in which all multisets representing the contents of the still existing membranes are empty.

Unconditional halting can be seen as corresponding to the way in which the language generated by a grammar is defined [4]: essentially, the contents of the output membrane of Π in any configuration Π can reach, projected on the terminal alphabet T, is part of the vector language generated by Π. On the other hand, halting by zero corresponds to the way in which blind register machines recognise input vectors. From these two variants, in this paper we will only consider unconditional halting.

We will use the notions *uncond* and *inappl* to refer to unconditional halting and halting by inapplicability of rules. Similarly, we will use the symbols *acc* and *gen* to refer to the accepting and generating modes. We will use the notation $Ps_{\mathbb{Z}}VAPS(m, h)$, $m \in \{acc, gen\}$, $h \in \{uncond, inappl\}$, to refer to the class of sets of vectors of integers accepted or generated by integer vector addition

P systems working with the corresponding halting conditions. We will add the symbol δ to refer to the vector languages associated with $\mathbb{Z}$-VAPS with dissolution rules ($Ps_{\mathbb{Z}}VAPS(m,h,\delta)$) and the symbol δ^* to refer to the languages of $\mathbb{Z}$-VAPS which are allowed to dissolve multiple membranes at a time; in order to distinguish between the possible two semantics for what happens if more than one membrane is dissolved in one step from an inner membrane, we write δ^* if the the contents of the intermediary membranes will be lost ($Ps_{\mathbb{Z}}VAPS(m,h,\delta^*)$) and δ'^* if the contents of the intermediary membranes will be taken into the parent membrane, too ($Ps_{\mathbb{Z}}VAPS(m,h,\delta'^*)$). Finally, we will replace $\mathbb{Z}$ by $\mathbb{N}$ to refer to the languages of vectors of natural numbers (non-negative integers).

We immediately observe that $Ps_{\mathbb{Z}}VAPS(m,inappl) = \{\varnothing\}$, because if a $\mathbb{Z}$-VAPS has any rules at all, it can never halt by rule inapplicability.

5 On the Power of Blind Register Machines

In this section, we will focus on relating integer vector addition systems to blind register machines, as well as on expressing the power of both models in terms of semilinear vectors of numbers. We will show that blind register machines and $\mathbb{Z}$-VASS generate exactly $\mathbb{N}$-linear sets of $\mathbb{Z}$-vectors.

Also in [2], the computational power of blind and partially blind register machines is discussed, but a different definition of blindness is used: a blind register machine is defined as a partially blind register machine which may halt with any values in the registers. In the present paper we use a definition which is closer to Sheila Greibach's blind and partially register machines as defined in [6].

We will start by giving a proof of the quite intuitive result that blind register machines recognise exactly the same sets of integer vectors as generated by integer vector addition systems with states.

Theorem 1. *$Ps_{\mathbb{Z}}BRM = \mathbb{Z}$-VASS.*

Proof. Take a blind register machine $B = (n, \mathbb{Z}, Q, q_0, q_h, P)$; we will construct a $\mathbb{Z}$-VASS $\Gamma = (w_0, S, s_0, s_h, p, \delta)$ with $w_0 = (0, \ldots, 0) \in \mathbb{Z}^n$, $S = Q$, $s_0 = q_h$, $s_h = q_0$. The set $\delta(p)$ contains all the states of B from which p can be reached:

$$\begin{aligned}\delta(s) \;=\; \{q \in Q \mid P(q) = (SUB(i), s) \text{ or } P(q) = (ADD(i), s, s')\\ \text{or } P(q) = (ADD(i), s', s)\}.\end{aligned}$$

The vector $p(s)$ associated with a state $s \in S$ does the opposite effect of the instruction associated with the same state in B:

$$p(s) \;=\; \begin{cases} \mathbf{1}_i, & \text{if } P(s) = (SUB(i), q),\\ -\mathbf{1}_i, & \text{if } P(s) = (ADD(i), q, q'),\end{cases}$$

where $\mathbf{1}_i \in \mathbb{Z}^n$ is a vector whose only non-zero component is the i-th component.

It follows from the construction of Γ that, for every computation of B accepting an input vector $\mathbf{x}$, there exists a computation of Γ halting on the same vector, and conversely, which proves that $Ps_{\mathbb{Z}}BRM \subseteq \mathbb{Z}\text{-}VASS$.

To prove the converse inclusion, it suffices to take an arbitrary integer vector addition system, and to construct a blind register machine by reversing the arrows in the state control graph of $\mathbb{Z}$-VASS and by simulating the inverse effect of the addition vectors using multiple states. Thus, the machine will nondeterministically and sequentially subtract addition vectors from an input vector w and, at the end of the simulation, it will subtract w_0. If this resets all registers to zero, the machine has found an evolution of the simulated $\mathbb{Z}$-VASS which generates w. □

The same construction can be used to show that partially blind register machine are equivalent in power to conventional vector addition systems with states. Taking into consideration the result on the equivalence between (conventional) VAS and VAS with states from [8], we formulate the following characterisation of the power of partially blind register machines.

Theorem 2. *$PsPBRM = VASS = VAS$.*

We will now show that blind register machines do not recognise more than $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors.

Lemma 1. *$Ps_{\mathbb{Z}}BRM \subseteq \mathbb{Z}^*SLIN_{\mathbb{N}}$.*

Proof. Consider a blind n-register machine B. At every step, B can increment or decrement a register, *independently* of the contents of the registers. Consider the alphabet of actions of B: $\mathcal{A}_B = \{ADD(i), SUB(i) \mid 1 \leq i \leq n\}$; every computation of B can be represented as a string over this alphabet. Let $valid(A_B) \subseteq A_B^*$ be the strings over A_B^* which correspond to all computations of B. Pick such a string $w \in valid(A_B)$. Since the actions do not depend on the contents of the registers, any permutation of w which is in $valid(A_B)$ will have the same effect as w. In particular, B will halt with the same values in its registers. Therefore, the set of vectors B recognises can be described as follows:

$$N(B) = \left\{(a_1, \ldots, a_n) \mid w \in valid(A_B), a_i = |w|_{SUB(i)} - |w|_{ADD(i)}\right\},$$

where $|w|_x$ is the number of copies of the symbol $x \in A_B$ in the string w.

Because B cannot read the values of its registers, the set $valid(A_B)$ is the regular language given by the state control of B. Therefore, the Parikh image $Ps\big(valid(A_B)\big)$ is an $\mathbb{N}$-semilinear set. This means that $N(B)$ is an $\mathbb{N}$-semilinear set of $\mathbb{Z}$-vectors, and so is the set of vectors including only the values of the input registers of B. Consequently, $Ps_{\mathbb{Z}}BRM \subseteq \mathbb{Z}^*SLIN_{\mathbb{N}}$, which is the statement of the lemma. □

We will now show that blind register machines can recognise all $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors.

Lemma 2. *$Ps_{\mathbb{Z}}BRM \supseteq \mathbb{Z}^*SLIN_{\mathbb{N}}$.*

Proof. Consider an $\mathbb{N}$-semilinear set A of $\mathbb{Z}$-vectors. There exists a finite collection of sets of generators $A_i \subseteq \mathbb{Z}^n$ and offsets $\mathbf{a}_i \in \mathbb{Z}^n$ such that $A = \bigcup_i \langle A_i, \mathbf{a}_i \rangle_{\mathbb{Z}}$. Consider a blind register machine B which starts by non-deterministically choosing a set of generators A_i and the corresponding offset $\mathbf{a}_i$. B then repeats the following procedure until the set A_i is exhausted:

1. remove a generator $\mathbf{a}$ from A_i;
2. subtract $\mathbf{a}$ from the vector describing the registers of B a number of times chosen non-deterministically.

At the end, B subtracts the vector $\mathbf{a}_i$ from its registers. If B manages to reset all its registers using this procedure, then, by construction, the input vector belongs to $\langle A_i, \mathbf{a}_i \rangle_{\mathbb{Z}} \subseteq A$ (and the computation of the machine gives a way to construct this vector from A_i and $\mathbf{a}_i$). This implies the statement of the lemma. □

It follows from Lemmas 1 and 2 that blind register machines recognise exactly the $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors.

Theorem 3. $Ps_{\mathbb{Z}}BRM = \mathbb{Z}^* SLIN_{\mathbb{N}}$.

Consequently, if we only take the vectors of natural numbers recognised by blind register machines, we obtain the positive-restricted $\mathbb{N}$-semilinear sets of $\mathbb{Z}$-vectors.

Corollary 1. $Ps_{\mathbb{N}}BRM = \mathbb{Z}^*_+ SLIN_{\mathbb{N}}$.

6 On the Power of $\mathbb{Z}$-VA(P)S

In this section we will describe the power of integer vector addition (P) systems in terms of semilinear sets of vectors and blind register machines. We will start by pointing out that $\mathbb{Z}$-VAPS without dissolution and with unconditional halting generate exactly the sets reachable by $\mathbb{Z}$-VAS.

Lemma 3. $Ps_{\mathbb{Z}}VAPS(gen, uncond) = \mathbb{Z}\text{-}VAS$.

Proof. The effect of rules of $\mathbb{Z}$-VAPS without dissolution does not depend on the contents of the membranes. Consider the set of rules R of such a P system Π; we will construct a $\mathbb{Z}$-VAS Γ whose starting vector is the initial contents of the output membrane h_o of Π, and whose addition vectors are given by the projection $\{r(h_o) \mid r \in R\}$. Since Π can halt at any moment, its output is exactly the set of reachable vectors of Γ. Therefore $Ps_{\mathbb{Z}}VAPS(gen, uncond) \subseteq \mathbb{Z}\text{-}VAS$.

The converse inclusions follows from the fact that any $\mathbb{Z}$-VAS can be seen as a one-membrane $\mathbb{Z}$-VAPS working with unconditional halting. □

Because of the direct equivalence between $\mathbb{Z}$-VAS and $\mathbb{N}$-linear sets of $\mathbb{Z}$-vectors, we can write the previous result in the following way.

Theorem 4. $Ps_{\mathbb{Z}}VAPS(gen, uncond) = \mathbb{Z}^* LIN_{\mathbb{N}} = \mathbb{Z}\text{-}VAS$.

Allowing membrane dissolution together with halting by rule inapplicability allows for generating any set from $\mathbb{Z}^* SLIN_{\mathbb{N}}$. An important feature for obtaining this rather surprising result is the applicability condition for rules in an integer vector addition P system Π which only allows rules to be applied if only still existing membranes are to be affected.

Lemma 4. $Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta) \supseteq \mathbb{Z}^* SLIN_{\mathbb{N}}$.

Proof. Consider a family F of n $\mathbb{Z}$-VAS, each of which generates a $\mathbb{Z}$-linear set of vectors of dimension d. We now construct an integer vector addition P system Π with dissolution in the following way.

$$\Pi = (O, T, \mu, w_1, w_2 \ldots, w_{n+1}, R, h_o = 1),$$

where $O = \{a_i \mid 1 \leq i \leq n\}$ is a finite alphabet of objects a_i for representing the components of the d-dimensional vectors, $T = O$ is the set of terminal objects, $\mu = [_1 [_2]_2 \ldots [_{n+1}]_{n+1}]_1$ is the membrane structure with the skin membrane containing n inner membranes, $w_i = \lambda$, $1 \leq i \leq n+1$, i.e., every membrane is empty at the beginning, and the output membrane is the skin membrane. R is a finite set of rules simulating the n $\mathbb{Z}$-VAS $V_i = (\mathbf{u}_i, U_i)$, $1 \leq i \leq n$, where u_i is the start vector of V_i and U_i is a finite set of vectors of dimension d:

$$\begin{aligned} R &= R_1 \cup R_2, \\ R_1 &= \{\{(i+1, v_i) \mid 1 \leq i \leq n\} \mid v_i \in U_i\}, \\ R_2 &= \{\{(i+1, \mathbf{u}_i x_i) \mid 1 \leq i \leq n\} \mid x_i \in \{\lambda, \delta\}, 1 \leq i \leq n, \prod_{i=1}^{n} x_i = 1\}. \end{aligned}$$

The rules in R_1 simulate the i-th $\mathbb{Z}$-VAS V_i in membrane $i+1$, for all i, $1 \leq i \leq n$, in parallel. At the end of the simulation, one rule in R_2 is applied, adding the corresponding start vectors, but also releasing exactly the result representing a vector generated by one V_i into the skin membrane. After the application of a rule from R_2 the computation in Π halts, as all rules in R require all membranes still to be present. Thus, Π generates the semilinear language generated by the family F. □

To prove the inverse inclusion, we will rely on Lemma 1, and we will even allow multiple membrane dissolutions.

Lemma 5. $Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta^*) \subseteq \mathbb{Z}^* SLIN_{\mathbb{N}}$.

Proof (Sketch). Consider a $\mathbb{Z}$-VAPS Π with multiple dissolution. We then construct a blind register machine B which recognises the vector language generated by Π in the following way. B has a group of working register per membrane of Π which will represent the multiplicities of the symbols in this membrane. B starts with the vector $\mathbf{x}$ in its input registers, and then simulates the applications of rules of Π in its working registers. Whenever Π dissolves a membrane (or multiple membranes), B non-deterministically guesses the multiplicities of

each symbol in the dissolved membrane and copies the guessed values into the working registers representing the corresponding parent membrane. When all inner membranes of the output membrane have been dissolved (B can encode the information about the membrane structure in its state), B simultaneously decrements the working registers representing the contents of the output membrane and the input registers. If, earlier during the simulation, B has guessed the value of a register in a wrong way, or, at the end of the simulation, the values of the input registers and the working registers representing the output membrane did not match, some registers of B will not be zero and the vector $\mathbf{x}$ will be rejected. It follows from the construction that B accepts exactly the vector generated by Π, which implies the statement of the theorem. □

A similar argument holds if we replace δ^* by δ'^*, i.e.,

$$Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta'^*) \subseteq \mathbb{Z}^*SLIN_{\mathbb{N}}.$$

Summarizing the preceding two lemmas and taking into account the obvious inclusions

$$Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta) \subseteq Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta^*)$$

and

$$Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta) \subseteq Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta'^*)$$

we obtain the following result:

Theorem 5.

$$\begin{aligned}\mathbb{Z}^*SLIN_{\mathbb{N}} &= Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta)\\ &= Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta^*)\\ &= Ps_{\mathbb{Z}}VAPS(gen, inappl, \delta'^*).\end{aligned}$$

We now consider a specific subclass of integer vector addition P systems with (multiple dissolution) where the membrane structure is linear, and we denote the corresponding classes by

$$Ps_{\mathbb{Z}}VAP_{lin}S(gen, inappl, \delta) \text{ and } Ps_{\mathbb{Z}}VAP_{lin}S(gen, inappl, \delta^*).$$

When allowing multiple membrane dissolution in one rule, a linear membrane structure is sufficient to again obtain $\mathbb{Z}^*SLIN$.

Theorem 6. $Ps_{\mathbb{Z}}VAP_{lin}S(gen, inappl, \delta^*) = \mathbb{Z}^*SLIN_{\mathbb{N}}$.

Proof. Consider a family F of n $\mathbb{Z}$-VAS, each of which generates a $\mathbb{Z}$-linear set of vectors of dimension d, i.e., we have n $\mathbb{Z}$-VAS $V_i = (\mathbf{u}_i, U_i)$, $1 \leq i \leq n$, where u_i is the start vector of V_i and U_i is a finite set of vectors of dimension d. We

now construct an integer vector addition P system Π with dissolution in the following way.

$$\Pi = (O, T, \mu, w_1, w_2 \ldots, w_{n+1}, R, h_o = 1),$$

where $O = \{a_i \mid 1 \leq i \leq n\}$ is a finite alphabet of objects a_i for representing the components of the d-dimensional vectors, $T = O$ is the set of terminal objects, $\mu = [_1 [_2, \ldots [_{n+1}]_{n+1} \ldots]_2]_1$ is the linear membrane structure of depth n, $w_i = \lambda$, $1 \leq i \leq n+1$, i.e., every membrane is empty at the beginning, and the output membrane is the skin membrane. R is a finite set of rules simulating the n $\mathbb{Z}$-VAS $V_i = (\mathbf{u}_i, U_i)$, $1 \leq i \leq n$:

$$\begin{aligned} R &= R_1 \cup R_2, \\ R_1 &= \{\{(i+1, v_i) \mid 1 \leq i \leq n\} \mid v_i \in U_i\}, \\ R_2 &= \{\{(i+1, \mathbf{u}_i x_i) \mid 1 \leq i \leq n\} \mid x_i \in \{\lambda, \delta^i\}, 1 \leq i \leq n, \\ &\qquad card\{x_i \mid x_i \neq \lambda, 1 \leq i \leq n\} = 1\}. \end{aligned}$$

The rules in R_1 simulate the i-th $\mathbb{Z}$-VAS V_i in membrane $i+1$, for all i, $1 \leq i \leq n$, in parallel. At the end of the simulation, one rule in R_2 is applied, adding the corresponding start vectors, but also releasing exactly the result representing a vector generated by one V_i into the skin membrane by going through all the membranes in between down to the skin. After the application of a rule from R_2 the computation in Π halts, as all rules in R require all membranes still to be present. Thus, Π generates the semilinear language generated by the family F. □

We now consider the special variant of families of $\mathbb{Z}$-VAS which may only differ in their start vectors. We will call such families *uniform* and will denote the class of vector languages generated by such families by $\mathbb{Z}$-$VAS_{\cup}$.

It turns out that this family $\mathbb{Z}$-$VAS_{\cup}$ can be characterized by integer vector addition P systems with only two membranes; we denote the corresponding class by $Ps_{\mathbb{Z}}VAP_2S(gen, inappl, \delta)$.

Theorem 7. $Ps_{\mathbb{Z}}VAP_2S(gen, inappl, \delta) = \mathbb{Z}$-$VAS_{\cup}$.

Proof. First consider a finite family of $\mathbb{Z}$-VAS $F = \{(\mathbf{u}_i, U) \mid 1 \leq i \leq n\}$. We now define an integer vector addition P system Π generating the vectors reachable by the systems from F in the following way. Π has two nested membranes, i.e., the membrane structure $\mu = [_1 [_2]_2]_1$, and two groups of rules R_1 and R_2. As in the previous proofs, the first group of rules applies the vectors from U to the inner membrane. A rule of the second group finally adds one of the vectors $\mathbf{u}_i$ to the inner membrane and at the same moment dissolves it. By construction, the vectors appearing in the halting configurations of Π are exactly the vectors which can be reached by the $\mathbb{Z}$-VAS from F, which proves the first inclusion.

For the other inclusion, consider an integer vector addition P system Π with the membrane structure $\mu = [_1 [_2]_2]_1$. In case the inner membrane is the output

membrane, there must not be any applicable rule, thus, we can only generate a singleton, i.e., the initial contents of the inner membrane. Hence, we now assume that the skin membrane is the designated output membrane. In order to allow the system to halt, there must not be a rule only involving the skin membrane. If there is no rule dissolving the inner membrane, we either obtain the empty set in case that there are applicable rules or else we again obtain a singleton result, i.e., the initial contents of the skin membrane.

Finally, if at some moment we can apply a rule dissolving the inner membrane, then we nearly have the same situation as in the first part of the proof. The main difference is that we may also add a vector to the skin membrane while adding another vector to the inner membrane. Yet as both the contents of the inner membrane and the skin membrane finally will be merged, we can immediately merge the results of the application of a rule in Π into a rule only affecting the contents of the inner membrane, thus obtaining an equivalent integer vector addition P system Π' which is exactly of the form of the integer vector addition P system constructed in the first part of the proof, which yields a set from $\mathbb{Z}$-$VAS_{\cup}$. This observation completes the proof. □

We now are going to show the interesting result that the class $\mathbb{Z}$-$VAS_{\cup}$ is strictly in between the classes $\mathbb{Z}$-VAS and $\mathbb{Z}$-$VASS$.

Lemma 6. $\mathbb{Z}$-$VAS \subsetneq \mathbb{Z}$-$VAS_{\cup}$.

Proof. The inclusion is trivial. Now consider two $\mathbb{Z}$-VAS having the axioms $(0,0)$ and $(0,1)$, and sharing the only addition vector $(1,1)$. The language of vectors reachable by these two systems is $L = \{(a,a),(a,a+1) \mid a \in \mathbb{N}\}$. Suppose there exists a $\mathbb{Z}$-VAS Γ generating the same vector language L. In order to generate all pairs of natural numbers (a,a), it must start with the axiom $(0,0)$ and have an addition vector of the form $(1,1)$. Then, in order to generate the pairs $(a,a+1)$, Γ needs to have an addition vector of the form $(x,x+1)$. However, applying this addition vector twice yields the vector $(2x,2x+2) \notin L$, which contradicts the supposition and proves that the inclusion from the statement of the lemma is strict. □

The following lemma describes the relationship between $\mathbb{Z}$-$VAS_{\cup}$ and $\mathbb{Z}$-$VASS$.

Lemma 7. $\mathbb{Z}$-$VAS_{\cup} \subsetneq \mathbb{Z}$-$VASS$.

Proof. The work of a finite family F of $\mathbb{Z}$-VAS can be simulated by a $\mathbb{Z}$-VASS by non-deterministically choosing a state in which one of the start vectors of F will be added, and by subsequent direct simulation of the application of the shared addition vectors.

Now consider the $\mathbb{Z}$-VASS Γ with the starting vector $(0,0)$, which applies the addition vector $(0,0)$ in the starting state q_0 and then non-deterministically chooses between $q_{(1,0)}$ and $q_{(0,1)}$. In $q_{(1,0)}$, Γ may apply the addition vector $(1,0)$

indefinitely, before entering q_h. Similarly, in $q_{(0,1)}$, Γ may apply the addition vector $(0,1)$ indefinitely, before moving into q_h. Thus, the vector language generated by Γ is $L = \{(a,0),(0,a) \mid a \in \mathbb{N}\}$.

Suppose there exists a family of $\mathbb{Z}$-VAS which generate the same language. The shared addition vectors of this family therefore must include both $(1,0)$ and $(0,1)$. But then, this family must also generate vectors in which both components are non-zero and which therefore do not belong to L. This contradicts our supposition and proves that the inclusion in the statement of the lemma is strict. □

The previous lemma also gives an example of a $\mathbb{Z}$-semilinear set which cannot be generated by uniform family of $\mathbb{Z}$-VAS systems, which implies the following result.

Corollary 2. $\mathbb{Z}\text{-}VAS_{\cup} \subsetneq \mathbb{Z}^* SLIN_{\mathbb{N}}$.

We now from Theorem 7, Lemma 6, and from the preceding Corollary 2 infer the following result:

Theorem 8. $\mathbb{Z}^* LIN_{\mathbb{N}} \subsetneq \mathbb{Z}\text{-}VAS_{\cup} = Ps_{\mathbb{Z}}VAP_2S(gen, inappl, \delta) \subsetneq \mathbb{Z}^* SLIN_{\mathbb{N}}$.

7 Conclusion

In this paper we continued the investigation of P systems with multisets with integer multiplicities, proposed in [10] and already studied in [3] and [1]. We focused on the model originally described in [1] and generalised it to integer vector addition P systems, in which the applicability of rules does not in any way depend on the contents of the membranes. Interestingly enough, this P system variant exhibits very strong connection with blind register machines and integer vector addition systems – two models which have received little attention in the scientific literature up to now.

We have studied a number of working modes and halting conditions for integer vector addition P systems and have given exact characterisations of the power of the corresponding variants in terms of linear and semilinear sets over $\mathbb{Z}$ and over $\mathbb{N}$. We have also pointed out a number of relations between the classes of languages generated or accepted by this model.

Some non-trivial open questions are revealed by our research. One of them concerns the semantics of multiple dissolution. In P systems, dissolution typically concerns one membrane at a time; in the present paper we suggest considering the possibility of dissolving multiple containing membranes in one step. One of the semantics we propose discards the contents of the dissolved intermediary membranes, so only the multiset of the innermost dissolved membrane is transferred to the corresponding parent membrane. Other semantics of multiple dissolution may be possible and are certainly worth to be explored.

A very interesting open question concerns the types of semilinear sets. In this paper we only deal with semilinear sets with generators and initial offsets

from $\mathbb{N}^n$ and $\mathbb{Z}^n$, restricted to non-negative values or not. It is however possible to consider the generators, the offsets, and the *coefficients* to belong to $\mathbb{N}^n$ or $\mathbb{Z}^n$, alternatively. This yields eight possibly different kinds of semilinear sets, not including restrictions to non-negative values. Exploring the relations between these kinds of semilinear sets may be useful in further refining certain characterisations.

Finally, we point out that classical halting by inapplicability of rules is not necessarily well adapted for dealing with generalisations of multisets to integers. We have given examples of different halting conditions inspired by other models of computing, but our list is far from being exhaustive and is definitely worth to be extended.

References

1. O. Belingheri, A. E. Porreca, and C. Zandron. P systems with hybrid sets, 2016. Workshop on Membrane Computing, submitted.
2. R. Freund, O. Ibarra, Gh. Păun, and H.-C. Yen. Matrix languages, register machines, vector addition systems. *Third Brainstorming Week on Membrane Computing*, pages 155–167, 2005.
3. R. Freund, S. Ivanov, and S. Verlan. P systems with generalized multisets over totally ordered abelian groups. In *Int. Conf. on Membrane Computing*, volume 9504 of *Lecture Notes in Computer Science*, pages 117–136. Springer, 2015.
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, editors, *Computation, Cooperation, and Life*, volume 6610 of *Lecture Notes in Computer Science*, pages 35–53. Springer Berlin Heidelberg, 2011.
5. R. Freund and S. Verlan. A formal framework for static (tissue) P systems. In G. Eleftherakis, P. Kefalas, Gh. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*, volume 4860 of *Lecture Notes in Computer Science*, pages 271–284. Springer Berlin Heidelberg, 2007.
6. S. A. Greibach. Remarks on blind and partially blind one-way multicounter machines. *Theoretical Computer Science*, 7(3):311–324, 1978.
7. C. Haase and S. Halfon. Integer vector addition systems with states. In J. Ouaknine, I. Potapov, and J. Worrell, editors, *Reachability Problems: 8th International Workshop, RP 2014, Oxford, UK, September 22-24, 2014. Proceedings*, pages 112–124. Springer, 2014.
8. J. Hopcroft and J.-J. Pansiot. On the reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8(2):135–159, 1979.
9. O. H. Ibarra. Automata with reversal-bounded counters: A survey. In H. Jürgensen, J. Karhumäki, and A. Okhotin, editors, *Descriptional Complexity of Formal Systems: 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings*, pages 5–22. Springer, 2014.
10. Gh. Păun. Some quick research topics. `http://www.gcn.us.es/files/OpenProblems_bwmc15.pdf`.
11. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, 61:108–143, 1998.
12. Gh. Păun, G. Rozenberg, and A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA, 2010.

Maximal Variants of the Set Derivation Mode

Artiom Alhazov[1], Rudolf Freund[2], and Sergey Verlan[3]

[1] Institute of Mathematics and Computer Science
Academy of Sciences of Moldova
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md
[2] Faculty of Informatics, TU Wien
Favoritenstraße 9-11, 1040 Wien, Austria
E-mail: rudi@emcc.at
[3] LACL, Université Paris Est – Créteil Val de Marne
61, av. Général de Gaulle, 94010, Créteil, France
Email: verlan@u-pec.fr

Abstract. We consider P systems with derivation modes where rules are only applied in at most one copy in each derivation step; especially for the variant of the maximally parallel derivation mode we investigate the case where each rule may only be used at most once. Moreover, we also consider the derivation mode where from these sets of rules only those are taken which have the maximal number of rules. We check the computational completeness proofs of several variants of P systems and show that some of them even literally still hold true for these two new set derivation modes. Moreover, we establish two new results for P systems using target selection for the rules to be chosen together with these two new set derivation modes.

1 Introduction

Membrane systems with symbol objects are a theoretical framework of parallel distributed multiset processing. Usually, multisets of rules are applied in parallel to the objects in the underlying configuration; for example, in the maximally parallel derivation mode (abbreviated *max*), a non-extendable multiset of rules is applied to the current configuration. In this paper we now consider variants of these derivation modes, where each rule is only used in at most one copy, i.e., we consider sets of rules to be applied in parallel, for example, in the *set-maximally parallel derivation mode* (abbreviated *smax*) we apply non-extendable *sets* of rules, and in another derivation mode we apply sets of rules which contain a maximal number of applicable rules (abbreviated max_{rule}).

Taking sets of rules instead of multisets is a quite natural restriction and it arises from different motivations, e.g., firing a maximal set of transitions in Petri Nets [5, 9] or optimizing an implementation of FPGA simulators [14]. A natural question arises concerning the power of set-based modes in contrast to multiset-based ones. The first attempt to go into this direction was done in [11] where the derivation mode *smax* was called *flat maximally parallel derivation*

mode. Yet we here keep the notation of the *set-maximally parallel derivation mode* as we have already used it at the Conference on Membrane Computing 2015. In [11] it was shown that in some cases the computational completeness results established for the *max*-mode also hold for the flat maximally parallel derivation mode, i.e., for the *smax*-mode.

In this paper we continue this line of research and we show that for several variants of P systems the proofs for computational completeness for *max* can be taken over even literally for *smax* and eventually even for max_{rule}, but on the other hand there are also variants of P systems where the derivation modes *smax* and max_{rule} yield even stronger results than the *max*-mode. Full proofs of the results established in this paper and a series of additional results can be found in [4].

2 Variants of P Systems

In this section we recall the well-known definitions of several variants of P systems as well as some variants of derivation modes and also introduce the variants of set derivation modes considered in the following.

A (cell-like) P system is a construct

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, f_O, f_I) \text{ where}$$

- O is the alphabet of objects,
- $C \subset O$ is the set of catalysts,
- μ is the membrane structure (with m membranes, labeled by 1 to m),
- $w_1, \ldots, w_m$ are multisets of objects present in the m regions of μ at the beginning of a computation,
- $R_1, \ldots, R_m$ are finite sets of rules, associated with the regions of μ,
- f_O is the label of the membrane region from which the outputs are taken (in the generative case),
- f_I is the label of the membrane region where the inputs are put at the beginning of a computation (in the accepting case).

$f_O = 0/f_I = 0$ indicates that the output/input is taken from the environment. If f_O and f_I indicate the same label, we only write f for both labels.

If a rule $u \to v$ has at least two objects in u, then it is called *cooperative*, otherwise it is called *non-cooperative*. *Catalytic rules* are of the form $ca \to cv$, where $c \in C$ is a special object which never evolves and never passes through a membrane, it just assists object a to evolve to the multiset v.

In *catalytic P systems* we use non-cooperative as well as catalytic rules. In a *purely catalytic P system* we only allow catalytic rules.

In the *maximally parallel derivation mode* (abbreviated by *max*), in any computation step of Π we choose a multiset of rules from $\mathcal{R}$ (which is defined as the union of the sets $R_1, \ldots, R_m$) 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 regions $1, \ldots, m$.

2.1 Set Derivation Modes

The basic set derivation mode is defined as the derivation mode where in each derivation step at most one copy of each rule may be applied in parallel with the other rules; this variant of a basic derivation mode corresponds to the asynchronous mode with the restriction that only those multisets of rules are applicable which contain at most one copy of each rule, i.e., we consider *sets* of rules:

$$Appl(\Pi, C, set) = \{R \in Appl(\Pi, C, asyn) \mid |R|_r \leq 1 \text{ for each } r \in \mathcal{R}\}$$

In the *set-maximally parallel derivation mode* (this derivation mode is abbreviated by *smax* for short), in any computation step of Π we choose a non-extendable multiset R of rules from $Appl(\Pi, C, set)$; following the notations elaborated in [8], we define the mode *smax* as follows:

$$Appl(\Pi, C, smax) = \{R \in Appl(\Pi, C, set) \mid \text{there is no } R' \in Appl(\Pi, C, set) \text{ such that } R' \supset R\}$$

The *smax*-derivation mode corresponds to the min_1-mode with the discrete partitioning of rules (each rule forms its own partition), see [8].

As already introduced for multisets of rules in [6], we now consider the variant where the maximal number of rules is chosen. The derivation mode $max_{rule}smax$ is a special variant where only a maximal set of rules is allowed to be applied. But it can also seen as the variant of the basic set mode where we just take a set of applicable rules with the maximal number of rules in it, hence, we will also call it the max_{rule} derivation mode. Formally we have:

$$Appl(\Pi, C, max_{rule}) = \{R \in Appl(\Pi, C, set) \mid \text{there is no } R' \in Appl(\Pi, C, set) \text{ such that } |R'| > |R|\}$$

As usual, with all these variants of derivation modes as defined above, we consider halting computations. We may generate or accept or even compute functions or relations. The inputs/outputs may be multisets or strings, defined in the well-known way.

For any derivation mode γ, $\gamma \in \{sequ, asyn, max, smax, max_{rule}, \dots\}$, the families of number sets ($Y = N$) and Parikh sets ($Y = Ps$) $Y_{\gamma,\delta}(\Pi)$, generated ($\delta = gen$) or accepted ($\delta = acc$) by P systems with at most m membranes and rules of type X, are denoted by $Y_{\gamma,\delta}OP_m(X)$.

3 Catalytic and Purely Catalytic P Systems

We now investigate proofs elaborated for catalytic and purely catalytic P systems working in the *max*-mode for the derivation modes *smax* and max_{rule}.

3.1 Computational Completeness of Catalytic P Systems

We first check the construction for simulating a register machine $M = (d, B, l_0, l_h, R)$ by a catalytic P system Π, with $m \leq d$ being the number of decrementable registers, elaborated in [2] for the max-mode, and argue why it works for the derivation modes $smax$-mode and max_{rule}, too.

For all d registers, n_i copies of the symbol o_i are used to represent the value n_i in register i, $1 \leq i \leq d$. For each of the m decrementable registers, we take a catalyst c_i and two specific symbols d_i, e_i, $1 \leq i \leq m$, for simulating SUB-instructions on these registers. For every $l \in B$, we use p_l, and also its variants $\bar{p}_l, \hat{p}_l, \tilde{p}_l$ for $l \in B_{SUB}$, where B_{SUB} denotes the set of labels of SUB-instructions.

$$
\begin{aligned}
&\Pi = (O, C, \mu = [\ ]_1, w_1 = c_1 \dots c_m d_1 \dots d_m p_1 w_0, R_1, f = 1),\\
&O = C \cup D \cup E \cup \Sigma\\
&\quad \cup \{\#\} \cup \{p_l \mid l \in B\} \cup \{\bar{p}_l, \hat{p}_l, \tilde{p}_l \mid l \in B_{SUB}\},\\
&C = \{c_i \mid 1 \leq i \leq m\},\\
&D = \{d_i \mid 1 \leq i \leq m\},\\
&E = \{e_i \mid 1 \leq i \leq m\},\\
&\Sigma = \{o_i \mid 1 \leq i \leq d\},\\
&R_1 = \{p_j \to o_r p_k D_m, p_j \to o_r p_l D_m \mid j : (ADD(r), k, l) \in R\}\\
&\quad \cup \{p_j \to \hat{p}_j e_r D_{m,r}, p_j \to \bar{p}_j D_{m,r}, \hat{p}_j \to \tilde{p}_j D'_{m,r},\\
&\qquad \bar{p}_j \to p_k D_m, \tilde{p}_j \to p_k D_m \mid j : (SUB(r), k, l) \in R\}\\
&\quad \cup \{c_r o_r \to c_r d_r, c_r d_r \to c_r, c_{r\oplus_m 1} e_r \to c_{r\oplus_m 1} \mid 1 \leq r \leq m\},\\
&\quad \cup \{d_r \to \#, c_r e_r \to c_r \# \mid 1 \leq r \leq m\}\\
&\quad \cup \{\# \to \#\}.
\end{aligned}
$$

Here $r \oplus_m 1$ for $r < m$ simply is $r+1$, whereas for $r = m$ we define $m \oplus_m 1 = 1$; w_0 stands for additional input present at the beginning.

Usually, every catalyst c_i, $i \in \{1, \dots, m\}$, is kept busy with the symbol d_i using the rule $c_i d_i \to c_i$, as otherwise the symbols d_i would have to be trapped by the rule $d_i \to \#$, and the trap rule $\# \to \#$ then enforces an infinite non-halting computation.

In the derivation modes smax-mode and max_{rule} only one trap rule $\# \to \#$ will be carried out, but this is the only difference!

Only during the simulation of SUB-instructions on register r the corresponding catalyst c_r is left free for decrementing or for zero-checking in the second step of the simulation, and in the decrement case both c_r and its "coupled" catalyst $c_{r\oplus_m 1}$ are needed to be free for specific actions in the third step of the simulation.

For the simulation of instructions, we use:

$$
\begin{aligned}
D_m &= \textstyle\prod_{i \in [1..m]} d_i,\\
D_{m,r} &= \textstyle\prod_{i \in [1..m] \setminus \{r\}} d_i,\\
D'_{m,r} &= \textstyle\prod_{i \in [1..m] \setminus \{r, r\oplus_m 1\}} d_i.
\end{aligned}
$$

The HALT-instruction labeled l_h is simply simulated by not introducing the corresponding state symbol p_{l_h}, i.e., replacing it by λ, in all rules defined in R_1.

Each ADD-instruction $j : (ADD(r), k, l)$, for $r \in \{1, \ldots, d\}$, can easily be simulated by the rules $p_j \to o_r p_k D_m$ and $p_j \to o_r p_l D_m$; in parallel, the rules $c_i d_i \to c_i$, $1 \leq i \leq m$, have to be carried out, as otherwise the symbols d_i would have to be trapped by the rules $d_i \to \#$.

Each SUB-instruction $j : (SUB(r), k, l)$, is simulated as shown in the table listed below (the rules in brackets [and] are those to be carried out in case of a wrong choice):

Simulation of the SUB-instruction $j : (SUB(r), k, l)$ if	
register r is not empty	register r is empty
$p_j \to \hat{p}_j e_r D_{m,r}$	$p_j \to \bar{p}_j D_{m,r}$
$c_r o_r \to c_r d_r$ $[c_r e_r \to c_r \#]$	c_r should stay idle
$\hat{p}_j \to \tilde{p}_j D'_{m,r}$	$\bar{p}_j \to p_k D_m$
$c_r d_r \to c_r$ $[d_r \to \#]$	$[d_r \to \#]$
$\tilde{p}_j \to p_k D_m$	
$c_{r \oplus_m 1} e_r \to c_{r \oplus_m 1}$	

In the first step of the simulation of each instruction (ADD-instruction, SUB-instruction, and even HALT-instruction) due to the introduction of D_m in the previous step (we also start with that in the initial configuration) every catalyst c_r is kept busy by the corresponding symbol d_r, $1 \leq r \leq m$.

Based on the construction elaborated in [2] and recalled above in sum we have obtained the following result:

Theorem 1. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a catalytic P system*

$$\Pi = (O, C, \mu = [\ \ ]_1, w_1, R_1, f = 1)$$

working in one of the derivation modes max, smax or max_{rule} and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 5 \times m + 1,$$

where $ADD^1(R)$ denotes the number of deterministic ADD-instructions in R, $ADD^2(R)$ denotes the number of non-deterministic ADD-instructions in R, and $SUB(R)$ denotes the number of SUB-instructions in R.

3.2 Computational Completeness of Purely Catalytic P Systems

For the purely catalytic case, one additional catalyst c_{m+1} is needed to be used with all the non-cooperative rules. Unfortunately, in this case a slightly more complicated simulation of SUB-instructions is needed, a result established in [13], where for catalytic P systems

$$|R_1| \leq 2 \times ADD^1(R) + 3 \times ADD^2(R) + 6 \times SUB(R) + 5 \times m + 1,$$

and for purely for catalytic P systems

$$|R_1| \leq 2 \times ADD^1(R) + 3 \times ADD^2(R) + 6 \times SUB(R) + 6 \times m + 1,$$

is shown. Yet also this proof literally works for the derivation modes $smax$ and max_{rule} as well, with the only exception that the trap rule $\# \to \#$ is carried out at most once.

3.3 Computational Completeness of (Purely) Catalytic P Systems with Additional Control Mechanisms

In this subsection we mention results for (purely) catalytic P systems with additional control mechanisms, in that way reaching computational completeness with only one (two) catalyst(s).

P Systems with Label Selection For all the variants of P systems of type X, we may consider to label all the rules in the sets $R_1, \ldots, R_m$ in a one-to-one manner by labels from a set H and to take a set W containing subsets of H. In any transition step of a *P system with label selection* Π 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. The families of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and $\gamma \in \{sequ, asyn, max, smax, max_{rule}, \ldots\}$, computed by P systems with label selection with at most m membranes and rules of type X is denoted by $Y_{\gamma,\delta}OP_m(X, ls)$.

Theorem 2. *$Y_{\gamma,\delta}OP_1(cat_1, ls) = Y_{\gamma,\delta}OP_1(pcat_2, ls) = YRE$ for any $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and $\gamma \in \{max, smax, max_{rule}\}$.*

The proof given in [7] for the maximally parallel mode max can be taken over for the derivation modes $smax$ and max_{rule} word by word; the only difference is that in non-successful computations where more than one trap symbol # has been generated, the trap rule $\# \to \#$ is only applied once.

Controlled P Systems and Time-Varying P Systems Another method to control the application of the labeled rules is to use control languages (see [10] and [3]). In a *controlled P system* Π, in addition we use a set H of labels for the rules in Π, and a string language L over 2^H (each subset of H represents an element of the alphabet for L) from a family FL. Every successful computation in Π has to follow a control word $U_1 \ldots U_n \in L$: in transition step i, only rules with labels in U_i are allowed to be applied (in the underlying derivation mode, for example, max or $smax$), and after the n-th transition, the computation halts; we may relax this end condition, i.e., we may stop after the i-th transition for any $i \leq n$, and then we speak of *weakly controlled P systems*. If $L = (U_1 \ldots U_p)^*$, Π 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 $Y_{\gamma,\delta}(\Pi), Y \in \{N, Ps\}$, 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 $Y_{\gamma,\delta}OP_m(X, C(FL))$ $(Y_{\gamma,\delta}OP_m(X, wC(FL)))$ and $Y_{\gamma,\delta}OP_m(X, TV_p)$ $(Y_{\gamma,\delta}OP_m(X, wTV_p))$, respectively, for $\delta \in \{gen, acc\}$ and $\gamma \in \{sequ, asyn, max, smax, max_{rule}, \dots\}$.

Theorem 3. *$Y_{\gamma,\delta}OP_1(cat_1, \alpha TV_6) = Y_{\gamma,\delta}OP_1(pcat_2, \alpha TV_6) = YRE$, for any $\alpha \in \{\lambda, w\}$, $Y \in \{N, Ps\}$, $\delta \in \{gen, acc\}$, and $\gamma \in \{max, smax, max_{rule}\}$.*

The proof given in [7] for the maximally parallel mode max again can be taken over for the derivation modes $smax$ and max_{rule} word by word, e.g., see [4].

4 P Systems with Toxic Objects

In many variants of (catalytic) P systems, for proving computational completeness it is common to introduce a trap symbol # for the case that the derivation goes the wrong way as well as the rule $\# \to \#$ (or $c\# \to c\#$ with a catalyst c) guaranteeing that the derivation will never halt. Yet most of these rules can be avoided if we specify a specific subset of *toxic* objects O_{tox}.

The P system with toxic objects is only allowed to continue a computation from a configuration C by using an applicable multiset of rules covering all copies of objects from O_{tox} occurring in C; moreover, if there exists no multiset of applicable rules covering all toxic objects, the whole computation having yielded the configuration C is abandoned, i.e., no results can be obtained from this computation.

For any variant of P systems, we add the set of *toxic* objects O_{tox} and in the specification of the families of sets of (vectors of) numbers generated by P systems with toxic objects using rules of type X we add the subscript tox to O, thus obtaining the families $Y_{\gamma,gen}O_{tox}P_m(X)$, for any $m \geq 1$, $\gamma \in \{sequ, asyn, max, smax, max_{rule}\}$, and $Y \in \{N, Ps\}$.

The following theorem stated in [1] only for the max-mode obviously holds for the $smax$-mode, too.

Theorem 4. *For $\beta \in \{max, smax\}$,*

$$PsRE = Ps_{\beta,gen}O_{tox}P_1([p]cat_2).$$

In general, we can formulate the following "metatheorem":

Metatheorem: *Whenever a proof has been established for the derivation mode max and literally also holds true for the derivation mode smax, then omitting trap rules by using the concept of toxic objects works for both derivation modes in the same way.*

In the following sections, we now turn our attention to models of P systems where the derivation mode $smax$ yields different, in fact, stronger results than the derivation mode max.

5 Atomic Promoters and Inhibtors

As shown in [12], P systems with non-cooperative rules and atomic inhibitors are not computationally complete when the maximally parallel derivation mode is used. P systems with non-cooperative rules and atomic promoters can at least generate $PsET0L$. On the other hand, already in [11], the computational completeness of P systems with non-cooperative rules and atomic promoters has been shown. In the following we will establish a new proof for the simulation of a register machine where the overall number of promoters only depends on the number of decrementable registers of the register machine. Moreover, we also show a new pretty surprising result, establishing computational completeness of P systems with non-cooperative rules and atomic inhibitors, and the number of inhibitors again only depends on the number of decrementable registers of the simulated register machine. Finally, in both cases, if the register machine is deterministic, then the P system is deterministic, too.

5.1 Atomic Promoters

We now establish our new proof for the computational completeness of P systems with non-cooperative rules and atomic promoters when using any of the derivation modes $smax$ and max_{rule}; the overall number of promoters only is $5m$ where m is the number of decrementable registers of the simulated register machine.

Theorem 5. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with atomic inhibitors*

$$\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$$

working in the smax- or max_{rule}-derivation mode and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 7 \times m,$$

where $ADD^1(R)$ denotes the number of deterministic ADD-instructions in R, $ADD^2(R)$ denotes the number of non-deterministic ADD-instructions in R, and $SUB(R)$ denotes the number of SUB-instructions in R; moreover, the number of atomic inhibitors is $5m$. Finally, if the register machine is deterministic, then the P system is deterministic, too.

Proof. The numbers of objects o_r represent the contents of the registers r, $1 \leq r \leq d$; moreover, we denote $B_{SUB} = \{p \mid p : (SUB(r), q, s) \in R\}$.

$$\begin{aligned} O = &\{o_r \mid 1 \leq r \leq d\} \cup \{o'_r, c_r, c'_r, c''_r, c'''_r \mid 1 \leq r \leq m\} \\ &\cup (B \setminus \{l_h\}) \cup \{p', p'', p''' \mid p \in B_{SUB}\} \end{aligned}$$

The symbols from $\{o'_r, c_r, c'_r, c''_r, c'''_r \mid 1 \leq r \leq m\}$ are used as promoters.

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the two rules $p \to qo_r$ and $p \to so_r$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in four steps as follows:

1. $p \to p'c_r$;
2. $p' \to p''c'_r$; $o_r \to o'_r \,|_{c_r}$, $c_r \to \lambda$;
3. $p'' \to p'''c'''_r$, $c'_r \to c''_r \,|_{o'_r}$, $o'_r \to \lambda$;
4. $p''' \to q \,|_{c''_r}$, $p''' \to s \,|_{c'_r}$, $c'_r \to \lambda \,|_{c'''_r}$, $c''_r \to \lambda$, $c'''_r \to \lambda$.

As final rule we could use $l_h \to \lambda$, yet we can omit this rule and replace every appearance of l_h in all rules as described above by λ. □

5.2 Atomic Inhibitors

We now show that even P systems with non-cooperative rules and atomic promoters using the derivation mode *smax* or max_{rule} can simulate any register machine needing only $2m+1$ inhibitors where m is the number of decrementable registers of the simulated register machine.

Theorem 6. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with atomic inhibitors*

$$\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$$

a P system with atomic inhibitors $\Pi = (O, \mu = [\ \ ]_1, w_1 = l_0, R_1, f = 1)$ working in the smax- or max_{rule}-derivation mode and simulating the computations of M such that

$$|R_1| \leq ADD^1(R) + 2 \times ADD^2(R) + 5 \times SUB(R) + 3 \times m + 1,$$

where $ADD^1(R)$ denotes the number of deterministic ADD-instructions in R, $ADD^2(R)$ denotes the number of non-deterministic ADD-instructions in R, and $SUB(R)$ denotes the number of SUB-instructions in R; moreover, the number of atomic inhibitors is $2m + 1$. Finally, if the register machine is deterministic, then the P system is deterministic, too.

Proof. The numbers of objects o_r represent the contents of the registers r, $1 \leq r \leq d$. The symbols d_r prevent the register symbols o_r, $1 \leq r \leq m$, from evolving.

$$\begin{aligned} O = {} & \{o_r \mid 1 \leq r \leq d\} \cup \{o'_r \mid 1 \leq r \leq m\} \cup \{d_r \mid 0 \leq r \leq m\} \\ & \cup (B \setminus \{l_h\}) \cup \{p', p'', \tilde{p} \mid p \in B_{SUB}\} \end{aligned}$$

We denote $D = \prod_{i=1}^{m} d_i$ and $D_r = \prod_{i=1, i \neq r}^{m} d_i$.

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the two rules $p \to qo_rD$ and $p \to so_rD$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in four steps as follows:

1. $p \to p'D_r$;

2. $p' \to p''Dd_0$; in parallel, the following rules are used:
   $o_r \to o'_r \,|_{\neg d_r}$, $d_k \to \lambda$, $1 \leq k \leq m$;
3. $p'' \to \tilde{p}D \,|_{\neg o'_r}$; $o'_r \to \lambda$, $d_0 \to \lambda$;
   again, in parallel the rules $d_k \to \lambda$, $1 \leq k \leq m$, are used;
4. $p'' \to qD \,|_{\neg d_0}$, $\tilde{p} \to sD$.

As final rule we could use $l_h \to \lambda$, yet we can omit this rule and replace every appearance of l_h in all rules as described above by λ. □

6 P Systems with Target Selection

In P systems with target selection, all objects on the right-hand side of a rule must have the same target, and in each derivation step, for each region a (multi)set of rules – non-empty if possible – having the same target is chosen. We show that for P systems with target selection in the derivation mode *smax no* catalyst is needed any more, and with max_{rule}, we even obtain a deterministic simulation of deterministic register machines.

Theorem 7. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with non-cooperative rules working in the smax-derivation mode and simulating the computations of M.*

Proof. As usual, we take an arbitrary register machine M with d registers satisfying the following conditions: the output registers are $m+1, \cdots, d$, and they are never decremented; moreover, registers $1, \cdots, m$ are empty in any reachable halting configuration. Clearly, these conditions do not restrict the generality. We construct the following P system Π simulating M.

By B_{ADD} we denote the set of labels for ADD-instructions, and by B_{SUB} we denote the set of labels for SUB-instructions of M. The value of each register r is represented by the multiplicity of objects o_r in the skin.

The correct behavior of the object associated with the simulated instruction of M is the following.

In the decrement case, we have in_{r+2}, *out*, in_2, idle, *out*, in_2, *here*, *out*, *here* (9 steps in total), whereas in the zero-test case, we have the same as before, except that the fourth and the fifth steps are *out* and *here* instead of idle and *out*, respectively.

In case of an increment instruction, we get *here*, *here*, *here*, *here*, in_2, *here*, *out*, *here* (8 steps in total). We remark that the first four steps are carried out in the skin, while the last four steps repeat the cases of zero-test and decrement.

For every decrementable register r, there is a rule sending o_r into region $r+2$. However, this rule may only be applied safely in the first step of the simulation of the SUB-instruction, as otherwise some other object will also enter the same region as # (either one of e, e', e'', $\hat{e}$, $\hat{e}'$, which we will in the following refer to as the *guards*, or an object associated to the label of the simulated instruction, which we will in the following call a *program symbol*) forcing an unproductive computation, see the rules in brackets in the tables below.

$$
\begin{aligned}
\Pi &= (O, \mu, w_1, \cdots, w_{m+2}, R_1, \cdots, R_{m+2}) \text{ where}\\
O &= \{o_r \mid 1 \le r \le d\} \cup \{\bar{p}, p \mid p \in B\} \cup \{p', p'', \hat{p} \mid p \in B_{ADD}\}\\
&\quad \cup \{p', p_-, p'_-, p_0, p'_0, p''_0 \mid p \in B_{SUB}\} \cup \{\bar{e}, e, e', e'', \hat{e}, \hat{e}', d, \#\},\\
\mu &= [\,[\;\;]_2 \cdots [\;\;]_{m+2}\,]_1,\\
w_1 &= l_0,\ w_2 = e,\ w_{r+2} = \lambda,\ 1 \le r \le m,\\
R_1 &= \bigcup_{i=1}^{m+2} (R_{1,i,s} \cup R_{1,i,\#}),\\
R_i &= R_{i,1,s} \cup R_{i,1,\#} \cup R_{i,i,s} \cup R_{i,i,\#},\ 2 \le j \le m+2,\\
R_{1,1,s} &= \{e \to e', e' \to e'', e'' \to \hat{e}, \hat{e} \to \hat{e}', e' \to \lambda\}\\
&\quad \cup \{p'_0 \to p''_0 \mid p \in B_{SUB}\} \cup \{\bar{p} \to p \mid p \in B\}\\
&\quad \cup \{p \to \tilde{p}o_r \mid p : (ADD(r), q, s) \in P\}\\
&\quad \cup \{\tilde{p} \to p', p' \to p'', p'' \to \hat{p} \mid p \in B_{ADD}\},\\
R_{1,2,s} &= \{p' \to (p_-, in_2), p' \to (p_0, in_2), p'_- \to (p'_-, in_2), p''_0 \to (p''_0, in_2)\\
&\quad \mid p \in B_{SUB}\} \cup \{\hat{p} \to (\hat{p}, in_2) \mid p \in B_{ADD}\} \cup \{d \to (d, in_2)\}\\
R_{1,r+2,s} &= \{o_r \to (o_r, in_{r+2})\} \cup \{p \to (p, in_{r+2})\\
&\quad \mid p : (SUB(r), q, s) \in P\},\ 1 \le r \le m,\\
R_{1,1,\#} &= \{p' \to \#, p''_0 \to \#, p'_- \to \# \mid p \in B_{SUB}\} \cup \{\hat{p} \to \# \mid p \in B_{ADD}\}\\
&\quad \cup \{\# \to \#\},\\
R_{1,2,\#} &= \{p'_0 \to (\#, in_2), e'' \to (\#, in_2) \mid p \in B_{SUB}\}\\
&\quad \cup \{\bar{p} \to (\#, in_2) \mid p \in B\},\\
R_{1,r+2,\#} &= \{x \to (\#, in_{r+2}\} \mid x \in \{e, e', e'', \hat{e}, \hat{e}'\}\\
&\quad \cup \{p'_0, p'_- \mid P \in B_{SUB}\} \cup \{\bar{p} \mid P \in B\}\}\\
&\quad \cup \{p \to (\#, in_{r+2}) \mid p : (SUB(i), q, s) \in P,\ i \ne r\}\\
&\quad \cup \{p' \to (\#, in_{r+2}) \mid p \in B_{SUB}\},\ 1 \le r \le m,\\
R_{2,1,s} &= \{e \to (e, out)\} \cup \{\bar{p} \to (\bar{p}, out) \mid p \in B\}\\
&\quad \cup \{p_0 \to (p'_0, out), p_- \to (p'_-, out) \mid p \in B_{SUB}\},\\
R_{2,2,s} &= \{d \to \lambda, \bar{e} \to e\} \cup \{\mid p \in B\}\\
&\quad \cup \{p''_0 \to \bar{s}\bar{e}, p'_- \to \bar{q}\bar{e} \mid p : (SUB(r), q, s) \in P\}\\
&\quad \cup \{\hat{p} \to \bar{q}\bar{e}, \hat{p} \to \bar{s}\bar{e} \mid p : (ADD(r), q, s) \in P\},\\
R_{2,1,\#} &= \{d \to (\#, out), \# \to (\#, out)\},\\
R_{2,2,\#} &= \{p_0 \to \# \mid p \in B_{SUB}\} \cup \{\bar{p} \to \# \mid p \in B\},\\
R_{r+2,1,s} &= \{p \to (p', out) \mid p \in B_{SUB}\} \cup \{o_r \to (d, out\},\ 1 \le r \le m\\
R_{r+2,r+2,\#} &= \{\# \to (\#, out)\},\ R_{r+1,r+1,s} = R_{r+1,r+1,\#} = \emptyset.
\end{aligned}
$$

The "correct" target selection for the inner regions normally coincides with that of the program symbol (described above) and no rule is applied there if

the program symbol is not there, with the following exceptions. In the first step of simulating an instruction, object e exits membrane 2, as it is the only rule applicable there in this step. In the last step of simulating an instruction, object $\bar{e}$ is rewritten into e in membrane 2, as it is the only rule applicable there in this step. In the fourth step of the decrement case, the program symbol is idle while object d is erased. The "correct" target selection for the skin coincides with that of the program symbol, and is *here* if the program symbol is missing in the skin.

$(p : (SUB(r), q, s))$

	$r+2$	1	2
1	-	$o_r \to (o_r, in_{r+2})$	$e \to (e, out)$
	-	$p \to (p, in_{r+2})$	
		$[p \to (\#, in_{i+2}),\ i \neq r]$	
2	$p \to (p', out)$	$e \to e'$	-
	$o_r \to (d, out)$	$[e \to (\#, in_{i+2})]$	
3	-	$p' \to (p_-, in_2)$	-
		$p' \to (p_0, in_2)$	
		$d \to (d, in_2)$	
		$[p' \to \#]$	
		$[e' \to (\#, in_{i+2})]$	

	1,-	1,0	2,-	2,0
4	$e' \to e''$		$d \to \lambda$	$p_0 \to (p'_0, out)$
			$[p_- \to (p'_-, out)]$	$[d \to (\#, out)]$
				$[p_0 \to \#]$
5	$e'' \to \hat{e}$	$p'_0 \to p''_0$	$p_- \to (p'_-, out)$	-
	$[p'_- \to (p'_-, in_2)]$	$e'' \to \hat{e}$		
	$[p'_- \to \#]$	$[p'_0 \to (\#, in_t)]$		
	$[e'' \to (\#, in_t)]$	$[e'' \to (\#, in_t)]$		
	[for $t > 1$]	[for $t > 1$]		
6	$p'_- \to (p'_-, in_2)$	$p''_0 \to (p''_0, in_2)$	-	
	$[p'_- \to \#]$	$[p''_0 \to \#]$		
	$[p'_- \to (\#, in_{i+2})]$	$[p''_0 \to (\#, in_{i+2})]$		
7	$\hat{e} \to \hat{e}'$		$p'_- \to \bar{q}\bar{e}$	$p''_0 \to \bar{s}\bar{e}$
	$[\hat{e} \to (\#, in_{i+2})]$			
8	$\hat{e}' \to \lambda$		$\bar{q} \to (\bar{q}, out)$	$\bar{s} \to (\bar{s}, out)$
	$[\hat{e}' \to (\#, in_{i+2})]$		$[\bar{q} \to \#]$	$[\bar{s} \to \#]$
9	$\bar{q} \to q$	$\bar{s} \to s$	$\bar{e} \to e$	
	$[\bar{q} \to (\#, in_t)]$	$[\bar{s} \to (\#, in_t)]$		

Most trapping rules, given in brackets in the tables and listed in rule groups $R_{i,j,\#}$ above, are only needed to force the "correct" target selection. The exception are some rules in steps 4 and 5 of the simulation of SUB instructions, needed for verifying that the decrement and the zero test have been performed correctly (the guess is made at step 3 by the program symbol, and is reflected in

its subscript). Indeed, if the zero-test is chosen while d is present (signifying that the register was decremented), causing a target conflict: either p_0 or d will be anyway rewritten into #. However, if the decrement is chosen while d is absent (signifying that the register was zero), then p_- will appear in the skin in step 4 instead of step 5, causing a target conflict: either p'_- or e'' will be anyway rewritten into #.

$(p : (ADD(r), q, s))$

	1	2
1	$p \to \tilde{p}o_r$	$e \to (e, out)$
2	$\tilde{p} \to p'$ $e \to e'$	-
3	$p' \to p''$ $e' \to e''$	-
4	$p'' \to \hat{p}$ $e'' \to \hat{e}$	-
5	$\hat{p} \to (\hat{p}, in_2)$ $[\hat{p} \to \#]$	-
6	$\hat{e} \to \hat{e}'$	$\hat{p} \to \bar{x}\bar{e}$
7	$\hat{e}' \to \lambda$	$\bar{x} \to (\bar{x}, out)$ $[\bar{x} \to \#]$
8	$\bar{x} \to x$	$\bar{e} \to e$

Auxiliary rules

$r+2$	1	2
$[\# \to (\#, out)]$	$[\# \to \#]$	$[\# \to (\#, out)]$

Nearly half of the steps in the preceding constructions is needed for releasing the auxiliary symbol e in the first step of a simulation from membrane 2, yet in our construction, e and its derivatives are needed to control the correct target selection in the skin membrane, and especially to keep the register objects o_r from moving into membrane $r + 2$. Again we mention that any application of one of the rules given in brackets in the tables above leads to non-halting computations, not contributing to the result. □

In contrast to the derivation mode max_{rule} where we take the maximal sets of rules which are applicable, in the *smax*-derivation mode we may have several non-extendable sets of rules which are applicable to the current configuration although being of different sizes, which makes the proof much more difficult than in the case of max_{rule}.

We now show that taking the maximal sets of rules which are applicable, the simulation of SUB-instructions can even be carried out in a deterministic way.

Theorem 8. *For any register machine $M = (d, B, l_0, l_h, R)$, with $m \leq d$ being the number of decrementable registers, we can construct a P system with non-cooperative rules*

$$\Pi = (O, \mu = [\ [\ ]_2 \dots [\ ]_{2m+1}\ ]_1, w_1, \lambda, \dots, \lambda, R_1 \dots R_{2m+1}, f = 1)$$

working in the max_{rule}-derivation mode and simulating the computations of M such that

$$|R_1| \leq 1 \times ADD^1(R) + 2 \times ADD^2(R) + 4 \times SUB(R) + 10 \times m + 3,$$

where $ADD^1(R)$ denotes the number of deterministic ADD-instructions in R, $ADD^2(R)$ denotes the number of non-deterministic ADD-instructions in R, and $SUB(R)$ denotes the number of SUB-instructions in R.

Proof. The contents of the registers r, $1 \leq r \leq d$ is represented by the numbers of objects o_r, and for the decrementable registers we also use a copy of the symbol o'_r for each copy of the object o_r. This second copy o'_r is needed during the simulation of SUB-instructions to be able to distinguish between the decrement and the zero test case. For each r, the two objects o_r and o'_r can only be affected by the rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$ sending them into the membrane $r+1$ corresponding to membrane r (and at the same time erasing them; in fact, we could also leave them in the membrane unaffected forever as a garbage). These are already two rules, so any other combination of rules with different targets has to contain at least three rules.

One of the main ideas of the proof construction is that in the skin membrane the label p of an ADD-instruction is represented by the three objects p and e, e', and the label p of any SUB-instruction is represented by the eight objects $p, e, e', e'', d_r, d'_r, \tilde{d}_r, \tilde{d}_r{}'$. Hence, for each $p \in (B \setminus \{l_h\})$ we define $R(p) = pee'$ for $p \in B_{ADD}$ and $R(p) = pee'e''d_r d'_r \tilde{d}_r \tilde{d}_r{}'$ for $p \in B_{SUB}$ as well as $R(l_h) = \lambda$; as initial multiset w_1 in the skin membrane, we take $R(l_0)$.

$$O = \{o_r \mid 1 \leq r \leq d\} \cup \{o'_r \mid 1 \leq r \leq m\} \cup (B \setminus \{l_h\})$$
$$\cup \left\{d_r, d'_r, \tilde{d}_r, \tilde{d}_r{}' \mid 1 \leq r \leq m\right\} \cup \{e, e', e''\}$$

An ADD-instruction $p : (ADD(r), q, s)$ is simulated by the rules $p \to R(q)o_r$ and $p \to R(s)o_r$ as well as the rules $e \to \lambda$ and $e' \to \lambda$. This combination of three rules supersedes any combination of rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$, for some $1 \leq r \leq m$.

A SUB-instruction $p : (SUB(r), q, s)$ is simulated in two steps as follows:

1. In R_1, for the first step we take one of the following tuple of rules
   $p \to (p, in_{r+1})$, $d_r \to (\lambda, in_{r+1})$, $d'_r \to (\lambda, in_{r+1})$, $\tilde{d}_r \to (\lambda, in_{r+1})$,
   $o_r \to (\lambda, in_{r+1})$, $o'_r \to (\lambda, in_{r+1})$;
   $p \to (p, in_{m+r+1})$, $d_r \to (\lambda, in_{m+r+1})$, $d'_r \to (\lambda, in_{m+r+1})$,
   $\tilde{d}_r \to (\lambda, in_{m+r+1})$, $\tilde{d}_r{}' \to (\lambda, in_{m+r+1})$;
   the application of the rules $o_r \to (\lambda, in_{r+1})$, $o'_r \to (\lambda, in_{r+1})$ in contrast to the application of the rule $\tilde{d}_r{}' \to (\lambda, in_{m+r+1})$ determines whether the first

or the second tuple of rules has to be chosen. Here it becomes clear why we have to use the two register symbols o_r and o'_r, as we have to guarantee that the target $r+1$ cannot be chosen if none of these symbols is present, as in this case then only four rules could be chosen in contrast to the five rules for the zero test case. On the other hand, if some of these symbols o_r and o'_r are present, then six rules are applicable superseding the five rules which could be used for the zero test case.

2. In the second step, the following three or four rules, again superseding any combination of rules $o_r \to (\lambda, in_{r+1})$ and $o'_r \to (\lambda, in_{r+1})$ for some $1 \leq r \leq m$, are used in the skin membrane:
   $e \to \lambda$, $e' \to \lambda$, $e'' \to \lambda$, and in the decrement case also the rule $\tilde{d}_r{}' \to \lambda$.
   In the second step, we either find the the symbol p in membrane $r+1$, if a symbol o_r together with its copy o'_r has been present for decrementing or in membrane $m+r+1$, if no symbol o_r has been present (zero test case).
   In the decrement case, the following rule is used in R_{r+1}: $p \to (R(q), out)$.
   In the zero test case, the following rule is used in R_{m+r+1}: $p \to (R(s), out)$.

The simulation of the SUB-instructions works deterministically, hence, although the P system itself is not deterministic syntactically, it works in a deterministic way if the underlying register machine is deterministic. □

7 Conclusion and Future Work

It is not very surprising that many of the computational completeness proofs elaborated in the literature for the derivation mode *max* also work for the derivation modes *smax* and max_{rule}, as many constructions elaborated just "break down" maximal parallelism to near sequentiality in order to work for the simulation of register machines. On the other hand, we also have shown that due to this fact some variants of P systems become even stronger with the modes *smax* and max_{rule}. A comprehensive overview of variants of P systems we have already investigated can be found in [4], many more variants wait for future research.

References

1. Alhazov, A., Freund, R.: P systems with toxic objects. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20–22, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8961, pp. 99–125. Springer (2014)
2. Alhazov, A., Freund, R.: Small catalytic P systems. In: Dinneen, M.J. (ed.) Proceedings of the Workshop on Membrane Computing 2015 (WMC2015), (Satellite workshop of UCNC2015), August 2015, CDMTCS Research Report Series, vol. CDMTCS-487. Centre for Discrete Mathematics and Theoretical Computer, Science Department of Computer Science University of Auckland, Auckland, New Zealand (2015)

3. Alhazov, A., Freund, R., Heikenwälder, H., Oswald, M., Rogozhin, Yu., Verlan, S.: Sequential P systems with regular control. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, Gy. (eds.) Membrane Computing - 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7762, pp. 112–127. Springer (2013)
4. Alhazov, A., Freund, R., Verlan, S.: Computational completeness of P systems using maximal variants of the set derivation mode. In: Proceedings 14th Brainstorming Week on Membrane Computing, Sevilla, February 1–5, 2016 (2016, submitted)
5. Burkhard, H.: Ordered firing in Petri nets. Elektronische Informationsverarbeitung und Kybernetik 17(2/3), 71–86 (1981)
6. Ciobanu, G., Marcus, S., Păun, Gh.: New strategies of using the rules of a P system in a maximal way. power and complexity. Romanian Journal of Information Science and Technology 12(2), 21–37 (2009)
7. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013. EPTCS, vol. 128, pp. 47–61 (2013)
8. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) 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. Springer (2007)
9. Frisco, P., Govan, G.: P systems with active membranes operating under minimal parallelism. In: Gheorghe, M., Păun, Gh., Rozenberg, G., Salomaa, A., Verlan, S. (eds.) Membrane Computing - 12th International Conference, CMC 2011, Fontainebleau, France, August 23-26, 2011, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7184, pp. 165–181. Springer (2011)
10. Krithivasan, K., Păun, Gh., Ramanujan, A.: On controlled P systems. In: Valencia-Cabrera, L., García-Quismondo, M., Macas-Ramos, L., Martínez-del-Amor, M., Păun, Gh., Riscos-Núñez, A. (eds.) Proceedings 11th Brainstorming Week on Membrane Computing, Sevilla, 4–8 February 2013, pp. 137–151. Fenix Editora, Sevilla (2013)
11. Pan, L., Păun, Gh., Song, B.: Flat maximal parallelism in P systems with promoters. Theoretical Computer Science 623, 83–91 (2016)
12. Sburlan, D.: Further results on P systems with promoters/inhibitors. International Journal of Foundations of Computer Science 17(1), 205–221 (2006)
13. Sosík, P., Langer, M.: Small catalytic P systems simulating register machines. Theoretical Computer Science accepted (2015)
14. Verlan, S., Quiros, J.: Fast hardware implementations of P systems. In: Csuhaj-Varj, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, Gy. (eds.) Membrane Computing. 13th International Conference, CMC 2012, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7762, pp. 404–423. Springer (2013)

Computational Power of Protein Networks

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science, Iaşi, Romania
Blvd. Carol I no.8, 700505 Iaşi, Romania
bogdan.aman@gmail.com, gabriel@info.uaic.ro

Abstract. Cell biology provides useful ideas to computer scientists in order to construct models which can provide more efficient computations. In this paper we prove that an abstract model of protein-protein interaction derived from membrane computing has the same computational power as a Turing machine by using a rather small number of proteins having at most length two, where length is an abstract measure of complexity.

1 Introduction

Biological cells are complex systems composed of many components which are themselves components in a large system of organs. The processes inside a cell are integrated through a complex protein-protein network. It is widely known the importance of proteins as active agents and targets in cellular biology. Membrane proteins play critical roles in many biological and pathological processes, and constitute the majority of all drug targets. During the last years, several studies were devoted to protein-protein interaction networks and their role in new therapeutic methods for numerous diseases.

Membrane proteins are often arranged in large complexes and are important for many biological functions. For example, signals from the exterior of a cell are mediated to the inside of that cell by protein-protein interactions of the signalling molecules. Proteins might interact to form part of a protein complex, a protein may be transporting another protein, or a protein may interact briefly with another protein just to modify it. Such a (conformation) modification of proteins can itself change protein-protein interactions. Most of the reactions taking place in a cell are in fact controlled by proteins bound on cell membrane. These proteins can be of two types: peripheral proteins (placed on the internal or external side of a membrane), and integral proteins (have parts on both internal and external sides of a membrane). Freely floating molecules interact with the proteins bounded on membranes, and can be activated, manipulated, and pushed across the cell membranes. According to [1], proteins constitute about 50% of the mass of most animal cell membranes. The increasing complexity of protein-protein interactions has driven the selection of longer proteins by the addition of functional motifs. This increasing is an important evolutionary strategy for achieving complex systems. In order to cope with the increased complexity of

protein-protein interactions that arise within complex systems, protein lengths are correlated with systems size [20].

The living cells provide useful ideas to theoretical computer scientists in order to define models which can provide more efficient computations, models which can be used by biologists. It is not about simply applying computer science to biology, but by a systemic approach of the biological phenomena in terms of computational inspiration in which the processing of information is essential [14]. According to [13], "Life is computation. Every single cell reads information from a memory, rewrites it, receives data input (information about the state of its environment), processes the data and acts according to the results of all this computation. Globally, the zillions of cells populating the biosphere certainly perform more computation steps per unit of time than all man made computers put together".

Membrane systems [18] represent such a class of computing devices inspired by living cells which are complex hierarchical membrane structures with a flow of materials and information which underlies their functioning, involving parallel application of rules, communication between membranes and membrane dissolution. The structure of the cell is represented by a set of hierarchically embedded regions, each delimited by a surrounding boundary (called membrane), and all contained inside a so called "skin membrane". A membrane without any other membrane inside is said to be elementary, while a membrane with other membranes inside is said to be non-elementary. Multisets of objects are distributed inside these regions, and they can be modified or communicated between adjacent compartments. Objects represent the formal counterpart of molecular species (ions, proteins, etc.) floating inside cellular compartments, and multisets of objects are described by means of strings over a given alphabet. Evolution rules represent the formal counterpart of chemical reactions, and are given in the form of rewriting rules which operate on the objects, as well as on the structure by membrane influx, membrane efflux and elementary division.

A *computation* in membrane systems starts from an initial structure and the system evolves by applying the rules in a nondeterministic and maximally parallel manner. The maximally parallel way of using the rules means that in each step we apply a maximal multiset of rules such that no further rule can be added to the multiset being applicable. A rule is applicable when all the objects that appear on its left-hand side are available in the region where the rule is placed (the objects are not used by other rules applied in the same step). Due to the competition for available objects, some rules are applied nondeterministically. A halting configuration is reached when no rule can be applied anymore; the result is then given by the number of objects (in a specified region).

There are various models of computation (e.g., Turing machines) providing different interpretations for the notion of algorithm. Turing computable functions are the formalized analogue of the intuitive notion of algorithm. By considering an abstract measure of complexity that we call length, we prove that protein interaction networks using proteins of small lengths and acting according to various biological inspired operations can simulate all computable functions.

2 Protein-Protein Interaction Systems

The study of many complex biological systems has reached a stage where much is known about the molecular components and their functional capacity and interactions. A real challenge is how to integrate this wealth of information to explain complex behaviours at various system levels [12]; cell polarity represents one such example. Functional analysis of the proteins involved have uncovered several subfunctions in the establishment and maintenance of cell polarity, including GTPase signalling, exocytic deposition of membrane components and cell wall materials, and endocytic recycling.

After presenting some technical notions, we define the protein-protein interaction systems. Their rules are applicable whenever there is a agreement between membranes expressed by appropriate proteins, or protein complexes represented by bonds between proteins, and co-proteins, or co-protein complexes, on their surfaces. Let $\mathbb{N}$ be the set of non-negative integers, and consider a finite alphabet V of proteins. A multiset over V is a mapping $u : V \to \mathbb{N}$. The empty multiset is represented by ε. We use the string representation of multisets that is widely used in the membrane protein systems; when a multiset is represented by a string u, it means that every permutation of this string is allowed as a representation of the multiset. An example of such a representation is $u = aabaca$, where $u(a) = 4$, $u(b) = 1$, $u(c) = 1$. Using such a representation, the operations over multisets are defined as operations over strings. 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 $(u \backslash v)(a) = max\{0, u(a) - v(a)\}$ as the multiset difference.

Consider an alphabet $V = \{a_i, \overline{a_i} \mid 1 \leq i \leq n\}$ in which if a denotes a protein, then $\overline{a}$ denotes the corresponding co-protein. We denote by V^* the set of all strings over V. V^* is a monoid with ε as its unit element (as strings are used to denote multisets), and $V^+ = V^* \backslash \{\varepsilon\}$. For a string $u \in V^*$, $|u|$ denotes the number of occurrences of symbols from V in the string u. By $V^{\sim} = \{a_1 \sim \ldots \sim a_n \mid a_1, \ldots, a_n \in V, n \geq 1\}$ we denote protein complexes, where proteins in protein complexes of length greater than two are bonded. We use $\overline{a_1 \sim \ldots \sim a_n}$ as a shorthand notation for $\overline{a_1} \sim \ldots \sim \overline{a_n}$.

Protein-protein interaction systems represent a rule-based formalism involving parallelism and mobility introduced in order to model more specific biological systems [5, 6]. The biologically inspired rules taken from the immune system [16] and describing the mobility of membrane proteins inside the structure are: pinocytosis (engulfing zero external membranes), phagocytosis (engulfing just one external membrane), and exocytosis (expelling the content of a membrane outside the membrane where it is placed). Pinocytosis and phagocytosis represent different types of endocytosis. In these rules membranes agree on their movement by using complementary objects a and $\overline{a}$. Biologically speaking, the objects a and their corresponding co-objects $\overline{a}$ fit properly.

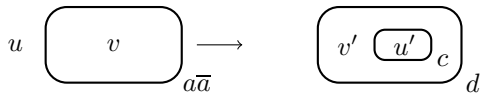
Definition 1. *A protein-protein interaction system with n membranes is a tuple* $\Pi = (V, \mu, u_1, \ldots, u_n, v_1, \ldots, v_n, R)$, *where*

1. *V is a finite (non-empty) alphabet of proteins;*

2. *μ is a membrane hierarchical structure (i.e., a rooted tree) with $n \geq 2$ membranes; the membranes are bijectively mapped to $\{1, \ldots, n\}$;*
3. *$u_1, \ldots, u_n$ are finite multisets of proteins (represented by strings over V) bounded to the n membranes at the beginning of the evolution;*
4. *$v_1, \ldots, v_n$ are finite multisets of proteins (represented by strings over V) placed inside the n membranes at the beginning of the evolution;*
5. *R is a finite set of rules of the following forms:*
   - $[a]_b \to [\ ]_{a\sim b}$, *for* $a \in V$, $b \in V^\sim$ (bondin)
   - $[\ ]_b a \to [\ ]_{b\sim a}$, *for* $a \in V$, $b \in V^\sim$ (bondout)
   - $u[v]_{a\,\overline{a}} \to [[u']_c v']_d$, *for* $a, \overline{a} \in V^\sim, u, v, u', v' \in V^*, c, d \in (V^\sim)^*$ (pino)
   - $[[u]_a v]_{\overline{a}} \to u'[v']_{c\,d}$, *for* $a, \overline{a} \in V^\sim, u, v, u', v' \in V^*, c, d \in (V^\sim)^*$ (exo)
   - $[u]_a[v]_{\overline{a}} \to [[[u']_c]_d v']_b$, *for* $a, \overline{a} \in V^\sim, u, v, u', v' \in V^*, c, d, b \in (V^\sim)^*$ (phago)

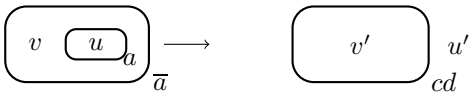
In rule (bondin), a bond is created between the protein labelled by a placed inside a membrane and the protein labelled by b placed on the membrane, while in rule (bondout), a bond is created between the protein labelled by a placed outside a membrane and the protein labelled by b placed on the membrane. The newly created protein complex has the length equal with the sum of lengths of a and b, and is able to interact with other protein complexes.

In rule (pino), a protein complex labelled by a together with a complementary protein complex labelled by $\overline{a}$ model the creation of an empty membrane within the membrane on which a and $\overline{a}$ proteins are attached. The connection between these proteins (a and $\overline{a}$) is activated by the presence of the multisets of proteins u and v. We should imagine that the original membrane protein receptor buckles towards the inside, and pinches off by breaking the connection between a and $\overline{a}$. The multiset of proteins u' inside the new created membrane is transferred from outside the initial membrane. The proteins a and $\overline{a}$, as well as the multisets u and v, can be modified during this step to the multisets c, d, u' and v', respectively. On the surface of the membrane appearing in the left hand side of the rule there are some proteins (others than $a\overline{a}$) which are ignored; these proteins are also not specified on the right hand side of the rule, being randomly distributed between the two resulting membranes. Graphically, this rule can be depicted as follows:

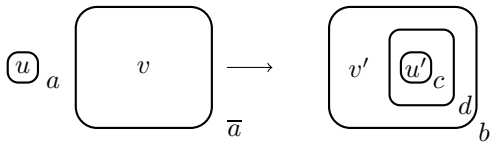


In rule (exo), a protein complex labelled by a together with a complementary protein complex labelled by $\overline{a}$ model the merging of a nested membrane with its surrounding membrane. The connection between these proteins (a and $\overline{a}$) is activated by the presence of the multisets of proteins u and v. We should imagine that the connection between a and $\overline{a}$ represent the point where the membranes connect each other. In this merging process (which is a smooth and continuous process), the membrane having the protein a on its surface gets expelled to the outside, and all proteins placed on the surface of the two membranes are united

into a multiset on the membrane which initially contained $\overline{a}$. The proteins a and $\overline{a}$, as well as the multisets u and v, can be modified during this step to the multisets c, d, u' and v', respectively. If the membrane protein receptor having on its surface the protein a is non-elementary, then its content is released near the newly merged membrane after applying the rule. On the surface of the membranes appearing in the left hand side of the rule there are some proteins (others than a and $\overline{a}$) which are ignored; these proteins are also not specified on the right hand side of the rule, being moved by default on the resulting membrane. Graphically, this process can be depicted as follows:



In rule (phago), a protein complex labelled by a together with its complementary protein complex labelled by $\overline{a}$ model a membrane (the one with $\overline{a}$ on its surface) "eating" an elementary membrane (the one with a on its surface). The connection between these proteins (a and $\overline{a}$) is activated by the presence of the multisets of proteins u and v. The membrane having $\overline{a}$ on its surface wraps around the membrane having a on its surface. An additional membrane is created around the eaten membrane; the proteins a and $\overline{a}$, as well as the multisets u and v, can be modified during this step to the multisets b, c, d, u' and v', respectively (the multiset c corresponds to a and remains on the eaten membrane, while the multisets b and d correspond to $\overline{a}$ and are placed on the new created membrane and the surrounding one). On the surface of the membranes appearing in the left hand side of the rule there are some proteins (others than a and $\overline{a}$) which are ignored, and these proteins are also not specified on the right hand side of the rule. The proteins appearing on the membrane having initially the protein a on surface remain unchanged, while the proteins appearing on the membrane having initially the protein $\overline{a}$ on surface are randomly distributed between the two resulting membranes (the ones with d and b). Graphically, this process can be depicted as follows:



Starting from an initial configuration of the newly defined protein-protein interaction system (given by the initial membrane structure and multisets of proteins), the evolution takes place by applying the rules activated by protein-protein interactions. A rule is applicable when all the involved proteins and membranes appearing in its left hand side are available. In each step a membrane can be used in at most one rule (pino), (exo) or (phago). In this way the evolution is parallel at the level of membranes. A halting configuration is reached when no rule is applicable.

3 Protein-Protein Interactions of the Immune System

The immune system is described well in [16], a book which is revised every few years to keep the pace with the new discoveries in this field. The cells of the immune system work together with different proteins to seek out and destroy anything dangerous which enters our body. It takes some time for the immune cell to be activated, but once this happens, there are very few hostile organisms having a chance. Immune cells are white blood cells produced in huge quantities in the bone marrow. There are a wide variety of immune cells, each of them with its own utility and selectivity of the antigen. Some seek out and engulf the invaders, while other destroy the infected or mutated body cells. Another type of cells, namely the B cells, following the binding to the antigen, have the ability to release special proteins called antibodies which mark intruders in order to be destroyed by macrophages. The immune system has also the ability to produce antibodies able to remember enemies which it fought in the past. In this way, once the immune system recognizes an invader, it attacks more quickly and strongly against it. Based on certain previous articles [3, 10], we illustrate here how the protein interaction networks work against infections. The description presented in [3] is based on the theoretical approach presented in [4] and [11].

Dendritic cells can engulf bacteria, viruses and other cells. Once a dendritic cells engulfs a bacterium, it dissolves this bacterium and places portions of bacterium proteins on its surface. These surface markers serve as an alarm to other immune cells, namely helper T cells, which then infer the form of the invader. This mechanism makes sensitive the T cells to recognize the antigens or other foreign agents which triggers a reaction from the immune system. Antigens are often found on the surface of bacterium and viruses.

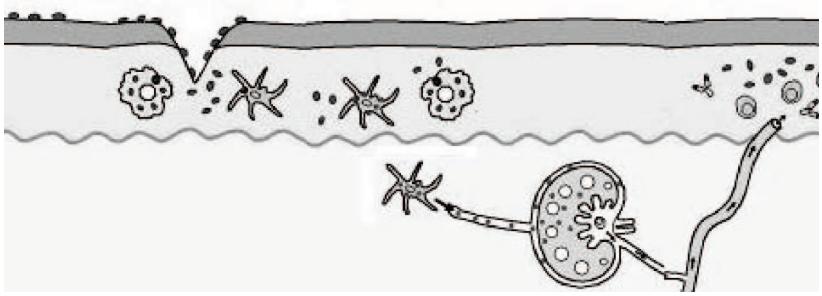


Fig. 1. Protection against infection [5]

In order to simulate the evolution presented in Figure 1, we need first to encode all the component of the immune system into our model. This can be realized by associating a membrane to each component, and some objects to the signals, states and parts of molecules. For the steps done by the dendritic cells presented in Figure 1, we use the following encodings:

– dendritic cell: $[eat]_{\overline{a_1}\,\overline{a_2}\,\overline{a_3}\,l}$
An immature dendritic cell is willing to eat any bacterium it encounters, so we translate it into a membrane which has inside an object *eat* used to engulf the bacterium and some proteins bounded to the membrane $\overline{a_1}$, $\overline{a_2}$, $\overline{a_3}$ (these are used to recognize viruses) and l (used to enter the lymph node). Once the dendritic cell matures, the object *eat* is consumed.
– bacterium cell: $[antigen]_{a_1}$
A bacterium cell contains antigen so we simply represent it as a membrane containing a single object *antigen* which contains the information of the bacterium and a protein a_1.
– lymph node: $[\ ]_{\overline{l}}$
The lymph node is the place where the mature dendritic cells migrate in order to start the immune response, so we translate it into a membrane that has bounded a protein $\overline{l}$.

Using the above encodings, we can describe the whole system as follows:

$$[eat]_{\overline{a_1}\,\overline{a_2}\,\overline{a_3}\,l}[\ ]_{\overline{l}}[antigen]_{a_1}\ ,$$

together with the following rules describing its evolution:

* $[eat]_{\overline{a_1}}[antigen]_{a_1} \to [eat[[antigen]_{a_1}]_{\overline{a_1}}]_{\overline{a_1}}$
Once an immature dendritic cell becomes sibling to a bacterium, it "eats" the bacterium by performing a phagocytosis rule. Until this moment the bacterium has controlled its own movement; in this step of the evolution the movement becomes controlled by the dendritic cell which eats the bacterium.
* $[[antigen]_{a_1}]_{\overline{a_1}} \to antigen[\ ]_{a_1}$
Once the bacterium has entered the dendritic cell, the content of the bacterium is released into the dendritic cell.
* $[[\ ]_{a_1}]_{\overline{a_1}} \to [\ ]_{a_1}$
The remaining parts of the membrane used to engulf the bacterium is joined with the membrane of the dendritic cell by a exocytosis rule.
* $[antigen]_l[\ ]_{\overline{l}} \to [[[antigen]_{l'}]_{\overline{l'\sim antigen}}]_{\overline{l}}$
Once the dendritic cell contains parts of antigen, it enters the lymph node in order to activate a special class of T cells, namely the helper T cells.
* $[antigen]_{l'} \to [\ ]_{l'\sim antigen}$
Once the dendritic cell enters the lymph node it displays on its surface the antigen of the bacterium in order to be able to interact with T cells.
* $[[eat]_{l'\sim antigen}]_{\overline{l'\sim antigen}} \to [\ ]_{l'\sim antigen}$
Once the dendritic cell enters the lymph node it matures and the capacity to engulf bacteria disappear, namely the *eat* object is consumed.

Using only these rules, we can simulate the way a bacterium is engulfed and its content is displayed by the eater cell. The proteins produced by helper T cells activate the B cells. Using the proteins produced by helper T cells, the B cell starts to divide and produce clones of itself. During this process, two new cell types are created: plasma cells which produce an antibody, and memory cells which are used to "remember" specific intruders.

This example motivates the introduction of the new class of membranes; more exactly, it motivates the new rules and the way they can be used in modelling some biological systems.

4 Computational Power of Interaction Networks

Formal languages and automata theory [19] are usually used to introduce abstract computing devices and investigating their computational power. Turing machines represent a classical model of computation. Actually, a Turing machine is a mathematical model that mechanically operates 0 and 1 symbols on an infinite tape according to a table of three simple operations: go left, go right, change symbol. Despite its simplicity, a Turing machine can simulate the logic of any computer algorithm. According to Church-Turing thesis, computable functions are exactly the functions that can be calculated using a Turing machine [21].

In this section we prove that protein-protein interaction networks can simulate all the computable functions by using a rather small number of components, small proteins (having at most length two, meaning actually two components) and operations inspired by membrane influx (pino, phago) and efflux (exo). The rules (pino) and (phago) are used to increase the number of membranes, while rule (exo) is used to decrease the number of membranes. Thus, we combine the rules (pino) and (phago) with (exo) just to balance the number of membranes. The result of a halting evolution consists of all the vectors describing the multiplicity of proteins inside and on all the membranes (a non-halting evolution provides no output).

In what follows, we study the computational power of the pair (pino, exo) of operations and prove its universality by using at most three membranes, while the lengths of the membrane proteins in both (pino) and (exo) operations are at most two. The number of three membranes represents the minimum number with respect to the movement rules provided in our approach. In order to prove this result, we construct a protein-protein interaction system with three nodes (membranes) able to simulate any function computed by a Turing machine (i.e., computable functions). This computational power supports the possibility of protein-protein interaction networks to describe algorithmically any normal and abnormal biological evolution.

The basic results used for proving the computational power are described in [15], where it was shown that any Turing machine can be simulated by a register machine with only two registers. The actions of such register machine can easily be simulated by matrix grammars with appearance checking. We define the notions used in what follows. A *matrix grammar with appearance checking* is a construct $G = (N, T, S, M, F)$ where N, T are disjoint alphabets of nonterminals and terminals, $S \in N$ is the axiom, M is a finite set of matrices of the form $mat = (A_1 \to x_1, \ldots, A_n \to x_n)$, $A_i \in N$, $x_i \in (N \cup T)^*$, $1 \leq i \leq n$, of context-free rules, and F is a set of occurrences of rules in M. For $w, z \in (N \cup T)^*$, we write $w \Rightarrow_{mat} z$ whenever (i) there is a matrix in M whose rules can be applied in order to obtain z from w, or whenever (ii) the j-th rule r_j of a matrix in M is not applicable to w_j ($w \Rightarrow_{mat} w_j$ in j steps) and $r_j \in F$, in which case r_j can be skipped obtaining $w_{j+1} = w_j$. The language generated by G is $L(G) = \{x \in T^* \mid S \Rightarrow^*_{mat} x\}$, where by $\Rightarrow^*_{mat}$ we denote the reflexive and transitive closure of the binary relation $\Rightarrow_{mat}$. A *matrix grammar in the strong binary normal form* is a construct $G = (N, T, S, M, F)$, where

$N = N_1 \cup N_2 \cup \{S, \#\}$ with these three sets mutually disjoint, two distinguished symbols $B^{(1)}, B^{(2)} \in N_2$, and the matrices in M are of one of the following forms:

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

(type-2) $(X \to Y, A \to x)$ with $X, Y \in N_1$, $A \in N_2$, $x \in (N_2 \cup T)^*$, $|x| \leq 2$;

(type-3) $(X \to Y, B^{(j)} \to \#)$ with $X, Y \in N_1$ and $B^{(j)} \to \# \in F$ for $j = 1, 2$;

(type-4) $(X \to \varepsilon, A \to x)$ with $X \in N_1, A \in N_2$, $x \in T^*$, $|x| \leq 2$.

If we do not use the empty string ε, then the rules of type (4) can be considered of the form $(X \to a, A \to x)$, with $X \in N_1, a \in T, A \in N_2, x \in T^*$, $|x| \leq 2$. Other notions and notations used here can be found in [19].

Theorem 1. *A protein-protein interaction system with three membranes, proteins of length two and using rules of types* (bondin), (bondout), (pino) *and* (exo) *has the same computational power as a Turing machine, i.e., are able to describe algorithmically any normal and abnormal biological evolution.*

Proof. We simulate a matrix grammar with appearance checking $G = (N, T, S, M, F)$ in the strong binary normal form. We construct a protein-protein interaction system Π with three membranes,

$$\Pi = (V, [\,[\,[\,]_2]_3]_1, \varepsilon, X, \overline{X}, \varepsilon, A, \overline{A}, R).$$

able to simulate a matrix grammar with appearance checking in the strong binary normal form. The symbols X, A correspond to the initial type-1 matrix $(S \to XA)$ with $X \in N_1, A \in N_2$. Let there be n_1 matrices of type-2 and type-4 labelled $1, \ldots, n_1$ and n_2 matrices of type-3 labelled $n_1 + 1, \ldots, n_1 + n_2$.

The finite alphabet V of proteins is defined as follows:

$$V = \{\beta, \overline{\beta}, x, x', A, \overline{A}, Y, \overline{Y}, a\} \cup \{B^{(j)}, \alpha_j, \overline{\alpha_j}, \alpha'_j, \overline{\alpha'_j} \mid 1 \leq j \leq 2\}$$
$$\cup \{X, \overline{X}, X_l, \overline{X_l}, X'_l, \overline{X'_l}, X''_l, \overline{X''_l}, X_l^{(j)}, \overline{X_l^{(j)}} \mid X \in N_1, 1 \leq l \leq n_1 + n_2, 1 \leq j \leq 2\}.$$

The set R of rules is constructed as follows:

(i) For each type-2 matrix $m_l : (X \to Y, A \to x)$ with $1 \leq l \leq n_1$, $X, Y \in N_1$, $A \in N_2$, $x \in (N_2 \cup T)^*$ and $|x| \leq 2$ we consider the rules:

1. $[\overline{A}]_X \to [\,]_{A \sim X}$
   $[\overline{A}]_{\overline{X}} \to [\,]_{\overline{A} \sim \overline{X}}$
2. $[\,[\,]_{A \sim X}]_{\overline{A} \sim \overline{X}} \to [\overline{A}]_{X_l \overline{X_l}} A$
3. $[\,]_{X_l} A \to [\,]_{X_l \sim A}$
   $[\overline{A}]_{\overline{X_l}} \to [\,]_{\overline{A} \sim \overline{X_l}}$
4. $[\,]_{\overline{A} \sim \overline{X}_l A \sim X_l} \to [\,[\,]_{X_l}]_{\overline{X_l}}$, if $x = \varepsilon$
   $[\,]_{\overline{A} \sim \overline{X}_l A \sim X_l} \to [\,[x]_{X_l}]_{\overline{X_l}}$, if $x \in T^*$
   $[\,]_{\overline{A} \sim \overline{X}_l A \sim X_l} \to [\,[x']_{X_l}]_{\overline{X_l}}$, otherwise
   (If $m_l : (X \to Y, A \to \alpha_1 \alpha_2), \alpha_1 \in N_2, \alpha_2 \in T \cup \{\varepsilon\}$ then $x' = \alpha'_1 \alpha_2$, and if $m_l : (X \to Y, A \to \alpha_1 \alpha_2), \alpha_1, \alpha_2 \in N_2$ then $x' = \alpha'_1 \alpha'_2$)
5. $[\,[\,]_{X_l}]_{\overline{X_l}} \to [\,]_{X'_l \overline{X'_l}}$
6. $[\,]_{X'_l \overline{X'_l}} \alpha'_1 \to [\,[\alpha'_1]_{X'_l} \overline{\alpha'_1}]_{\overline{X'_l}}$
7. $[\,]_{X'_l} \alpha'_1 \to [\,]_{X'_l \sim \alpha'_1}$
   $[\overline{\alpha'_1}]_{\overline{X'_l}} \to [\,]_{\overline{\alpha'_1} \sim \overline{X'_l}}$

8. $[\ ]_{\overline{\alpha_1'}\sim\overline{X_l'}X_l'\sim\alpha_1'} \to [\ [\alpha_1]_{X_l'}\overline{\alpha_1}]_{\overline{X_l'}}$
9. $[\ [\ ]_{X_l'}]_{\overline{X_l'}} \to [\ ]_{X_l''}\overline{X_l''}$
10. $[\ ]_{X_l''}\overline{X_l''}\alpha_2' \to [\ [\alpha_2']_{X_l''}\overline{\alpha_2'}]_{\overline{X_l''}}$
11. $[\ ]_{X_l''}\alpha_2' \to [\ ]_{X_l''\sim\alpha_2'}$
    $[\overline{\alpha_2'}]_{\overline{X_l''}} \to [\ ]_{\overline{\alpha_2'}\sim\overline{X_l''}}$
12. $[\ ]_{\overline{\alpha_2'}\sim\overline{X_l''}X_l''\sim\alpha_2'} \to [\ [\alpha_2]_{X_l'}\overline{\alpha_2}]_{\overline{X_l'}}$
13. $[\ ]_{X_l''}\overline{X_l''} \to [\ [\ ]_Y]_{\overline{Y}}$
14. $[\ ]_{X_l'}\overline{X_l'} \to [\ [\ ]_Y]_{\overline{Y}}$
15. $[\ [\ ]_X]_{\overline{X}} \to [\ ]_{\beta\overline{\beta}}$ (X does not correspond to a nonterminal matrix)
16. $[\ ]_{\beta\overline{\beta}} \to [\ [\ ]_\beta]_{\overline{\beta}}$
17. $[\ [\ ]_\beta]_{\overline{\beta}} \to [\ ]_{\beta\overline{\beta}}$

The simulation of a type-2 matrix can be done as follows:

- Whenever there exist proteins A, X, $\overline{A}$ and $\overline{X}$, by using two rules 1 applied in parallel, two protein complexes are created: $A \sim X$ and $\overline{A} \sim \overline{X}$. These protein complexes are able to interact, and by using rule 2, the proteins A and $\overline{A}$ break their bonds, while X and $\overline{X}$ are replaced by X_l and $\overline{X_l}$, marking the beginning of the simulation. Using two rules 3 in parallel, two new protein complexes are created: $A \sim X_l$ and $\overline{A} \sim \overline{X_l}$. This is followed by rule 4, where A and $\overline{A}$ break their bonds, and are replaced by either x (if $x \in T^*$) or x' (if $x \notin T^*$ and $x \neq \varepsilon$). Also by rule 4, the elements X_l and $\overline{X_l}$ are distributed between the two obtained membranes. Then, by applying rule 5, X_l and $\overline{X_l}$ are replaced by X_l' and $\overline{X_l'}$ in order to prevent replacing A's anymore. Now one of the following evolutions is possible:
  - Whenever there exists a protein α_1', rule 6 is applied to introduce the corresponding protein $\overline{\alpha_1'}$. Rules 7 and 8 use protein complexes, and replace the proteins α_1' and $\overline{\alpha_1'}$ by the proteins α_1 and $\overline{\alpha_1}$. Rule 9 is used to reach a configuration in which one of the following rule can be applied:
    - rule 10 if there exists the protein α_2', in order to introduce the corresponding protein $\overline{\alpha_2'}$; rules 11 and 12 use protein complexes, and replace the proteins α_2' and $\overline{\alpha_2'}$ by the proteins α_2 and $\overline{\alpha_2}$;
    - if it does not exist a protein α_2', rule 13 is used to replace X_l'' and $\overline{X_l''}$ by Y and $\overline{Y}$, respectively.
  - Whenever it does not exist proteins α_1' and α_2', then rule 14 is applied to replace X_l' and $\overline{X_l'}$ by Y and $\overline{Y}$, respectively.

  Rules 13 and 14 end a successfully simulation of a type-2 matrix, and return to the initial membrane structure. The proteins $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are introduced in order to be able to use the corresponding proteins $\overline{\alpha_1}$ and $\overline{\alpha_2}$ when simulating other matrices.
- If the symbol $A \in N_2$ is not present (i.e., we cannot apply rule 1), then rule 15 introduces two symbols β and $\overline{\beta}$ which lead to an infinite evolution (by using rules 16 and 17).

(ii) For each type-3 matrix $m'_l : (X \to Y, B^{(j)} \to \#)$, $X, Y \in N_1$ and $B^{(j)} \to \# \in F$, where $n_1 + 1 \leq l \leq n_1 + n_2$, $l \in lab_j$, $j = 1, 2$, we consider the rules:

18. $[\ [\ ]_X]_{\overline{X}} \to [\ ]_{X_l^{(j)}\overline{X_l^{(j)}}}$
19. $[B^{(j)}]_{X_l^{(j)}\overline{X_l^{(j)}}} \to [\ [\ ]_\beta B^{(j)}]_{\overline{\beta}}$
20. $[\ ]_{X_l^{(j)}\overline{X_l^{(j)}}} \to [\ [\ ]_Y]_{\overline{Y}}$

Rule 18 starts the simulation of a type-3 matrix by replacing X and $\overline{X}$ with $X_l^{(j)}$ and $\overline{X_l^{(j)}}$, and thereby remembering the index l of the matrix and the index j of the possibly present symbol $B^{(j)}$. This is followed by rule 19 that checks if the symbol $B^{(j)} \in N_2$ is present. If this is the case, $X_l^{(j)}$ and $\overline{X_l^{(j)}}$ are replaced by β and $\overline{\beta}$ which lead to an infinite evolution (by using rules 16 and 17). Regardless the presence of $B^{(j)}$, rule 20 is applied replacing $X_l^{(j)}$ and $\overline{X^{(j)}}$ by Y and $\overline{Y}$, thus successfully simulating a type-3 matrix, and returning to the initial membrane structure.

(iii) For a terminal type-4 matrix $m_l : (X \to a, A \to x)$ with $1 \leq l \leq n_1$, $X \in N_1$, $a \in T$, $A \in N_2$, $x \in T^*$ and $|x| \leq 2$, we consider the rules

21. $[\ ]_{X'_l\overline{X'_l}} \to [\ [a]]$
22. $[\ ]_{X'_l\overline{X'_l}} \to [\ [a]]$

We do not involve the protein $\overline{a}$, because $a \in T$. By replacing rule 14 with rule 21, and rule 13 by rule 22 in the sequence 1-17, a terminal type-4 matrix is faithfully simulated. The result of a simulation is the multiset of all the proteins present in the protein-protein interaction system.

We also investigate the computational power of the pair (phago, exo) of operations and prove its universality by using at most four membranes, while the length of proteins of (phago) and (exo) operations are at most two. We consider initially a system of three membranes. Comparing with Theorem 1, the higher number (four) of membranes is triggered by the use of (phago) operation.

Theorem 2. *A protein-protein interaction system with four membranes, proteins of length two and using rules of types* (bondin), (phago) *and* (exo) *has the same computational power as a Turing machine, i.e., are able to describe algorithmically any normal and abnormal biological evolution.*

Proof. We simulate a matrix grammar with appearance checking $G = (N, T, S, M, F)$ in the strong binary normal form. We construct a protein-protein interaction system Π with three membranes,

$$\Pi = (V, [\ [\ ]_2[\ ]_3]_1, \varepsilon, X, \overline{X}, \varepsilon, A, \overline{A}, R)$$

able to simulate a matrix grammar with appearance checking in the strong binary normal form. The symbols X, A correspond to the initial type-1 matrix $(S \to XA)$ with $X \in N_1, A \in N_2$. Let there be n_1 matrices of type-2 and type-4 labelled $1, \ldots, n_1$ and n_2 matrices of type-3 labelled $n_1 + 1, \ldots, n_1 + n_2$.

The finite alphabet V of proteins is defined as

$V = \{\beta, \overline{\beta}, \gamma, \overline{\gamma}, x, x', Y, \overline{Y}, a\} \cup \{B^{(j)}, \alpha_j, \overline{\alpha_j}, \alpha'_j, \overline{\alpha'_j} \mid 1 \leq j \leq 2\}$
$\cup \{A, \overline{A}, A_l, \overline{A_l} \mid A \in N_2, 1 \leq l \leq n_1 + n_2\}$
$\cup \{X, \overline{X}, X_l, \overline{X_l}, X'_l, \overline{X'_l}, X''_l, \overline{X''_l}, X_l^{(j)}, \overline{X_l^{(j)}} \mid$
$X \in N_1, 1 \leq l \leq n_1 + n_2, 1 \leq j \leq 2\}$.

The set R of rules is constructed as follows:

(i) For each type-2 matrix $m_l : (X \to Y, A \to x)$ with $1 \leq i \leq n_1$, $X, Y \in N_1$, $A \in N_2$, $x \in (N_2 \cup T)^*$ and $|x| \leq 2$, we consider the rules:

1. $[\overline{A}]_X \to [\ ]_{A \sim X}$
   $[\overline{A}]_{\overline{X}} \to [\ ]_{\overline{A} \sim \overline{X}}$
2. $[\ ]_{A \sim X}[\ ]_{\overline{A} \sim \overline{X}} \to [\ [\ [A_l]_{X_l}]_X]_{\overline{X}}$
3. $[\ [\ ]_X]_{\overline{X}} \to [\overline{A_l}]_{\overline{X_l}}$
4. $[A_l]_{X_l} \to [\ ]_{A_l \sim X_l}$
   $[\overline{A_l}]_{\overline{X_l}} \to [\ ]_{\overline{A_l} \sim \overline{X_l}}$
5. $[\ ]_{A_l \sim X_l}[\ ]_{\overline{A_l} \sim \overline{X_l}} \to [\ [\ [\ ]_{\overline{X'_l}}]_{X_l}]_{\overline{X_l}}$, if $x = \varepsilon$
   $[\ ]_{A_l \sim X_l}[\ ]_{\overline{A_l} \sim \overline{X_l}} \to [\ [\ [x]_{\overline{X'_l}}]_{X_l}]_{\overline{X_l}}$, if $x \in T^*$
   $[\ ]_{A_l \sim X_l}[\ ]_{\overline{A_l} \sim \overline{X_l}} \to [\ [\ [x']_{\overline{X'_l}}]_{X_l}]_{\overline{X_l}}$, otherwise
   (If $m_l : (X \to Y, A \to \alpha_1\alpha_2), \alpha_1 \in N_2, \alpha_2 \in T \cup \{\varepsilon\}$ then $x' = \alpha'_1\alpha_2$,
   and if $m_l : (X \to Y, A \to \alpha_1\alpha_2), \alpha_1, \alpha_2 \in N_2$ then $x' = \alpha'_1\alpha'_2$)
6. $[\ [\ ]_{X_l}]_{\overline{X_l}} \to [\ ]_{\overline{X'_l}}$
7. $[\alpha'_1]_{X'_l}[\ ]_{\overline{X'_l}} \to [\ [\ [\alpha'_1]_{X'_l}]_{X'_l}\overline{\alpha'_1}]_{\overline{X'_l}}$
8. $[\ [\ ]_{X'_l}\ \overline{\alpha'_1}]_{\overline{X'_l}} \to [\overline{\alpha'_1}]_{\overline{X'_l}}$
9. $[\alpha'_1]_{X'_l} \to [\ ]_{\alpha'_1 \sim X'_l}$
   $[\overline{\alpha'_1}]_{\overline{X'_l}} \to [\ ]_{\overline{\alpha'_1} \sim \overline{X'_l}}$
10. $[\ ]_{\alpha'_1 \sim X'_l}[\ ]_{\overline{\alpha'_1} \sim \overline{X'_l}} \to [\ [\ [\alpha_1]_{X'_l}]_{X'_l}\overline{\alpha_1}]_{\overline{X'_l}}$
11. $[\ [\ ]_{X'_l}\ \overline{\alpha_1}]_{\overline{X'_l}} \to [\overline{\alpha_1}]_{\overline{X''_l}}$
12. $[\alpha'_2]_{X''_l}[\ ]_{\overline{X''_l}} \to [\ [\ [\alpha''_2]_{X''_l}]_{X''_l}\overline{\alpha'_2}]_{\overline{X''_l}}$
13. $[\ [\ ]_{X''_l}\ \overline{\alpha'_2}]_{\overline{X''_l}} \to [\overline{\alpha'_2}]_{\overline{X''_l}}$
14. $[\alpha'_2]_{X''_l} \to [\ ]_{\alpha'_2 \sim X''_l}$
   $[\overline{\alpha'_2}]_{\overline{X''_l}} \to [\ ]_{\overline{\alpha'_2} \sim \overline{X''_l}}$
15. $[\ ]_{\alpha'_2 \sim X''_l}[\ ]_{\overline{\alpha'_2} \sim \overline{X''_l}} \to [\ [\ [\alpha_2]_{X''_l}]_{X''_l}\overline{\alpha_2}]_{\overline{X''_l}}$
16. $[\ [\ ]_{X''_l}\ \overline{\alpha_2}]_{\overline{X''_l}} \to [\overline{\alpha_2}]_{\overline{X''_l}}$
17. $[\ ]_{X'_l}[\ ]_{\overline{X'_l}} \to [\ [\ [\ ]_Y]_{X'_l}]_{\overline{X'_l}}$
   $[\ ]_{X''_l}[\ ]_{\overline{X''_l}} \to [\ [\ [\ ]_Y]_{X'_l}]_{\overline{X'_l}}$
18. $[\ [\ ]_{X'_l}]_{\overline{X'_l}} \to [\ ]_{\overline{Y}}$
19. $[\ ]_X[\ ]_{\overline{X}} \to [\ [\ [\ ]_\beta]_\beta]_{\overline{\beta}}$ (X does not correspond to a nonterminal matrix)
20. $[\ [\ ]_\beta]_{\overline{\beta}} \to [\ ]_{\overline{\beta}}$
21. $[\ ]_\beta[\ ]_{\overline{\beta}} \to [\ [\ [\ ]_\beta]_\beta]_{\overline{\beta}}$

The simulation of a type-2 matrix can be done as follows:

- Whenever there exist proteins A, X, $\overline{A}$ and $\overline{X}$, by using two rules 1 applied in parallel, two protein complexes are created: $A \sim X$ and $\overline{A} \sim \overline{X}$. These protein complexes are able to interact, and by using rule 2, the proteins A and $\overline{A}$ break their bonds and the proteins A_l, X_l, X and $\overline{X}$ are created, marking the beginning of the simulation. This is followed by rule 3, where X and $\overline{X}$ are replaced by $\overline{A_l}$, $\overline{X_l}$. Using two rules 4 in parallel, two new protein complexes are created: $A \sim X_l$ and $\overline{A} \sim \overline{X_l}$. This is followed by rule 5, where A and $\overline{A}$ break their bonds and are replaced by either x (if $x \in T^*$) or x' (if $x \notin T^*$ and $x \neq \varepsilon$). Also, by rule 5, the elements X_l and $\overline{X_l}$ are replaced by the proteins $\overline{X_l'}$, X_l and $\overline{X_l}$. Then, by applying rule 6, X_l and $\overline{X_l}$ are replaced by $\overline{X_l'}$ in order to prevent replacing A's anymore. Now one of the following evolutions is possible:
  - Whenever there exists a protein α_1', rule 7 is applied to introduce the corresponding protein $\overline{\alpha_1'}$, and is followed by rule 8 that simplifies the membrane structure. Rules 9 and 10 use protein complexes, and replace the proteins α_1' and $\overline{\alpha_1'}$ by the proteins α_1 and $\overline{\alpha_1}$, respectively. Rule 11 is used to reach a configuration in which one of the following rule can be applied:
    - rule 12 if there exists a protein α_2', in order to introduce the corresponding protein $\overline{\alpha_2'}$; it is followed by rule 13 that simplifies the membrane structure. Rules 14 and 15 use protein complexes, and replace the proteins α_2' and $\overline{\alpha_2'}$ by the proteins α_2 and $\overline{\alpha_2}$, respectively.
    - whenever it does not exist such a protein α_2', rules 17 and 18 are used to replace X_l'' and $\overline{X_l''}$ by Y and $\overline{Y}$, respectively.
  - Whenever it does not exist proteins α_1' and α_2', then rules 17 and 18 are applied to replace X_l' and $\overline{X_l'}$ by Y and $\overline{Y}$, respectively.

  Rules 17 and 18 end a successfully simulation of a type-2 matrix, and return to the initial membrane structure. The proteins $\overline{\alpha_1}$ and $\overline{\alpha_2}$ are introduced in order to be able to use the corresponding proteins $\overline{\alpha_1}$ and $\overline{\alpha_2}$ when simulating other matrices.
- Whenever the symbol $A \in N_2$ is not present (i.e., we cannot apply rule 1), then rule 19 is applied introduces two symbols β and $\overline{\beta}$ which lead to an infinite evolution (by using rules 20 and 21).

(ii) For each type-3 matrix $m_l' : (X \to Y, B^{(j)} \to \#)$, $X, Y \in N_1$ and $B^{(j)} \to \# \in F$, where $n_1 + 1 \leq i \leq n_1 + n_2$, $i \in lab_j$, $j = 1, 2$, we consider the rules:

22. $[\,]_X[\,]_{\overline{X}} \to [\,[\,[\,]_{X_l^{(j)}}]_X]_{\overline{X}}$
23. $[\,[\,]_X]_{\overline{X}} \to [\,]_{\overline{X_l^{(j)}}}$
24. $[B^{(j)}]_{X_l^{(j)}}[\,]_{\overline{X_l^{(j)}}} \to [\,[\,[B^{(j)}]_\beta]_\beta]_{\overline{\beta}}$
25. $[\,]_{X_l^{(j)}}[\,]_{\overline{X_l^{(j)}}} \to [\,[\,[\,]_Y]_\gamma]_{\overline{\gamma}}$
26. $[\,[\,]_\gamma]_{\overline{\gamma}} \to [\,]_{\overline{Y}}$

Rule 22 starts the simulation of a type-3 matrix by replacing X and $\overline{X}$ by $X_l^{(j)}$, X and $\overline{X}$, and thereby remembering the index l of the matrix and the

index j of the possibly present symbol $B^{(j)}$. This is followed by rule 23 in which the proteins X and $\overline{X}$ are replaced by $\overline{X_l^{(j)}}$. At this step we need to verify if the symbol $B^{(j)} \in N_2$ is present. If $B^{(j)}$ is present, rule 24 replaces $X_l^{(j)}$ and $\overline{X_l^{(j)}}$ with two proteins β placed on the inner membranes, while keeping the protein $\overline{\beta}$ on the outer membrane leads to an infinite evolution (by using rules 20 and 21). Regardless the presence of $B^{(j)}$, rule 25 is applied and $X_l^{(j)}$ and $\overline{X^{(j)}}$ are replaced by Y, γ and $\overline{\gamma}$ on the outer membrane. Rule 26 involves the creation of $\overline{Y}$, successfully simulating a type-3 matrix and returning to the initial membrane structure.

(iii) For a terminal type-4 matrix $m_l : (X \to a, A \to x)$ with $1 \leq i \leq n_1$, $X \in N_1$, $a \in T$, $A \in N_2$, $x \in T^*$ and $|x| \leq 2$, we consider the rules:

27. $[\,]_{X_l'}[\,]_{\overline{X_l'}} \to [\,[\,[\,]_a]_{X_l'}]_{\overline{X_l'}}$
    $[\,]_{X_l''}[\,]_{\overline{X_l''}} \to [\,[\,[\,]_a]_{X_l'}]_{\overline{X_l'}}$
28. $[\,[\,]_{X_l'}]_{\overline{X_l'}} \to [\,]$

We do not involve the protein $\overline{a}$, because $a \in T$. By replacing rules 17 and 18 with rules 27 and 28 in the sequence 1-21, a terminal type-4 matrix is faithfully simulated. The result of a simulation is the multiset of all the proteins present in the system of membranes.

5 Conclusion

In this work we proposed a computing system inspired by protein-protein interaction networks that uses a minimal number of membranes with respect to the cell movement operations that are initiated by proteins of different length. We proved that such protein-protein interaction networks can simulate all computable functions. We prove that protein-protein interaction networks have the same computational power as a Turing machine by using a rather small number of proteins having at most length two, where length is an abstract measure of complexity. Up to our knowledge, this is one of the first qualitative and quantitative approach in terms of an abstract measure of complexity (called length) studying the computational power of protein-protein interaction systems.

Inspired by the proteins of the living cell and by fact that membranes proteins are highly dynamic, several types of membrane proteins were previously investigated. In [8] there were defined several (biological inspired) transformations of membranes as pino (engulfing zero external membranes), exo (the process of expelling some material), phago (engulfing just one external membrane), mate (merging of membranes), drip (splitting off zero internal membranes), bud (splitting off one internal membrane). These operations were defined in terms of membrane computing, and used in defining classes of membrane systems where objects are placed on membranes [2]. Other approaches have considered proteins placed both on the membranes and in their compartments. In [17] the objects do not change their places (those bound on membranes remain there), while in [7, 9] the objects can move from compartments to membranes and back.

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. Alberts, A. Johnson, J. Lewis, M. Raff, K Roberts, P. Walter. *Molecular Biology of the Cell*, 5th edition, Garland Science, Taylor & Francis Group (2008).
2. B. Aman, G. Ciobanu. Mutual Mobile Membranes with Objects on Surface. *Natural Computing* **10**, 777–793 (2011).
3. B. Aman, G. Ciobanu. Describing the Immune System Using Enhanced Mobile Membranes. *Electronic Notes in Theoretical Computer Science* **194**(3), 5–18 (2008).
4. B. Aman, G. Ciobanu. Simple, Enhanced and Mutual Mobile Membranes *Transactions on Computational Systems Biology* **XI**, 26–44 (2009).
5. B. Aman, G. Ciobanu. *Mobility in Process Calculi and Natural Computing*. Natural Computing Series, Springer (2011).
6. B. Aman, G. Ciobanu. Computational Power of Protein Interaction Networks. *Lecture Notes in Computer Science* **7956**, 248–249 (2013).
7. R. Brijder, M. Cavaliere, A. Riscos-Núñez, G. Rozenberg, D. Sburlan. Membrane Systems with Proteins Embedded in Membranes. *Theoretical Computer Science* **404**, 26-39 (2008).
8. L. Cardelli. Brane Calculi. Interactions of Biological Membranes. *Lecture Notes in BioInformatics* **3082**, 257-278 (2004).
9. M. Cavaliere, S. Sedwards. Decision Problems in Membrane Systems with Peripheral Proteins, Transport and Evolution. *Theoretical Computer Science* **404**, 40–51 (2008).
10. G. Ciobanu. Modeling Cell-Mediated Immunity by Means of P Systems. *Applications of Membrane Computing, Natural Computing Series*, Springer, 159–180 (2006).
11. G. Ciobanu, S.N. Krishna. Enhanced Mobile Membranes: Computability Results. *Theory of Computing Systems* **48**, 715–729 (2011).
12. J.T. Gao, R. Guimera, H. Li, I.M. Pinto, M. Sales-Pardo, S. Wai, B. Rubinstein, R. Li. Modular Coherence of Protein Dynamics in Yeast Cell Polarity System. *Proceedings of the National Academy of Sciences* **198**, No. 18, 7647–7652 (2011).
13. T. Gramss, S. Bornholdt, M. Gross, M. Mitchel, Th. Pellizzari (Eds.). *Non-Standard Computation*. Wiley-VCH, Weinheim (1998).
14. V. Manca. *Infobiotics. Information in Biotic Systems*. Springer-Verlag (2013).
15. M.L. Minsky. *Computation: Finite and In5nite Machines*. Prentice-Hall, 1967.
16. K. Murphy, C. Weaver. *Janeway's Immunobiology*, 9th edition, Garland Publishing (2016).
17. A. Păun, B. Popa. P Systems with Proteins on Membranes. *Fundamenta Informaticae* **72**, 467–483 (2006).
18. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.). *The Oxford Handbook of Membrane Computing*. Oxford University Press (2010).
19. A. Salomaa. *Formal Languages*. Academic Press (1973).
20. T. Tan, D. Frenkel, V. Gupta, M. Deem. Length, Protein-protein Interactions, and Complexity. *Physica A* **350**, 52–62 (2005).
21. A.M. Turing. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society* **42**, 230–265 (1937).

Comparative Analysis of Statistical Model Checking Tools

Mehmet Emin Bakir[1], Marian Gheorghe[2], Savas Konur[2], and Mike Stannett[1]

[1] Department of Computer Science, The University of Sheffield
Regent Court, 211 Portobello, Sheffield, S1 4DP, UK
mebakir1@sheffield.ac.uk, m.stannett@sheffield.ac.uk
[2] School of Electrical Engineering and Computer Science, University of Bradford
West Yorkshire, Bradford, BD7 1DP, UK
m.gheorghe@bradford.ac.uk, s.konur@bradford.ac.uk

Abstract. Statistical model checking is a powerful and flexible approach for formal verification of computational models like P systems, which can have very large search spaces. Various statistical model checking tools have been developed, but choosing between them and using the most appropriate one requires a significant degree of experience, not only because different tools have different modelling and property specification languages, but also because they may be designed to support only a certain subset of property types. Furthermore, their performance can vary depending on the property types and membrane systems being verified. In this paper we evaluate the performance of various common statistical model checkers against a pool of biological models. Our aim is to help users select the most suitable SMC tools from among the available options, by comparing their modelling and property specification languages, capabilities and performances.

Keywords: Membrane systems, P systems, Statistical Model Checking, SMC tools, SMC performance on SBML models.

1 Introduction

In order to understand the structure and functionality of biological systems, we need methods which can highlight the spatial and time-dependent evolution of systems. To this end, scientists have started to utilize the computational power of machine-executable models, including implementations of membrane system models, to get a better and deeper understanding of the spatial and temporal features of biological systems [21]. In particular, the executable nature of computational models enables scientists to conduct experiments, in silico, in a fast and cheap manner.

The vast majority of models used for describing biological systems are based on ordinary differential equations (ODEs) [10], but scientists have recently started to use *computational models* as an alternative to mathematical modelling. The basis of such models is the *state machine*, which can be used to model numerous

variables and relate different system states (configurations) to one another [21]. There have been various attempts to model biological systems from a computational point of view, including the use of Boolean networks [31], Petri nets [45], the π-calculus [39], interacting state machines [25], L-systems [38] and variants of P systems (membrane systems) [5, 17, 23, 29, 33, 42]. These techniques are useful for investigating the qualitative features, as are their stochastic counterparts (e.g., stochastic Petri Nets [26] and stochastic P systems [8, 43]) are useful for investigating the quantitative features of computation models. More updated details regarding the use of membrane systems in modelling systems and synthetic biology applications can be found in [22].

Having built a model, the goal is typically to *analyse* it, so as to determine the underlying system's properties. Various approaches have been devised for analysing computational models. One widely used method, for example, based on generating the execution traces of a model, is *simulation*. Although the simulation approach is widely applicable, the large number of potential execution paths in models of realistic systems means that we can often exercise only a fraction of the complete trace set using current techniques. Especially for non-deterministic and stochastic systems each state may have more than one possible successor, which means that different runs of the same basic model may produce different outcomes [6]. Consequently, some computational paths may never be exercised, and their conformance to requirements never assessed.

Model checking is another widely recognized approach for analysis and verification of models, which has been successfully applied both to computer systems and biological system models. This technique involves representing each (desired or actual) property as a temporal logic formula, which is then verified against the model. It formally demonstrates the correctness of a system by means of strategically investigating the whole of the model's state space, considering all paths and guaranteeing their correctness [4, 15, 28]. Model checking has advantages over conventional approaches like simulation and testing, because it checks all computational paths and if the specified property is not satisfied it provides useful feedback by generating a counter-example (i.e. execution path) that demonstrates how the failure can occur [28].

Initially, model checking was employed for analysing *transition systems* used for describing discrete systems. A transition system regards time as discrete, and describes a set of states and the possible transitions between them, where each state represents some instantaneous configuration of the system. More recently, model checking has been extended by adding probabilities to state transitions (*probabilistic model checking*); in practice, such systems include discrete-time Markov chains (DTMC), continuous-time Markov chains (CTMC), and Markov decision processes (MDP). Probabilistic models are useful for verifying quantitative features of systems.

Typically, the model checking process comprises the following steps [4, 28]:

1. Describing the system model in a high-level modelling language, so as to provide an unambiguous representation of the input system.

2. Specifying the desired properties (using a property specification language) as a set of logical statements, e.g., temporal logic formulas.
3. Verifying whether each property is valid on the model. For non-probabilistic models the response is either 'yes' or 'no'. For probabilistic systems the response may instead be some estimate of the 'probability of correctness'.

"Exact" model checking considers whole state spaces while verifying a property, but if the model is relatively large, the verification process can be prohibitively resource intensive and time consuming which is known as 'state-space explosion' problem, so this approach can only be applied to a small number of biological models. Nonetheless, the intrinsic power of the approach has gained a good deal of attention from researchers, and model checking has been applied to various biological phenomena, including, for example, gene regulator networks (GRNs) and signal-transduction pathways [8, 13] (see [20] for a recent survey of the use of model checking in systems biology).

To overcome the state-space explosion problem, the *statistical* model checking (SMC) approach does not analyse the entire state space, but instead generates a number of independent simulation traces and uses statistical (e.g., Monte Carlo) methods to generate an approximate measure of system correctness. This approach does not guarantee the absolute correctness of the system, but it allows much larger models be verified (within specified confidence limits) in a faster manner [12, 37, 49, 51]. This approach allows verifying much larger models with significantly improved performance.

The number of tools using statistical model checking has been increasing steadily, as has their application to biological systems [14, 53]. Although the variety of SMC tools gives a certain amount of flexibility and control to users, each model checker has its own specific pros and cons. One tool may support a large set of property operators but perform property verifications slowly, while another may be more efficient at analysing small models, and yet another may excel at handling larger models. In such cases, the user may need to cover all of their options by using more than one model checker, but unfortunately the various SMCs generally use different modelling and property specification languages. Formulating properties using even a single SMC modelling language can be a cumbersome, error-prone, and time wasting experience for non-experts in computational verification (including many biologists), and the difficulties multiply considerably when more than one SMC needs to be used.

In order to facilitate the modelling and analysis tasks, several software suites have been proposed, such as Infobiotics Workbench [9] (based on stochastic P systems [10]) and kPWorkbench framework (based on kernel P systems [17]) [17, 34]. As part of the computational analysis, these tools employ more than one model checker. Currently, they allow only a manual selection of the tools, relying on the user expertise for the selection mechanism. These systems automatically translate the model and queries into the target model checker's specification language. While this simplifies the checking process considerably, one still has to know which target model checker best suits ones needs, and this requires a significant degree of experience. It is desirable, therefore, to introduce another

processing layer, so as to reduce human intervention by *automatically* selecting the best model checker for any given combination of P system and property query.

As part of this wider project (Infobiotics Workbench) to provide machine assistance to users, by automatically identifying the best model checker, we evaluate the performance of various statistical model checkers against a pool of biological models. The results reported here can be used to help select the most suitable SMC tools from the available options, by comparing their modelling and property specification languages, capabilities and performances (see also [7]).

Paper structure. We begin in Sect. 2 by describing some of the most commonly used SMC tools, together with their modelling and property-specification languages. Section 3 compares the usability of these tools in terms of expressibility of their property specification languages. In Sect. 4 we benchmark the performance of these tools when verifying biological models, and describe the relevant experiment settings. We conclude in Sect. 5 with a summary of our findings, and highlight open problems that warrant further investigation.

2 A Brief Survey of Current Statistical Model Checkers

In this section, we review some of the most popular and well-maintained statistical model checking tools, together with their modelling and property specification languages.

2.1 Tools

PRISM. PRISM (Probabilistic and Symbolic Model Checker) is a widely-used, powerful probabilistic model checker tool [27, 35]. It has been used for analysing a range of systems including biological systems, communication, multimedia and security protocols and many others [46]. It allows building and analysing several types of probabilistic systems including discrete-time Markov chains (DTMCs) and continuous-time Markov chains (CTMCs) with their 'reward' extension. PRISM can carry out both probabilistic model checking based on numerical techniques with exhaustive traversal of model, and statistical model checking with a discrete-event simulation engine [36, 46]. The associated modelling language, the PRISM language (a high-level state-based language), is the probabilistic variant of Reactive Modules [1, 35] (for a full description of PRISMs modelling language, see [46]), which subsumes several property specification languages, including PCTL, PCTL*, CSL, probabilistic LTL. However, statistical model checking can only be applied to a limited subset of properties; for example, it does not support steady-state and LTL-style path properties.

PRISM can be run via both a Graphical User Interface (GUI) or directly from the command line. Both options facilitate model checking process by allowing to modify a large set of parameters. The command line option is particularly useful when users need to run a large number of models. PRISM is open source software and is available for Windows, Linux and Mac OS X platforms.

PLASMA-Lab. PLASMA-Lab is a software platform for statistical model checking of stochastic systems. It provides a flexible plug-in mechanism which allows users to personalise their own simulator, and it also facilitates distributed simulations [11]. The tool has been applied to a range of problems, such as systems biology, rare events, motion planning and systems of systems [44].

The platform supports four modelling languages: Reactive Module Language (RML) implementation of the PRISM tool language, with two other variants of RML (see Table 1), and Biological Language [11, 44]. In addition, it provides a few simulator plug-ins which enable external simulators to be integrated with PLASMA-Lab, e.g., MATLAB/Simulink. The associated property specification language is based on Bounded Linear Temporal Logic (B-LTL) which bounds the number of states by number of steps or time units.

PLASMA-Lab can be run from a GUI or command line with plug-in system, and while it is not open source it can be embedded within other software programs as a library. It has been developed using the Java programming language, which provides compatibility with different operating systems.

Ymer. Ymer is a statistical model checking tool for verifying continuous-time Markov chains (CTMCs) and generalized semi-Markov processes (GSMPs). The tool supports parallel generation of simulation traces, which makes Ymer a fast SMC tool [50].

Ymer uses the PRISM language grammar for its modelling and property specification language. It employs the CSL formalism for property specification [48].

Ymer can be invoked via a command line interface only. It has been developed using the C/C++ programming language, and the source code is open to the public.

MRMC. MRMC is a tool for numerical and statistical model checking of probabilistic systems. It supports DTMC, CTMC, and using the reward extension of DTMC and CTMC [30].

The tool does not employ a high-level modelling language, but instead requires a sparse matrix representation of probabilities or rates as input. Describing systems in transition matrix format is very hard, especially for large systems, and external tools should be used to automatically generate the required inputs. Both PRISM and Performance Evaluation Process Algebra (PEPA) have extensions which can generate inputs for the MRMC tool [52]. The matrix representation also requires that state labels with atomic propositions be provided in another structure. Properties can be expressed with PCTL and CSL, and with their reward extensions.

MRMC is a command line tool. It has been developed using the C programming language, and the source code is publicly available. Binary distributions for Windows, Linux and Mac OS X are also available [41].

MC2. The MC2 tool enables statistical model checking of simulation traces, and can perform model checking in parallel.

MC2 does not need a modelling language, instead it imports simulation traces generated by external tools for stochastic and deterministic models. The tool uses probabilistic LTL with numerical constraints (PLTLc) for its property specification language, which enables defining numerical constraints on free variables [16].

MC2 can be executed only through its command line interface. The tool was developed using the Java programming interfaces and is distributed as a `.jar` file, therefore the source code is not available to public. The tool is bundled with a Gillespie simulator, called *Gillespie2*. As will be explained in the following section, it is possible to use Gillespie2 to generate simulation traces for the MC2 tool.

2.2 Modelling Languages

As part of the model checking process the system needs to be described in the target SMC modelling language. If the SMC tool relies on external tools, as in the case of MRMC and MC2, users will also have to learn the usage and modelling language of these external tools as well. For example, if users want to use the MRMC tool, they also have to learn how to use PRISM and how to model in the PRISM language.

Table 1 summarises the modelling languages associated with each SMC tool. The PLASMA and Ymer tools provide fair support for the PRISM language. MRMC expects a transition matrix input, but in practice, for large models, it is not possible to generate the transition matrix manually, so an external tool should be used for generating the matrix. MC2 also relies on external tools, because it does not employ a modelling language, instead it expects externally generated simulation traces. If users want to use the MC2 tool, they first have to learn a modelling language and usage of an appropriate simulation tool. For example, in order to use the Gillespie2 simulator as an external tool for MC2, the user should be able to describe their model using the Systems Biology Markup Language (SBML).

3 Usability

Model checking uses *temporal logics* as property specification languages. In order to query probabilistic features, probabilistic temporal logics should be used. Several probabilistic property specification languages exist, such as Probabilistic Linear Temporal Logic (PLTL) [4], probabilistic LTL with numerical constraints (PLTLc) [16] and Continuous Stochastic Logic (CSL) [2, 3, 36].

In order to ease the property specification process, frequently used properties, called *patterns*, have been identified by previous studies [18, 24]. Patterns represent recurring properties (e.g., something is *always* the case, something is

Table 1. Modelling languages and external dependency of SMC tools.

SMCs	Modelling Language(s)	Needs an External Tool?	External Tool Modelling Language
PRISM	PRISM language	NO	N/A
PLASMA-Lab	RML of PRISM, Adaptive RML (extension of RML for adaptive systems), RML with importance sampling, Biological Language	NO	N/A
Ymer	PRISM language	NO	N/A
MRMC	Transition matrix	YES, e.g., PRISM	PRISM language
MC2	N/A	YES, e.g., Gillespie2	Systems Biology Markup Language (SBML)

possibly the case), and are generally represented by natural language-like keywords. An increasing number of studies have been conducted to identify appropriate pattern systems for biological models [23, 32, 40]. Table 2 lists various popular patterns [24], giving a short description and explaining how they can be represented using existing temporal logic operators.

Table 2. Property patterns

Patterns	Description	Temporal Logic
Existence	ϕ_1 will eventually hold, within the $\bowtie p$ bounds.	$P_{\bowtie p}[F\ \phi_1]$ or $P_{\bowtie p}[true\ \mathrm{U}\ \phi_1]$
Until	ϕ_1 will hold continuously until ϕ_2 eventually holds, within the $\bowtie p$ bounds.	$P_{\bowtie p}[\phi_1\ \mathrm{U}\ \phi_2]$
Response	If ϕ_1 holds, then ϕ_2 must hold within the $\bowtie p$ bounds.	$P_{\geq 1}[G\ (\phi_1 \rightarrow (P_{\bowtie p}[F\ \phi_2]))]$
Steady-State (Long-run)	In the long-run ϕ_1 must hold, within the $\bowtie p$ bounds.	$S_{\bowtie p}[\phi_1]$ or $P_{\bowtie p}[FG\ (\phi_1)]$
Universality	ϕ_1 continuously holds, within the $\bowtie p$ bounds.	$P_{\bowtie p}[G\ \phi_1]$ or $P_{\overline{\bowtie}(1-p)}[(F\ (\neg\phi_1)]$

Key. ϕ_1, and ϕ_2 are state formulas; $\bowtie$ is one of the relations in $\{<, >, \leq, \geq\}$; $p \in [0, 1]$ is a probability with rational bounds; and $\overline{\bowtie}$ is negation of inequality operators. $P_{\bowtie p}$ is the *qualitative* operator which enables users to query qualitative features, those whose result is either 'yes' or 'no'. In order to query *quantitative* properties, $P_{=?}$ (quantitative operator) can be used to returns a numeric value which is the probability that the specified property is true.

The SMCs investigated here employ different grammar syntaxes for property specification, which makes it harder to use other tools at the same time. Although Ymer uses the same grammar as PRISM, it excludes some operators, such as the Always (G) operator. In addition, different SMCs tools may support different sets of probabilistic temporal logics. In the following, we compare the expressibility of their specification languages, by checking if the properties can be defined using just one temporal logic operator (directly supported (DS)), which will be easier

for practitioners to express; or as a combination of multiple operators (indirectly supported (IS)); or not supported at all (not supported (NS)). Qualitative and quantitative operators, with five property patterns which are identified as widely used by [24], are listed in Table 3.

Table 3. Specifying various key patterns using different SMC tools.

SMCs	Qualitative Operator	Quantitative Operator ($\mathbf{P}_{=?}$)	Existence	Until	Response	Steady -State	Universality
PRISM	DS	DS	DS	DS	**NS**	**NS**	DS
PLASMA-Lab	**NS**	**NS**	DS	DS	IS	IS	DS
Ymer	DS	DS	DS	DS	**NS**	**NS**	IS
MRMC	DS	**NS**	DS	DS	IS	DS	DS
MC2	DS	DS	DS	DS	IS	IS	DS

Key. DS = Directly Supported; IS = Indirectly Supported; NS = Not Supported.

The PRISM, Ymer and MC2 tools directly support both Qualitative and Quantitative operators, but MRMC supports only the Qualitative operator. While PLASMA-Lab does not allow these operators to be expressed directly with B-LTL, the verification outputs contain information about the probability of the property, hence users can interpret the results. Existence, Until and Universality properties are directly supported by all SMCs, except that Ymer does not employ an operator for Universality patterns (it needs to be interpreted using the Not (!) and Eventually (F) operators, i.e. it is indirectly supported). There is no single operator to represent the Response pattern directly, but it is indirectly supported by PLASMA-Lab, MRMC and MC2. The Steady-State pattern can be either represented by one operator, S, or two operators, F and G. Only the MRMC tool employs the S operator to allow Steady-State to be expressed directly, while PLASMA-Lab and MC2 allow it to be expressed indirectly.

4 Experimental Findings

The wide variety of SMC tools gives a certain flexibility and control to users, but practitioners need to know which of the tools is best for their particular models and queries. The expressibility of the associated modelling and specification languages are not the only criteria, because SMC performance may also vary depending on the nature of the models and property specifications. We have therefore conducted a series of experiments to determine the capabilities and performance of the most commonly used tools [7]. The experiments are conducted on Intel i7-2600 CPU @ 3.40GHz 8 cores, with 16GB RAM running on Ubuntu 14.04.

We tested each of the five tools against a representative selection of 465 biological models (in SBML format) taken from the BioModels database [19] (as modified in [47] to fix the stochastic rate constants of all reactions to 1).

The models tested ranged in size from 2 species and 1 reaction, to 2631 species and 2824 reactions, and each tool/model pair was tested against five different property specification patterns [24], namely Existence, Until, Response, Steady-State and Universality. To this end, we first developed a tool for translating SBML models to SMCs modelling languages, and translating property patterns to the corresponding SMC specification languages. For each SMC, the number of simulation traces was set to 500, and the depth of each trace was set to 5000.

The time required for each run is taken to be the combined time required for model parsing, simulation and verification. Each SMC/model/pattern combination was tested three times, and the figures reported here give the average total time required. Where an SMC depends on external tools, we also added the external tool execution time into the total execution time. In particular, therefore, the total times reported for verifying models with MRMC and MC2 tools are not their execution times only, but include the time consumed for generating transition matrices and simulation traces, respectively. We used the PRISM tool for generating transition matrices to MRMC, and the Gillespie2 for generating simulation traces to MC2. Where the external tool failed to generate the necessary input for its corresponding SMC, we have recorded the SMC as being incapable of verifying the model. In order to keep the experiment tractable, if an SMC required more than 1 hour to complete the run, we halted the process and again recorded the model as unverifiable.

Table 4 shows the experiment results. The SMCs and the property patterns are represented in first column and row respectively. The *Verified* columns under each pattern show the number of models that could be verified by the corresponding SMC. The *Fastest* column shows the number of models for which the corresponding SMC was the fastest tool.

Table 4. The number of model/pattern combinations verified by each SMC tool.

	Existence		**Until**		**Response**		**Steady -State**		**Universality**	
	Verified	**Fastest**	Verified	**Fastest**	Verified	**Fastest**	Verified	**Fastest**	Verified	**Fastest**
PRISM	337	15	435	84	NS	NS	NS	NS	370	57
PLASMA -Lab	465	143	465	54	465	390	465	392	465	80
Ymer	439	304	439	324	NS	NS	NS	NS	439	325
MRMC	75	0	72	0	75	17	57	11	77	0
MC2	458	3	458	3	458	58	458	62	458	3

Key. NS = Not Supported.

The results show that SMC tool capabilities vary depending on the queried properties. For example, PRISM was only able to verify 337 models against Existence, and 435 and 370 models against Until and Universality, respectively. The main reason PRISM failed to verify *all* of the models is that it expects user to increase the depth of the simulation traces, and otherwise it cannot verify the unbounded properties with a reliable approximation. In contrast, PLASMA-Lab

was able to verify all of the models within 1 hour. Ymer could verify 439 models for those patterns it supports, thus failing to complete 26 models in the time available. MRMC was able to verify relatively few models, because it relied on the PRISM model checker to construct the model and export the associated transition matrices. Especially for relatively large models PRISM crashed while generating these matrices (we believe this is related to its CU Decision Diagram (CUDD) library). MC2 was able to verify 458 models against all of the patterns tested, and only failed for 7 of them.

Regarding the *Fastest* data, Ymer was fastest for most model/pattern pairs (where those patterns were supported). It was fastest for 304 models against Existence, 324 models against Until and 325 models against Universality. PLASMA-Lab fastest for only 143 models against Existence, 54 against Until and 80 against Universality, but did particularly well against Response and Steady-State patterns (390 and 392 respectively), where it was only competing with MRMC and MC2. PRISM was fastest for only 15 models with Existence, 84 models with Until and 57 models with Universality. MC2 was fastest for only 3 models (using Gillespie2) against Existence, Until and Universality, although it did better with 58 models against Response, and 62 against Steady-State. Finally, MRMC (with PRISM dependency) was slower than other tools for Existence, Until and Universality, but did better handling Response (fastest for 17 models) and Steady-State (11 models).

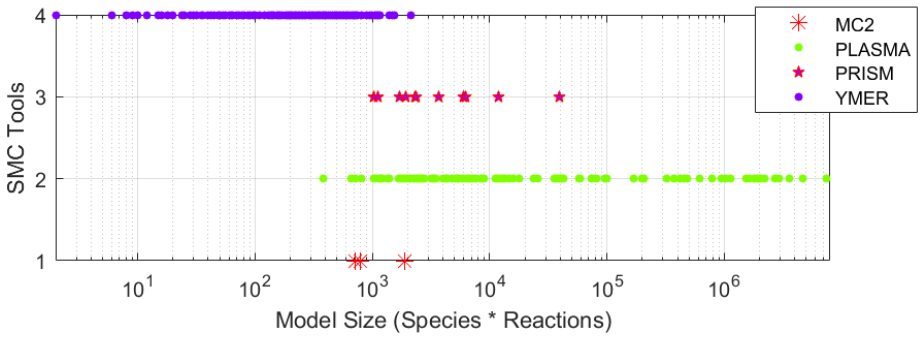


Fig. 1. The range of model sizes for which each SMC tool was fastest.

Key. The x-axis (log scale) indicates model size, where we take 'size' to be the product of species count and reaction count. The graph shows results for each model/tool combination when tested against Existence; the results for other patterns may differ.

Figure 1 illustrates how the size of the model affects which SMC acts as the fastest verifier (in this case, when verifying against an Existence pattern), where we take the 'size' of the model to be the product of species count and reactions. Ymer is the fastest tool for verification of relatively small sized models (the minimum model size verified fastest by Ymer was 2, maximum 2128, median 137.5), whereas PLASMA-Lab is the fastest for larger models (min = 380, max =

7429944, median = 11875). PRISM (min = 1023, max = 39770, median = 2304) and MC2 (min = 722, max = 1892, median = 800) are best for medium-sized models. The results show that for both the 231 smallest models (size ranging from 2 to 374), and the 55 largest models (size = 39770 to 7429944), we can uniquely identify the fastest SMC tool, but for remaining 179 medium-sized models (size = 380 to 39770), there is no obvious 'winner' (note, however, that these results only relate to verification against the Existence pattern, and the results for other patterns may be different).

5 Conclusion

The experimental results clearly show that certain SMC tools are best for certain tasks, but there are also situations where the best choice of SMC is far less clear-cut, and it is not surprising that users may struggle to select and use the most suitable SMC tool for their needs. Users need to consider the modelling language of tools and the external tools they may rely on, and need detailed knowledge as to which property specification operators are supported, and how to specify them. Even then, the tool may still fail to complete the verification within a reasonable time, whereas another tool might be able to run it successfully.

These factors make it extremely difficult for users to know which model checker to choose, and point to a clear need for automation of the SMC-selection process. We are currently working to identify novel methods and algorithms to automate the selection of best SMC tool for a given computational model (more specifically for P system models) and property patterns. We aim to enable the integration of our methods within larger software platforms, e.g., IBW and kP-Workbench, and while this is undoubtedly a challenging task, we are encouraged by recent developments in related areas, e.g., the automatic selection of stochastic simulation algorithms [47].

References

1. Alur, R., Henzinger, T.A.: Reactive modules. Form. Methods Syst. Des. 15, 7–48 (Jul 1999), http://dx.doi.org/10.1023/A:1008739929481
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Model-checking continuous-time Markov chains. ACM Trans. Comput. Logic 1(1), 162–170 (Jul 2000), http://doi.acm.org/10.1145/343369.343402
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model-checking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering 29(6), 524–541 (June 2003)
4. Baier, C., Katoen, J.P.: Principles of model checking. The MIT Press (2008)
5. Bakir, M.E., Ipate, F., Konur, S., Mierla, L., Niculescu, I.: Extended simulation and verification platform for kernel P systems. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing, pp. 158–178. Lecture Notes in Computer Science, Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-14370-5_10

6. Bakir, M.E., Konur, S., Gheorghe, M., Niculescu, I., Ipate, F.: High performance simulations of kernel P systems. 2014 IEEE 16th International Conference on High Performance Computing and Communications (HPCC) (2014)
7. Bakir, M.E., Stannett, M.: Selection criteria for statistical model checking. In: UCNC'16 Workshop on Membrane Computing (WMC '16) (2016), submitted
8. Bernardini, F., Gheorghe, M., Romero-Campero, F.J., Walkinshaw, N.: A hybrid approach to modeling biological systems. In: Eleftherakis, G., Kefalas, P., Păun, G., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 4860, pp. 138–159. Springer Berlin Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-77312-2_9
9. Blakes, J., Twycross, J., Romero-Campero, F.J., Krasnogor, N.: The Infobiotics Workbench: An integrated in silico modelling platform for systems and synthetic biology. Bioinformatics 27(23), 3323–3324 (Dec 2011)
10. Blakes, J., Twycross, J., Konur, S., Romero-Campero, F.J., Krasnogor, N., Gheorghe, M.: Infobiotics Workbench: A P systems based tool for systems and synthetic biology. In: Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.) Applications of Membrane Computing in Systems and Synthetic Biology, Emergence, Complexity and Computation, vol. 7, pp. 1–41. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-03191-0_1
11. Boyer, B., Corre, K., Legay, A., Sedwards, S.: Plasma-lab: A flexible, distributable statistical model checking library. In: Proceedings of Quantitative Evaluation of Systems - 10th International Conference, QEST 2013, Buenos Aires, Argentina, August 27-30, 2013. pp. 160–164 (2013)
12. Buchholz, P.: A new approach combining simulation and randomization for the analysis of large continuous time Markov chains. ACM Trans. Model. Comput. Simul. 8(2), 194–222 (Apr 1998), http://doi.acm.org/10.1145/280265.280274
13. Carrillo, M., Góngora, P.A., Rosenblueth, D.A.: An overview of existing modeling tools making use of model checking in the analysis of biochemical networks. Frontiers in Plant Science 3(155), 1–13 (2012)
14. Cavaliere, M., Mazza, T., Sedwards, S.: Statistical model checking of membrane systems with peripheral proteins: Quantifying the role of estrogen in cellular mitosis and DNA damage. In: Applications of Membrane Computing in Systems and Synthetic Biology, Emergence, Complexity and Computation, vol. 7, pp. 43–63. Springer International Publishing (2014), http://dx.doi.org/10.1007/978-3-319-03191-0_2
15. Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)
16. Donaldson, R.;Gilbert, D.: A Monte Carlo model checker for Probabilistic LTL with numerical constraints. Tech. rep., University of Glasgow, Department of Computing Science (2008)
17. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierla, L.: Model checking kernel P systems. In: Membrane Computing, Lecture Notes in Computer Science, vol. 8340, pp. 151–172. Springer Berlin Heidelberg (2014), http://dx.doi.org/10.1007/978-3-642-54239-8_12
18. 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, New York, NY, USA (1999), http://doi.acm.org/10.1145/302405.302672
19. The European Bioinformatics Institute. http://www.ebi.ac.uk/, [Online; accessed 08/01/15]
20. Fisher, J., Piterman, N.: Model checking in biology, pp. 255–279. Springer Verlag (2014)

21. Fisher, J., Henzinger, T.A.: Executable cell biology. Nat Biotech 25(11), 1239–1249 (2007)
22. Frisco, P., Gheorghe, M., Pérez-Jiménez, M.J. (eds.): Emergence, Complexity and Computation, vol. 7. Springer International Publishing (2014)
23. Gheorghe, M., Konur, S., Ipate, F., Mierla, L., Bakir, M.E., Stannett, M.: An integrated model checking toolset for kernel P systems. In: Rozenberg, G., Salomaa, A., Sempere, M.J., Zandron, C. (eds.) Membrane Computing: 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers. pp. 153–170. Springer International Publishing, Cham (2015), http://dx.doi.org/10.1007/978-3-319-28475-0_11
24. Grunske, L.: Specification patterns for probabilistic quality properties. In: Proceedings of the 30th International Conference on Software Engineering. pp. 31–40. ICSE '08, ACM, NY, USA (2008), http://doi.acm.org/10.1145/1368088.1368094
25. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8(3), 231 – 274 (1987), http://www.sciencedirect.com/science/article/pii/0167642387900359
26. Heiner, M., Gilbert, D., Donaldson, R.: Petri nets for systems and synthetic biology. In: Proceedings of the Formal Methods for the Design of Computer, Communication, and Software Systems 8th International Conference on Formal Methods for Computational Systems Biology. pp. 215–264. SFM'08, Springer-Verlag, Berlin, Heidelberg (2008), http://dl.acm.org/citation.cfm?id=1786698.1786706
27. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 441–444. Springer Berlin Heidelberg (2006)
28. Huth, M., Ryan, M.: Logic in computer science: Modelling and reasoning about systems. Cambridge University Press (2004), http://books.google.co.uk/books?id=eUggAwAAQBAJ
29. Ibarra, O.H., Păun, G.: Membrane computing: A general view. Ann Eur Acad Sci. EAS Publishing House, Liege pp. 83–101 (2006)
30. Katoen, J.P., Zapreev, I.S., Hahn, E.M., Hermanns, H., Jansen, D.N.: The ins and outs of the probabilistic model checker MRMC. In: Quantitative Evaluation of Systems (QEST). pp. 167–176. IEEE Computer Society (2009)
31. Kauffman, S.A.: Metabolic stability and epigenesis in randomly constructed genetic nets. Journal of Theoretical Biology 22, 437–467 (1969)
32. Konur, S., Gheorghe, M.: A property-driven methodology for formal analysis of synthetic biology systems. IEEE/ACM Transactions on Computational Biology and Bioinformatics 12(2), 360–371 (March 2015)
33. Konur, S., Gheorghe, M., Dragomir, C., Mierla, L., Ipate, F., Krasnogor, N.: Qualitative and quantitative analysis of systems and synthetic biology constructs using P systems. ACS Synthetic Biology 4(1), 83–92 (2015), http://dx.doi.org/10.1021/sb500134w, pMID: 25090609
34. kPWorkbench. http://kpworkbench.org/, [Online; accessed 08/01/15]
35. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic symbolic model checker. In: Computer performance evaluation: modelling techniques and tools, pp. 200–204. Springer Berlin Heidelberg (2002)
36. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic model checking. In: Proceedings of the 7th International Conference on Formal Methods for Performance Evaluation. pp. 220–270. SFM'07, Springer-Verlag, Berlin, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1768017.1768023

37. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: Proceedings of Runtime Verification: First International Conference, RV 2010, St. Julians, Malta, November 1-4. pp. 122–135. Springer, Berlin, Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-16612-9_11
38. Lindenmayer, A., Jürgensen, H.: Grammars of development: Discrete-state models for growth, differentiation, and gene expression in modular organisms. In: Lindenmayer Systems: Impacts on Theoretical Computer Science, Computer Graphics, and Developmental Biology, pp. 3–21. Springer, Berlin, Heidelberg (1992), http://dx.doi.org/10.1007/978-3-642-58117-5_1
39. Milner, R.: Communicating and mobile systems: The Pi-calculus. Cambridge University Press, New York, NY, USA (1999)
40. 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 (Aug 2008), http://dx.doi.org/10.1093/bioinformatics/btn275
41. Markow Reward Model Checker (MRMC). http://www.mrmc-tool.org/, [Online; accessed 18/02/15]
42. Păun, G.: Introduction to membrane computing. In: Ciobanu, G., Păun, G., Pérez-Jiménez, M. (eds.) Applications of Membrane Computing, pp. 1–42. Natural Computing Series, Springer Berlin Heidelberg (2006), http://dx.doi.org/10.1007/3-540-29937-8_1
43. Pérez-Jiménez, M.J., Romero-Campero, F.J.: P systems, a new computational modelling tool for systems biology. In: Priami, C., Plotkin, G. (eds.) Transactions on Computational Systems Biology VI, pp. 176–197. Springer, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11880646_8
44. Plasma-Lab. https://project.inria.fr/plasma-lab/, [Online; accessed 18/02/15]
45. Reisig, W.: The basic concepts. In: Understanding Petri Nets, pp. 13–24. Springer Berlin Heidelberg (2013), http://dx.doi.org/10.1007/978-3-642-33278-4_2
46. Probabilistic and Symbolic Model Checker (PRISM). http://www.prismmodelchecker.org/, [Online; accessed 08/01/15]
47. Sanassy, D., Widera, P., Krasnogor, N.: Meta-stochastic simulation of biochemical models for systems and synthetic biology. ACS Synthetic Biology 4(1), 39–47 (2015), http://dx.doi.org/10.1021/sb5001406, pMID: 25152014
48. Ymer website. http://www.tempastic.org/ymer/, [Online; accessed 25/8/15]
49. Younes, H., Kwiatkowska, M., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. International Journal on Software Tools for Technology Transfer (STTT) 8(3), 216–228 (2006)
50. Younes, H.L.S.: Ymer: A statistical model checker. In: Proceedings of the 17th International Conference on Computer Aided Verification. pp. 429–433. CAV'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11513988_43
51. Younes, H.L.S., Simmons, R.G.: Probabilistic verification of discrete event systems using acceptance sampling. In: Proceedings of Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, pp. 223–235. Springer, Berlin, Heidelberg (2002), http://dx.doi.org/10.1007/3-540-45657-0_17
52. Zapreev, I.S., Jansen, C.: Markov reward model checker manual, http://www.mrmc-tool.org/downloads/MRMC/Specs/MRMC_Manual.pdf

53. Zuliani, P.: Statistical model checking for biological applications. International Journal on Software Tools for Technology Transfer 17(4), 527–536 (2014), http://dx.doi.org/10.1007/s10009-014-0343-0

P Colonies with evolving environment

Lucie Ciencialová, Luděk Cienciala, and Petr Sosík

Research Institute of the IT4Innovations Centre of Excellence,
Faculty of Philosophy and Science, Silesian University in Opava, Czech Republic
{lucie.ciencialova,ludek.cienciala,petr.sosik}@fpf.slu.cz

Abstract. We study two variants of P colonies with dynamic environment changing due to an underlying 0L scheme: systems with two objects inside agents that can only consume objects, and systems with one object inside agents using rewriting and communication rules. We show that the first kind of systems with one consumer agent can generate all sets of natural numbers computed by partially blind register machines. The second kind of systems with two agents with rewriting/consuming rules is computationally complete. Finally, we demonstrate that P colonies with one such agent with checking program can simulate catalytic P systems with one catalyst, and consequently, another relation to partially blind register machines is established.

Keywords: P colony, catalytic P system, 0L scheme, computational completeness, partially blind register machine

1 Introduction

P colony was introduced in [10] as a very simple variant of membrane systems inspired by so-called *colonies* of formal grammars. See [14] for more information about membrane systems and [11] for details on grammar systems theory.

There are three basic entities in the P colony model: objects, agents and the environment. A P colony is composed of agents, each containing a collection of objects embedded in a membrane. The objects can be placed in the environment, too. Agents are equipped with programs composed of rules that allow interactions of objects. The number of objects inside each agent is set by definition and it is usually very low – 1, 2 or 3. The environment of a P colony serves as a communication channel for agents: an agent is able to affect the behaviour of another agent by sending objects via the environment. There is also a special type of *environmental objects* denoted by e which are present in the environment in an unlimited number of copies.

A specific variant of P colony called *eco-P colony* with two object inside each agent, where the environment can change independently from the agents, was introduced in [1]. The evolution of the environment is controlled by a 0L scheme applying context-free rules in parallel to all possible objects in the environment which are unused by the agents in the current step of computation.

The activity of agents is based on rules that can be rewriting, communication or checking; these three types were introduced in [10]. Furthermore, generating, consuming and transporting rules were introduced in [5].

Rewriting rule $a \to b$ allows an agent to rewrite (evolve) one object a placed inside the agent to object b.

Communication rule $c \leftrightarrow d$ exchanges one object c placed inside the agent for object d from the environment.

Checking rule r_1/r_2, where each of r_1, r_2 is a rewriting or a communication rule, sets a priority between these two rules. The agent tries to apply the first rule and if it cannot be performed, the agent executes the second rule.

Generating rule $a \to bc$ creates two objects b, c from one object a.

Consuming rule $ab \to c$ rewrites two objects a, b to one object c.

Transporting rule of the form $(a\ in)$ or $(a\ out)$ is used to transport one object from the environment into the agent, or from the agent to the environment, respectively. The rule is always associated with a consuming/generating rule to keep a constant number of objects inside the agent.

The rules are combined into programs in such a way that all objects inside the agent are affected by execution of the rules in every step. Consequently, the number of rules in the program is the same as the number of objects inside the agent.

The programs that contain consuming rules are called consuming programs and the programs with generating rules are called generating programs. The agent that only contains consuming resp. generating programs is called consumer resp. sender.

P colonies with senders and consumers without evolving environment were studied in [5] and the authors proved their computational completeness (in the Turing sense), as well as computational completeness of P colonies with senders and consumers with 0L scheme for the environment. Many papers were devoted to P colonies with rewriting and communication rules without evolving environment, e.g., [4, 6, 12], and there are two book chapters in [3] and [14] describing this topic.

In this paper we focus on P colonies with evolving environment. The paper is structured as follows: The second section is devoted to definitions and notations used in the paper. The third section contains results obtained during studies of P colonies with consumers only. In the fourth section we study P colonies with one object inside the agent and rewriting/communication rules. The fifth section studies the relation of P colonies with evolving environment and catalytic P systems. The paper concludes with a summary of presented results and possibilities of future work.

2 Definitions

Throughout the paper we assume the reader to be familiar with basic of formal automata and language theory. We introduce notation used in the paper.

We use $\mathbb{N}\cdot\mathsf{RE}$ to denote the family of recursively enumerable sets of natural numbers and $\mathbb{N}$ to denote the set of natural numbers.

Σ is a notation for the alphabet. Let Σ^* be set of all words over alphabet Σ (including the empty word ε). For the length of the word $w \in \Sigma^*$ we use the notation $|w|$ and the number of occurrences of symbol $a \in \Sigma$ in w is denoted 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 $f : V \to N$; f assigns to each object in V its multiplicity in M. The cardinality of M, denoted by $card(M)$, is defined by $card(M) = \sum_{a \in V} f(a)$.

Any multiset of objects M with the set of objects $V = \{a_i, \ldots a_n\}$ can be represented as a string w over alphabet V with $|w|_{a_i} = f(a_i)$; $1 \leq i \leq n$. Obviously, all words obtained from w by permuting the letters can also represent M, and ε represents the empty multiset. Because of string representation we can denote the set of all multisets over the set of objects V by V^*.

The mechanism of evolution of the environment is based on a *0L scheme.* It is a pair (Σ, P), where Σ is the alphabet of 0L scheme and P is the set of context-free rules fulfilling the condition $\forall a \in \Sigma\ \exists \alpha \in \Sigma^*$ such that $(a \to \alpha) \in P$. For $w_1, w_2 \in \Sigma^*$ we write $w_1 \Rightarrow w_2$ if $w_1 = a_1 a_1 \ldots a_n, w_2 = \alpha_2 \alpha_2 \ldots \alpha_n$, for $a_i \to \alpha_i \in P, 1 \leq i \leq n$

A *register machine* [13] is a construct $M = (m, H, l_0, l_h, P)$ where:
- m is the number of registers, H is a set of instruction labels,
- l_0 is the initial/start label, l_h is the final label,
- P is the finite set of instructions injectively labelled with the elements from the given set H.

The instructions of the register machine are of the following forms:

l_1 : $(ADD(r), l_2, l_3)$ Add 1 to the contents of the register r and non-deterministically choose one of the instructions (labelled with) l_2 or l_3 to proceed with.

l_1 : $(SUB(r), l_2, l_3)$ If the register r is not empty, then subtract 1 from its contents and go to instruction l_2, otherwise proceed to instruction l_3.

l_h : *HALT* Stop the machine. The final label l_h is only assigned to this instruction.

Without loss of generality, one can assume that in each *ADD*-instruction $l_1 : (ADD(r), l_2, l_3)$ and in each conditional *SUB*-instruction $l_1 : (SUB(r), l_2, l_3)$ the labels l_1, l_2, l_3 are mutually distinct. The register machine M computes a set $N(M)$ of numbers in the following way: we start with all registers empty (hence storing the number zero) with the instruction with label l_0 and we proceed to apply the instructions as indicated by the labels (and made possible by the contents of registers). If we reach the halt instruction, then the number stored at that time in the register 1 is said to be computed by M and hence it is introduced in $N(M)$. Because of non-determinism of the machine, $N(M)$ can be an infinite set. The family of sets of numbers computed by register machines is denoted by $\mathbb{N}\cdot RM$.

Theorem 1 ([13]). $\mathbb{N}\cdot RM = \mathbb{N}\cdot\mathsf{RE}$.

Moreover, we call a register machine *partially blind* if we interpret the subtract instructions in the following way: $l_1 : (SUB(r); l_2; l_3)$ - if register r is not empty, then subtract one from its contents and non-deterministically choose to continue with one of the instructions l_2 or l_3; if register r is empty when attempting to decrement register r, then the program ends without yielding a result.

When the register machine reaches the final state, the result obtained in the first register is only taken into account if the remaining registers are empty. The family of sets of non-negative integers generated by partially blind register machines is denoted by $\mathbb{N}\cdot RM_{pb}$. The partially blind register machine accepts a proper subset of $\mathbb{N}\cdot\mathsf{RE}$:

Theorem 2 ([7]).
$\mathbb{N}\cdot RM_{pb} = \mathbb{N}\cdot MAT$, *where* $\mathbb{N}\cdot MAT$ *is the Parikh image of the class of languages generated by matrix grammars without appearance checking.*

2.1 Catalytic P systems

Definition 1. *An* extended catalytic P system *of degree* $m \geq 1$ *is a construct*

$$\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_0) \text{ where}$$

1. O *is the alphabet of objects;*
2. $C \subseteq O$ *is the alphabet of catalysts;*
3. μ *is a membrane structure of degree* m *with membranes labeled in a one-to-one manner with the natural numbers* $1, 2, \ldots, m$*;*
4. $w_1, \ldots, w_m \in O^*$ *are the multisets of objects initially present in the* m *regions of* μ*;*
5. R_i, $1 \leq i \leq m$, *are finite sets of evolution rules over* O *associated with the regions* $1, 2, \ldots, m$ *of* μ*; these evolution rules are of the forms* $ca \to cv$ *or* $a \to v$*, where* c *is a catalyst,* a *is an object from* $O \setminus C$*, and* v *is a string from* $((O \setminus C) \times \{here, out, in\})^*$*;*
6. $i_0 \in \{0, 1, \ldots, m\}$ *indicates the output region of* Π*.*

The membrane structure and the multisets in Π constitute a *configuration* of the P system; the *initial configuration* is given by the initial multisets $w_1, \ldots, w_m$. A transition between configurations is governed by the application of the evolution rules, which is done in the maximally parallel way, i.e., only applicable multisets of rules which cannot be extended by further rules have to be applied to the objects in all membrane regions.

The application of a rule $u \to v$ in a region containing a multiset M results in subtracting from M the multiset identified by u, and then in adding the multiset identified by v. The objects can eventually be transported through membranes due to the targets *in* and *out*. We refer to [14] for further details and examples.

The P system continues with applying multisets of rules in the maximally parallel way until there remain no applicable rules in any region of Π. Then the system halts. We consider the number of objects from $O \setminus C$ contained in the output region i_0 at the moment when the system halts as the *result* of the underlying computation of Π. The system is called *extended* since the catalytic objects in C are not counted to the result of a computation. The set of results of all computations possible in Π is called the set of natural numbers *generated by* Π and it is denoted by $N(\Pi)$.

The problem how to count the catalysts in the case of generating catalytic P systems can be avoided if using external output, i.e., the output is sent to the environment, indicated by $i_0 = 0$. Denote by $NOP_m(cat_k)$ the class of sets of natural numbers generated by catalytic P systems with external output, with at most m membranes and at most k catalysts.

2.2 Generalized P colonies

Definition 2. *A P colony with capacity $c \geq 1$ is the structure*

$$\Pi = (\Sigma, e, f, v_E, D_E, B_1, \ldots, B_n), \text{ where}$$

- *Σ is the alphabet of the colony, its elements are called objects,*
- *e is the basic (environmental) object of the colony, $e \in \Sigma$,*
- *f is final object of the colony, $f \in \Sigma$,*
- *v_E is the initial content of the environment, $v_E \in (\Sigma - \{e\})^*$,*
- *D_E is 0L scheme (Σ, P_E), where P_E is the set of context free rules,*
- *B_i, $1 \leq i \leq n$, are the agents, every agent is the structure $B_i = (o_i, P_i)$, where o_i is the multiset over Σ, it defines the initial state (content) of the agent B_i and $|o_i| = c$ and $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$ is the finite set of programs of three types:*
  - *(1) generating program with generating rules $a \to bc$ and transporting rules d out - the number of generating rules is the same as the number of transporting rules.*
  - *(2) consuming program with consuming rules $ab \to c$ and transporting rules d in - the number of consuming rules is the same as the number of transporting rules.*
  - *(3) rewriting/communication program can contain three types of rules:*
    - *$a \to b$, called a rewriting rule,*
    - *$c \leftrightarrow d$, called a communication rule,*
    - *r_1/r_2, called a checking rule; each of r_1, r_2 is a rewriting or a communication rule.*

Every agent has only one type of programs. The agent with generating programs is called *sender* and the agent with consuming programs is called *consumer*. The capacity of a P colony with senders and consumers must be an even number.

The *initial configuration* of a P colony is the $(n+1)$-tuple $(o_1, \ldots, o_n, v_E)$, with the interpretation of the symbols $o_1, \ldots, o_n, v_E$ as in Definition 2. In general,

the *configuration* of the P colony Π is defined as $(n+1)$-tuple $(w_1, \ldots, w_n, w_E)$, where w_i represents the multiset of objects inside the i-th agent, $|w_i| = c,\ 1 \leq i \leq n$, and $w_E \in (\Sigma - \{e\})^*$ is the multiset of objects different from e placed in the environment.

At each step of the (parallel) computation every agent tries to find one of its programs to apply. If the number of applicable programs is higher than one, the agent non-deterministically chooses one. At each step of a computation, the set of active agents executing a program must be maximal, i.e., no further agent can be added to it.

By applying programs, the P colony passes from one configuration to another configuration. Objects in the environment unaffected by any program in the given step are rewritten by the 0L scheme D_E. A sequence of configurations starting from the initial configuration is called a *computation.* A configuration is *halting* if the P colony has no applicable program. Each halting computation has associated a *result* – the number of copies of the final object placed in the environment in a halting configuration.

$$N(\Pi) = \{|w_E|_f \mid (o_1, \ldots, o_n, v_E) \Rightarrow^* (w_1, \ldots, w_n, w_E)\},$$

where $(o_1, \ldots, o_n, v_E)$ is the initial configuration, $(w_1, \ldots, w_n, w_E)$ is the final configuration, and $\Rightarrow^*$ denotes reflexive and transitive closure of $\Rightarrow$.

Let us denote $NEPCOL(i, j, k, u, v, w)$ the family of the sets computing by P colonies with at most $j \geq 1$ agents with $i \geq 1$ objects inside the agent and with at most $k \geq 1$ programs associated with each agent such that:

- $u = check$ if the P colony uses rewriting/communication rules with checking rules
- $u = no\text{-}check$ if the P colony uses rewriting/communication rules without checking rules
- $u = s/c/sc$ if the P colony contains only sender / only consumer / both sender and consumer agents
- $v = pas$ if the rules of 0L scheme are of type $a \to a$ only,
- $v = act$ if the set of rules of 0L scheme disposes of at least one rule of another type than $a \to a$,
- $w = ini$ if the environment or agents contain initially objects different from e, otherwise w is omitted,

If a numerical parameter is unbounded, we denote it by a $*$.

In [5] the authors deal with P colonies with senders and consumers with "passive" environment, they show that

$$NEPCOL(2, 3, *, sc, pas) = \mathbb{N}\cdot\mathsf{RE}.$$

In [1] there are results of P colonies with "active" environment:

$$NEPCOL(2, 2, *, c, act, ini) = \mathbb{N}\cdot\mathsf{RE}$$

$$NEPCOL(2, 2, *, sc, pas, ini) \supseteq \mathbb{N}\cdot RM_{pb}.$$

Other results are shown for P colonies with "passive" environment and rewriting/communication rules and with only one object inside the agent in [2]

$$NEPCOL(1, 4, *, check, pas) = \mathbb{N}\cdot \mathsf{RE}$$

and in [5]

$$NEPCOL(1, 6, *, no\text{-}check, pas) = \mathbb{N}\cdot \mathsf{RE}.$$

3 P colonies with senders and consumers

In this section we study computational power of P colonies with two objects inside the agent - consumer and with active environment. We extend the previous results reported in [1].

Theorem 3. $\mathbb{N}\cdot RM_{pb} \subseteq NEPCOL(2, 1, *, c, act, ini)$.

Proof. Consider register machine $M = (m, H, l_0, l_h, P)$. All labels from the set H are objects in the P colony. The content of register r is represented by the number of copies of objects a_r placed in the environment.

We construct the P colony $\Pi = (\Sigma, e, a_1, l_0 d, D_E, B_1)$ with:

- $\Sigma = \{l_i, l_i', l_i'', \overline{l_i}, \underline{l_i}, L_i \mid l_i \in H\} \cup \{a_i \mid 1 \leq i \leq m\} \cup \{e, d, f\}$,
- $B_1 = (de, P_1)$.

At the beginning of computation there is object l_0 and auxiliary object d in the environment. Object l_0 corresponds with the initial label of the instructions of M.

An instruction $l_i = (ADD(r), l_j, l_k)$ is simulated by the environment and the agent by using the following rules and programs:

ENV :

$A : l_i \to a_r l_j d;$
$B : l_i \to a_r l_k d;$

B_1 :

$1 : \langle de \to e; d\ in \rangle;$

The computation is done in such a way that 0L scheme works in the environment, it executes adding one to the content of register r (generate one copy of object a_r – the rule labelled A or B) and generating of the objects l_j or l_k, label of instruction which will be executed in the next steps of computation of register machine M. Agent B_1 consumes object d from the environment. Notice that there is at most one copy of d in the environment.

An instruction $l_i : (SUB(r), l_j, l_k)$ is simulated by following rules and programs:

ENV :	B_1 :
$C : l_i \to \overline{l_i} l_i'$;	$2 : \langle de \to e; \overline{l_i}\ in \rangle$;
$D : l_i' \to l_i''$;	$3 : \langle \overline{l_i} e \to L_j; a_r\ in \rangle$;
$E : l_i'' \to l_j \underline{l_k}$;	$4 : \langle \overline{l_i} e \to f; e\ in \rangle$;
$F : \underline{l_j} \to \overline{l_j} d$;	$5 : \langle L_j a_r \to e; \underline{l_j}\ in \rangle$;
$G : \underline{\overline{l_k}} \to l_k d$;	$6 : \langle L_j a_r \to e; \underline{\overline{l_k}}\ in \rangle$;
	$7 : \langle \underline{l_j} e \to e; d\ in \rangle$;
	$8 : \langle \underline{\overline{l_k}} e \to e; d\ in \rangle$;
	$9 : \langle \overline{f} e \to f; e\ in \rangle$;

If there is the object l_i (the label of SUB-instruction) in the environment, the 0L scheme generates (using the rule labelled C) the object $\overline{l_i}$. This is the message for the agent B_1 to try to consume one copy of object a_r from the environment (try to subtract one from the content of register r.)

If the agent is successful (using program labelled 3), then in the next step the agent consumes $\overline{l_j}$ or $\overline{l_k}$ and the computation will follow in simulation of instruction labelled l_k or l_j.

If the agent does not consume object a_r – there is no a_r in the environment or the agent non-deterministically chooses program 4 instead of the applicable program 3 – the agent generates object f and the computation will never end because the program 9 will be applicable.

For the halting instruction we add the rule $l_h \to l_h$ to the 0L scheme, as well as "passive" 0L rules for other objects which are not changed by other environmental rules (for example $e \to e$, $a_r \to a_r, \dots$). Whenever the simulated register machine executes the HALT instruction, neither object d nor object $l_i \in H \setminus \{l_h\}$ will appear in the environment any more and the computation will halt.

P colony Π starts its computation with object l_0 in the environment and the simulation of the instruction labelled l_0. By the rules and programs it places and deletes from the environment the objects a_r and halts its computation only after object l_h appears in the environment. The result of a computation is the number of copies of object a_1 placed in the environment at the end of the computation. No other halting computation can be executed in the constructed P colony.

4 P colonies with rewriting/communication rules

In this section we deal with P colonies with "active" environment and with one object inside agents. We prove that such a P colony with two agents can generate every recursively enumerable set of natural numbers.

Theorem 4. $NEPCOL(1, 2, *, no\text{-}check, act, ini) = \mathbb{N} \cdot \mathsf{RE}$.

Proof. Let us consider register machine $M = (m, H, l_0, l_h, P)$. For all labels from the set H we construct corresponding objects in the P colony Π. The content of register i will be represented by the number of copies of objects a_i placed in the environment.

We construct the P colony $\Pi = (\Sigma, e, a_1, l_0 d, D_E, B_1)$ with:

- $\Sigma = \{l_i, l_i', l_i'', \overline{l_i}, \overline{\overline{l_i}}, \underline{l_i}, l_i^1, l_i^2, l_i^3, l_i^4, M_i, M_i^1, M_i^2, M_i^3, M_i^4, N_i, N_i^1, N_i^2, N_i^3, N_i^4 \mid l_i \in H\} \cup \{a_i \mid 1 \leq i \leq m\} \cup \{e, d, f, g\}$,
- $B_1 = (e, P_1)$
- $B_2 = (e, P_2)$.

The object l_0 corresponds to the label of the first instruction executed by the register machine.

The instruction $l_i : (ADD(r), l_j, l_k)$ will be simulated by the environment and the agent B_1 by using following rules and programs:

ENV :	B_1 :
$A : l_i \to a_r l_j'$;	$1 : \langle e \leftrightarrow d \rangle$;
$B : l_i \to a_r l_k'$;	$2 : \langle d \to e \rangle$;
$C : l_j' \to l_j d$;	
$D : l_k' \to l_k d$;	

The simulation of the ADD-instruction starts by rewriting the object l_i to object a_r (adds one to the content of register r) by executing rule A or B and object l_j (rule C) or object l_k (rule D).

The instruction $l_i : (SUB(r), l_j, l_k)$ is simulated by using the following rules and programs:

ENV :	B_1 :	B_2 :
$E : l_i \to l_i^1 \overline{l_i}$;	$3 : \langle e \leftrightarrow g \rangle$;	$7 : \langle e \leftrightarrow \overline{l_i} \rangle$;
$F : l_i^1 \to l_i^2 d$;	$4 : \langle g \to f \rangle$;	$8 : \langle \overline{l_i} \to \overline{\overline{l_i}} \rangle$;
$G : l_i^2 \to l_i^3$;	$5 : \langle f \leftrightarrow \overline{\overline{l_i}} \rangle$;	$9 : \langle \overline{\overline{l_i}} \leftrightarrow a_r \rangle$;
$H : l_i^3 \to l_i^4 d$;	$6 : \langle \overline{\overline{l_i}} \to e \rangle$;	$10 : \langle a_r \to g \rangle$;
$I : l_i^4 \to M_i N_i$;		$11 : \langle g \leftrightarrow N_i \rangle$;
$J : M_i \to M_i^1$;		$12 : \langle N_i \to e \rangle$;
$K : M_i^1 \to M_i^2$;		$13 : \langle \overline{\overline{l_i}} \leftrightarrow M_i \rangle$;
$L : M_i^2 \to M_i^3$;		$14 : \langle M_i \to e \rangle$;
$M : M_i^3 \to M_i^4$;		
$N : M_i^4 \to l_j d$;		
$O : N_i \to N_i^1$;		
$P : N_i^1 \to N_i^2$;		
$Q : N_i^2 \to N_i^3$;		
$R : N_i^3 \to N_i^4$;		
$S : N_i^4 \to l_k d$;		

The simulation starts with rule E generating objects $l_i^1 \overline{l_i}$. Object l_i^1 keeps the environmental rules busy for 6 steps until actions of agents are completed. Object $\overline{l_i}$ causes the agent B_2 to generate object $\overline{\overline{l_i}}$ and, in turn, to consume object a_r.

Table 1 shows the example of the simulation of the SUB-instruction when the register to be decremented stores a value greater than zero. Symbol w in the environment (column w_E) denotes an arbitrary multiset containing objects in $\{a_i \mid 1 \leq i \leq m\}$. Symbol ? in the last row means that the applicable rule depends on the next instruction to be simulated labelled l_j.

step	w_E	w_1	w_2	applicable rules and programs		
0.	$l_i d a_r w$	e	e	E	1	–
1.	$l_i^1 \overline{l_i} a_r w$	d	e	F	2	7
2.	$l_i^2 d a_r w$	e	$\overline{l_i}$	G	1	8
3.	$l_i^3 a_r w$	d	$\overline{\overline{l_i}}$	H	2	9
4.	$l_i^4 d \overline{\overline{l_i}} w$	e	a_r	I	1	10
5.	$M_i N_i \overline{\overline{l_i}} w$	d	g	J	2	11
6.	$M_i^1 g \overline{\overline{l_i}} w$	e	N_i	K	3	12
7.	$M_i^2 L_k \overline{\overline{l_i}} w$	g	e	L	4	–
8.	$M_i^3 \overline{\overline{l_i}} w$	f	e	M	5	–
9.	$M_i^4 f w$	$\overline{\overline{l_i}}$	e	N	6	–
10.	$l_j d f w$	e	e	?	1	–

Table 1. The simulation of the *SUB*-instruction when register r is not empty

Table 2 shows the example of the simulation of the *SUB*-instruction when register to be decremented stores zero. Symbol w in the environment (column w_E) denotes an arbitrary multiset containing objects in $\{a_i \mid 1 \leq i \leq m, i \neq r\}$. Symbol ? in the last row means that the applicable rule depends on the next instruction to be simulated labelled l_k.

step	w_E	w_1	w_2	applicable rules and programs		
0.	$l_i d w$	e	e	E	1	–
1.	$l_i^1 \overline{l_i} w$	d	e	F	2	7
2.	$l_i^2 d w$	e	$\overline{l_i}$	G	1	8
3.	$l_i^3 w$	d	$\overline{\overline{l_i}}$	H	2	–
4.	$l_i^4 d w$	e	$\overline{\overline{l_i}}$	I	1	–
5.	$M_i N_i w$	d	$\overline{\overline{l_i}}$	O	2	13
6.	$N_i^1 g \overline{\overline{l_i}} w$	e	M_i	P	3	14
7.	$N_i^2 \overline{\overline{l_i}} w$	g	e	Q	4	–
8.	$N_i^3 \overline{\overline{l_i}} w$	f	e	R	5	–
9.	$N_i^4 f w$	$\overline{\overline{l_i}}$	e	S	6	–
10.	$l_k d f w$	e	e	?	1	–

Table 2. The simulation of the *SUB*-instruction when register r is empty

For instruction l_h :*HALT* we add the rule $T: \ l_h \to l_h$. The environment does not produce object d any more. The computation halts as no agent can execute a program. The result is the number of objects a_1 placed in the environment and it corresponds to the result of a successful computation in the register machine.

5 P colonies versus catalytic P systems

In this section we study the relation of P colonies with communication rules, and catalytic P systems in the generating mode. We show that any catalytic P system with one catalyst can be simulated by a P colony with checking rules and with one agent. As a consequence, the relation of these P colonies and partially blind register machines is established.

Theorem 5. *For an arbitrary extended catalytic P system Π with one catalyst there exists a P colony Π' with checking rules and one agent containing one object such that $N(\Pi) = N(\Pi')$.*

Proof. Note first that in the generating case, due to the existence of flattening procedures described, e.g., in [8], one can assume without loss of generality that the catalytic P system $\Pi = (O, C, [], w, R, i_0)$ has a single membrane and that its output is collected in the environment, i.e., $i_0 = 0$.

We construct the P colony $\Pi' = (\Sigma, e, f, w \setminus C, D_E, B_1)$ as follows: each step of Π is simulated by two steps of Π'.

1. In the first step the P colony checks whether the P system can apply at least one rule (if not, the P colony halts in the next step). Then both catalytic and non-catalytic rules are randomly applied in the environment, rewriting all objects to their primed versions.
2. In the second step the agent checks whether at most one catalytic rule was applied (if not, a trap symbol is produced). Simultaneously the environment signals whether there had been unused objects in the previous step to which catalytic rules could have been applied. If yes, and simultaneously if no catalytic rules was chosen, the agent produces the trap symbol in the next step (which is the first step of a new cycle). All primed objects are rewritten back to their original versions.

Formally, let

$$O_c = \{a \in (O \setminus C) \mid \exists u : (ca \to cu) \in R \text{ and } \nexists v : (a \to v) \in R\}$$

be the set of objects to which only catalytic rules can be applied. Let the alphabet of the P colony be constructed as

$$\Sigma = O \cup \{a' \mid a \in (O \setminus C)\} \cup \{a'' \mid a \in O_c\} \cup \{c, f, k, n, s, t\}$$

such that the newly added symbols do not appear in O. Define further a mapping $\varphi : (O \setminus C) \times \{here, out\} \longrightarrow \Sigma$ as follows:

$$\varphi(a, dest) = \begin{cases} f & \text{if } dest = out, \\ a' & \text{otherwise.} \end{cases}$$

Let the agent of the P colony adopt the form $B_1 = (n, P_1)$. We construct the rules of the environment and the agent as specified in the following table:

Step	Programs of the agent	Rules of the environment
1.	$\langle n \leftrightarrow s / n \rightarrow k \rangle$ $\langle c \rightarrow k \rangle$	$\{a \rightarrow \varphi(u)n \mid (a \rightarrow u) \in R\} \cup$ $\{a \rightarrow \varphi(u)c \mid (ca \rightarrow cu) \in R\} \cup$ $\{a \rightarrow a''n \mid a \in O_c\} \cup \{s \rightarrow \varepsilon\}$
2.	$\langle k \leftrightarrow c / k \leftrightarrow n \rangle$	$\{a' \rightarrow a \mid a \in O\} \cup$ $\{a'' \rightarrow as \mid a \in O_c\} \cup$ $\{c \rightarrow t,\ n \rightarrow \varepsilon\}$

In addition, the agent contains programs $\langle k \leftrightarrow t \rangle$, $\langle t \rightarrow t \rangle$, $\langle s \rightarrow s \rangle$, and the environment contains rules $a \rightarrow a$ for each object $a \in \Sigma$ not affected by any of the rules described above.

In the first simulation step the agent contains object n and there is no s in the environment, hence $n \rightarrow k$ is the only applicable rule. Simultaneously the environment simulates the application of rules of the P system to all objects to which any rule can be applied. If more than one rule is applicable to an object, one is randomly chosen. Each application of a catalytic rule produces also one object c and a non-catalytic rule produces n.

When the simulated P system sends an object to the output region 0 by applying a rule with target *out*, the P colony produces instead the final object f, hence the number of final objects equals the number of objects in the output region of the simulated P system.

In the second step the agent checks whether any rule of the P system was applicable, i.e., whether there is at least one c or n in the environment. If not, the agent can run no program and the colony halts. Simultaneously, if there were more than one object c in the environment, indicating that more than one catalytic rule has been applied, then at least one object c is rewritten by the environment to t. Object t is the trap object – whenever it appears, the colony never halts and produces no output (note the programs $\langle k \leftrightarrow t \rangle$ and $\langle t \rightarrow t \rangle$ of the agent).

Finally, rules of the form $a'' \rightarrow as$ in the environment produce the object s. If there is n in the agent and s in the environment in the next step, this means that no catalytic rule was applied even if there were unused objects to which such a rule was applicable, i.e., the maximal parallelism condition of the P system was broken. In this situation the rule $n \leftrightarrow s$ is selected, and s enters the agent where it acts as another trap symbol.

The whole cycle repeats again and again for each simulated step of the P system Π. As explained above, any incorrect simulation results in the appearance of the trap symbol s or t which forces the P colony to run forever and produce no result. Hence only correct simulation results can be produced, and the P colony halts if and only if so does the simulated P system. We can conclude that $N(\Pi) = N(\Pi')$.

Corollary 1. $\mathbb{N} \cdot RM_{pb} \subseteq NEPCOL(1, 1, *, check, act, ini)$.

Proof. The paper [9] demonstrates that $\mathbb{N}\cdot RM_{pb} \subseteq NOP_1(cat_1)$. Then the statement follows by Theorem 5.

6 Conclusions

In this paper we presented the results obtained during the research of P colonies with evolving environment. We have shown that P colonies with one consumer agent can generate all sets of natural numbers computable by partially blind register machines. If we place two agents with one object inside each of them and with no-checking rewriting/communication programs into the evolving environment, the obtained P colony is computationally complete in the Turing sense.

Finally, we have demonstrated that when checking programs are allowed, then a P colony with one agent is sufficient to simulate a catalytic P system with one catalyst. Consequently, these colonies can again generate sets of natural numbers computable by partially blind register machines. We conjecture that this simulation principle is extensible to the case of more catalysts / more agents.

It remains an *open problem* whether any of the presented results can be further strengthened, or whether the class $\mathbb{N}\cdot RM_{pb}$ is also an upper bound of the generative power of the mentioned types of P colonies.

Acknowledgments. This work was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602, and by the Silesian University in

Opava under the Student Funding Scheme, project SGS/13/2016.

References

1. Cienciala, L., Ciencialová, L.: Eco-P colonies. In: Păun, Gh., Pérez-Jiménez, M., Riscos-Núñez, A. (eds.) Pre-Proceedings of the 10th Workshop on Membrane Computing, Curtea de Arges, Romania. pp. 201–209 (2009)
2. Cienciala, L., Ciencialová, L., Kelemenová, A.: On the number of agents in P colonies. In: Membrane Computing, LNCS, vol. 4860, pp. 193–208. Springer (2007)
3. Cienciala, L., Ciencialová, L.: P colonies and their extensions. In: Kelemen, J., Kelemenová, A. (eds.) Computation, Cooperation, and Life – Essays Dedicated to Gheorghe Paun on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 6610, pp. 158–169. Springer-Verlag, Berlin Heidelberg (2011)
4. Ciencialová, L., Cienciala, L., Csuhaj-Varjú, E., Kelemenová, A., Vaszil, G.: On very simple P colonies. In: Proceeding of The Seventh Brainstorming Week on Membrane Computing. vol. 1, pp. 97–108 (2009)
5. Ciencialová, L., Kelemenová, A., Csuhaj-Varjú, E., Vaszil, Gy.: Variants of P colonies with very simple cell structure. Int. J. of Computers, Communications & Control 3(IV), 224–233 (2009)

6. Freund, R., Oswald, M.: P colonies working in the maximally parallel and in the sequential mode. In: Proceedings - Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2005. pp. 419–426 (Sept 2005)
7. Freund, R., Ibarra, O.H., Păun, Gh., Yen, H.C.: Matrix languages, register machines, vector addition systems. In: Proceeding of the Third Brainstorming Week on Membrane Computing. vol. 1, pp. 155–167. E.T.S. de Ingeniería Informática, Universidad de Sevilla (2005)
8. Freund, R., Leporati, A., Mauri, G., Porreca, A.E., Verlan, S., Zandron, C.: Flattening in (tissue) P systems. In: Membrane Computing, LNCS, vol. 8340, pp. 173–188. Springer (2013)
9. Freund, R., Sosík, P.: On the power of catalytic P systems with one catalyst. In: Membrane Computing, LNCS, vol. 9504, pp. 137–152. Springer (2015)
10. Kelemen, J., Kelemenová, A., Păun, Gh.: Preview of P colonies: A biochemically inspired computing model. In: Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX). pp. 82–86. Boston, Mass (2004)
11. Kelemen, J., Kelemenová, A.: A grammar-theoretic treatment of multiagent systems. Cybern. Syst. 23(6), 621–633 (Nov 1992)
12. Kelemen, J., Kelemenová, A.: On P colonies, a biochemically inspired model of computation. Proc. of the 6th International Symposium of Hungarian Researchers on Computational Intelligence pp. 40–56 (2005)
13. Minsky, M.L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
14. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)

Continuation Passing Semantics for Membrane Systems

Gabriel Ciobanu[1] and Eneia Nicolae Todoran[2]

[1] Romanian Academy, Institute of Computer Science, Iaşi, Romania
[2] Technical University, Department of Computer Science, Cluj-Napoca, Romania
gabriel@info.uaic.ro, eneia.todoran@cs.utcluj.ro

Abstract. Membrane systems are described by a language in which multisets of objects are encapsulated in hierarchical structures of compartments. The language provides primitives for parallel communication of objects across membranes and a primitive for membrane creation. The behaviour of each membrane is specified by means of multiset rewriting rules. We provide a compositional semantics for membrane systems by using the continuation passing style and metric spaces.

1 Introduction

Membrane systems are presented in [11], and several of their applications in [5]. In this paper we investigate membrane systems by using continuation passing style in the tradition of programming language semantics. By using metric spaces, we provide a denotational semantics for a simple concurrent language in which computations are specified by means of multiset rewriting rules distributed into membrane-delimited compartments. The language provides a primitive for membrane creation, but we ignore other operations that can be used for expressing the dynamics of systems with active membranes, such as membrane division or membrane dissolution [11]. However, we are confident that most of the membrane computing concepts that are investigated in the literature can receive a denotational semantics by using the techniques that we employ in this paper.

The essential tools in this semantic investigation are the classic continuation-passing style in which the control is passed explicitly in the form of a continuation [2], as well as the continuations for concurrency [9, 12]; both are used to describe in a compositional manner the behaviour of dynamic hierarchical systems. An important feature of this semantic approach is *compositionality*: the meaning of a compound construction is determined solely on the basis of the meanings of its components.

The aim of this paper is to offer a semantic investigation in the area of membrane computing by using methods and tools from the tradition of programming language semantics. We assume the reader is familiar with the denotational (compositional) style of assigning meanings to language constructs. The mathematical methodology of metric semantics used in this paper is presented in detail in the monograph [4]. We use a language $\mathcal{L}_{MC}$ in which the syntactic constructions are

statements and *programs*. When referring to the behaviour of an $\mathcal{L}_{MC}$ program we use the term *execution*. An $\mathcal{L}_{MC}$ program comprises a list of membrane declarations, which is similar to a list of class declarations in an object-oriented language. The list of membrane declarations is followed by an $\mathcal{L}_{MC}$ statement which is executed in the skin membrane.

In previous papers we have defined the operational semantics for membrane systems [3, 6], and proved certain semantic properties. In [7, 8] we have applied continuation semantics for concurrency [12] in providing denotational semantics for a simple multiset rewriting concurrent language. Subsequently, in [10], we have investigated the semantics of a language where computations are distributed to membrane-delimited regions, but without any feature for transmembrane communication. The languages described in [7, 10] are all subsumed by the language $\mathcal{L}_{MC}$ investigated in this paper. The language $\mathcal{L}_{MC}$ combines various advanced control concepts, including maximal parallelism, nondeterministic behaviour and the sequencing of phases within each computation step. In order to achieve a compositional approach of the complex control structures incorporated in $\mathcal{L}_{MC}$ we employ a domain of continuations combined with a standard powerdomain construction to represent nondeterministic behaviour. An element of a powerdomain is a collection of sequences of observables representing dynamic membrane structures. We use a fixed point construction to express the semantics of multiset rewriting computations.

2 Mathematical Preliminaries

Mathematical preliminaries are those presented in [10]. A notation $(x \in)X$ introduces the set X with typical element x ranging over X. A *multiset* is a generalization of a set. Intuitively, a multiset is a collection in which an element may occur more than once. One can represent the concept of a multiset of elements of type X by using functions from $X \to \mathbb{N}$, or partial functions from $X \to \mathbb{N}^+$, where $\mathbb{N}^+ = \mathbb{N} \setminus \{0\}$. One can represent a multiset $m \in [X]$ by enumerating its elements between parentheses '[' and ']'. The elements in a multiset are *not* ordered; to give yet another intuition, a multiset is an unordered list of elements. For example, [] is the empty multiset, i.e. the function with empty graph. As another example $[x_1, x_1, x_2] = [x_1, x_2, x_1] = [x_2, x_1, x_1]$ is the multiset with two occurrences of x_1 and one occurrence of x_2, i.e. the function $m : \{x_1, x_2\} \to \mathbb{N}^+, m(x_1) = 2, m(x_2) = 1$. There are several operations defined on multisets $m_1, m_2 \in [X]$, defined as in [10].

The denotational semantics given in this paper is built within the mathematical framework of *1-bounded complete metric spaces*. We work with the following notions which we assume known: *metric* and *ultrametric* space, *isometry* (distance preserving bijection between metric spaces, denoted by '$\cong$'), *complete* metric space, and *compact* set. For details the reader may consult [4]. We recall that if (X, d_X), (Y, d_Y) are metric spaces, a function $f{:}X \to Y$ is a *contraction* if $\exists c \in \mathbb{R},\ 0 \leq c < 1,\ \forall x_1, x_2 \in X : d_Y(f(x_1), f(x_2)) \leq c \cdot d_X(x_1, x_2)$. In metric semantics it is customary to attach a contracting factor of $c = \frac{1}{2}$ to each compu-

tation step. When $c = 1$ the function f is called *non-expansive*. In what follows we denote the set of all nonexpansive functions from X to Y by $X \stackrel{1}{\to} Y$. The following theorem is at the core of metric semantics.

Theorem 1 (Banach). *Let (X, d_X) be a complete metric space. Each contraction $f : X \to X$ has a* unique *fixed point.*

If $(x, y \in)X$ is any nonempty set, one can define the *discrete metric* on X ($d : X \times X \to [0, 1]$) as follows: $d(x, y) = 0$ if $x = y$, and $d(x, y) = 1$ otherwise. (X, d) is a complete ultrametric space.

Let $(x \in)X$ be a countable set. We denote $[X] \stackrel{not.}{=} \bigcup_{A \in \mathcal{P}_{finite}(X)} \{m \mid m \in (A \to \mathbb{N}^+)\}$, where $\mathcal{P}_{finite}(X)$ is the powerset of all *finite* subsets of X. $[X]$ is the set of finite multisets of elements of type X. Since X is countable, $\mathcal{P}_{finite}(X)$ is also countable. We use the abbreviation $\mathcal{P}_{nco}(\cdot)$ to denote the powerset of *nonempty and compact* subsets of '$\cdot$'. We also assume it is known how to construct composed spaces ($\frac{1}{2}$-identity, product, disjoint union, function space, all nonempty and compact subsets) for given metric spaces [4]. We often suppress the metrics in domain definitions, and write $\frac{1}{2} \cdot X$ instead of $(X, d_{\frac{1}{2} \cdot X})$.

3 Syntax and Semantics of $\mathcal{L}_{MC}$

We assume that O is a countable set. Let $(w \in)W = [O]$ be the set of all finite multisets of O objects, and $(M \in)Mname$ be a set of *membrane names*.

Syntax of $\mathcal{L}_{MC}$ is given by

(a) (Statements) $x(\in X) ::= o \mid \mathsf{in}(o) \mid \mathsf{out}(o) \mid \mathsf{new}(M, y) \mid x \| x$
where $y(\in Y) ::= o \mid y \parallel y$
(b) (Rules) $r(\in R) ::= r_\epsilon \mid w \Rightarrow x; r$
(c) (Membrane declarations)
$d(\in MD) ::= \mathsf{membrane}\ M\,\{r\}$
$D(\in MDs) ::= d \mid d; D$
(d) (Programs) $\rho(\in \mathcal{L}_{MC}) ::= D; x$.

An $\mathcal{L}_{MC}$ statement may be either an object o, a communication statement $\mathsf{in}(o)$ or $\mathsf{out}(o)$, a membrane creation statement of the form $\mathsf{new}(M, y)$, where M is a membrane name and y is a statement of the type Y, or a parallel composition of two $\mathcal{L}_{MC}$ statements of the form $x_1 \parallel x_2$. Note that, a statement $y \in Y$ may be either an object, or a parallel composition of two or more objects, but y cannot contain communication or membrane creation primitives. Obviously, $Y \subseteq X$.

A statement $\mathsf{out}(o)$ indicates that the object o must leave the membrane where it is currently located, and becomes an element of the surrounding region. A statement $\mathsf{in}(o)$ indicates that the object o must enter one of the child membranes, nondeterministically chosen.

The execution of a membrane created by a statement $\mathsf{new}(M, y)$ will always begin by a multiset rewriting step. In the first step after its creation, a membrane cannot perform any communication or membrane creation operations. However,

by using the rewriting rules specified in the definition of M, the newly created membrane (instance) can next immediately create new inner membranes and communication objects and it can proceed as any other membrane.

An $\mathcal{L}_{MC}$ program $D; x$ consists of a list $D \in MDs$ of membrane declarations and a statement $x \in X$. A membrane declaration introduces a type of membranes, which can be instantiated. In $\mathcal{L}_{MC}$ we speak of *membrane types*, and *membrane instances*. Each membrane instance has a (unique) label. The execution of the program $D; x$ starts with the creation of an instance of the first membrane type in the declarations list D, which becomes a *skin* membrane; the skin membrane starts the execution of the program by executing statement x.

A membrane declaration $\mathsf{membrane}\ M\,\{r\}$ indicates the name $M \in Mname$ of the membrane (type) and a (possibly empty) list of rules $r \in R$, which specify the behaviour of objects inside any instance of a membrane of the type M. A rule $w \Rightarrow x$ is composed of two elements: a multiset $w \in W$ and a statement $x \in X$. An $\mathcal{L}_{MC}$ statement is a concurrent composition of objects, which behave as a multiset. In this interpretation $w \Rightarrow x$ is a multiset rewriting rule, specifying that w is rewritten as x. Intuitively, a rule $w \Rightarrow x$ is like a 'procedure' definition, with 'name' w and 'body' x. The objects $o_1, \cdots, o_n$ in the multiset $w = [o_1 \cdots o_n]$ are 'fragments' of the procedure name. Only when all the 'fragments' of such a 'procedure name' are prepared for interaction a rewriting rule is applied, and so replaces the 'name of the procedure' with its 'body'. The 'body' (the right hand side) of a rule is a statement. Further explanations concerning the syntactic constructions for specifying membranes and rules in $\mathcal{L}_{MC}$ are provided in [10].

If we consider the analogy with object-oriented programming, we can state clearly that in $\mathcal{L}_{MC}$ an object $o \in O$ is *not* an instance of a membrane. In $\mathcal{L}_{MC}$, an object is just an elementary statement, a symbol taken from the alphabet O. The reader may wonder why we use the semantic notion of a *multiset* in the syntax definition of $\mathcal{L}_{MC}$. According to the syntax, we can use rules of the form $j \Rightarrow x$, where $j ::= o \mid j \,\&\, j$ is the set of 'method names'. We use multisets as 'method names' because the order in which fragments occur in such a 'method name' is irrelevant.

Semantics of $\mathcal{L}_{MC}$ is given by using a denotational approach. Denotational semantics (known also as mathematical semantics) is an important step in formalizing the meanings of languages/systems. The most important principle in denotational semantics is *compositionality*: the meaning of a compound construction is determined solely on the basis of the meanings of its components. In general, denotational semantics assigns to every construction of a language a certain formal *meaning*, which is an element from a suitably chosen mathematical model. Following [4], we choose to use the mathematical framework of *complete metric spaces* for our semantic description. In this approach, one can prove the semantic properties by making use of Banach's theorem which states that contracting functions on complete metric spaces have *unique* fixed points.

We introduce the metric domains to express the behaviour of $\mathcal{L}_{MC}$ programs, and then give a continuation-based denotational semantics for $\mathcal{L}_{MC}$. The final yield of the denotational semantics is an element of a linear time domain [4].

We assume given a (countably) infinite set $(l \in)L$ of *membrane labels*. Let also $(\varsigma \in)\mathcal{P}_{finite}(L)$ be the set of all finite subsets of L. We also assume given a function $\nu : \mathcal{P}_{finite}(L) \to L$, such that $\nu(\varsigma) \notin \varsigma$, for any $\varsigma \in \mathcal{P}_{finite}(L)$. We obtain a possible example of such a set L and function ν by putting $L = \mathbb{N}$, and $\nu(\varsigma) = 1 + max\{n \mid n \in \varsigma\}$, with the convention that $\nu(\emptyset) = 0$.

Following [3], we define the set $(\mu \in)Mb$ of *membranes* inductively:

- If $M \in Mname$ is a membrane name, $l \in L$ is a label and $w \in W = [O]$ is a multiset of O objects then $\langle M, l \mid w; \rangle \in Mb$; $\langle M, l \mid w; \rangle$ is called a (*simple* or) *elementary membrane*;
- If $M \in Mname$ is a membrane name, $l \in L$ is a label, $w \in W$ is a multiset of O objects, and $\mu_1, \ldots, \mu_n \in Mb$ then $\langle M, l \mid w; \mu_1, \ldots, \mu_n \rangle \in Mb$; $\langle M, l \mid w; \mu_1, \ldots, \mu_n \rangle$ is called a *composite membrane*.

We provide a continuation-based denotational semantics for $\mathcal{L}_{MC}$ in which we use the linear time domain $(p \in)\mathbf{P} = \mathcal{P}_{nco}(\mathbf{Q})$, where $(q \in)\mathbf{Q} \cong \{\epsilon\} + (Mb \times \frac{1}{2} \cdot \mathbf{Q})$. In this domain equation, the set Mb is endowed with the discrete metric (i.e. an ultrametric). According to [4], the above domain equation has a *unique* solution (up to isomorphism). The solution is a complete ultrametric space.

An element of the domain $\mathbf{P}$ is a non-empty and compact collection of $\mathbf{Q}$ sequences. $\mathbf{Q}$ is a domain of finite and infinite sequences over Mb; ϵ is the empty sequence. Instead of $(\mu_1, (\mu_2, \ldots (\mu_n, \epsilon) \ldots))$ and $(\mu_1, (\mu_2, \ldots))$, we write $\mu_1\mu_2 \ldots \mu_n$ and $\mu_1\mu_2 \ldots$, respectively. In particular, instead of (μ, ϵ) (a sequence of length 1) we write just μ. Nondeterministic behaviour in $\mathcal{L}_{MC}$ is defined by using the operator $+ : (\mathbf{P} \times \mathbf{P}) \to \mathbf{P}$ defined by $p_1 + p_2 = \{q \mid q \in p_1 \cup p_2, q \neq \epsilon\} \cup \{\epsilon \mid \epsilon \in p_1 \cap p_2\}$. It is easy to see that $+$ is well-defined, non-expansive, associative and commutative [4]. Also, we use the following notations: $\mu \cdot q = (\mu, q)$ and $\mu \cdot p = \{\mu \cdot q \mid q \in p\}$, for any $\mu \in Mb, q \in \mathbf{Q}, p \in \mathbf{P}$. Based on the results presented in [4], we have $d(\mu \cdot p_1, \mu \cdot p_2) = \frac{1}{2} \cdot d(p_1, p_2)$.

Let $(\alpha \in)A = Mb \to \mathcal{P}_{finite}(Mb)$. We define a set $(\theta, \vartheta \in)\Theta$ of *actions*:

- $\theta_0 \in \Theta$ (θ_0 is a distinct element of Θ);
- If $\alpha \in A$ then $\alpha \in \Theta$, for any $\alpha \in A$;
- If $\alpha \in A$ and $\theta \in \Theta$, then $(\,\cdot\,, \alpha, \theta) \in \Theta$; we use a triple $(\,\cdot\,, \alpha, \theta)$ to represent a sequential composition between α and θ, but write $\alpha \cdot \theta$ instead of $(\,\cdot\,, \alpha, \theta)$;
- If $\theta_1, \theta_2 \in \Theta$, then $(\,\|\,, \theta_1, \theta_2) \in \Theta$; instead of $(\,\|\,, \theta_1, \theta_2)$, we use $\theta_1 \parallel \theta_2$ for parallel composition between θ_1 and θ_2.

We define a valuation $\mathcal{T}[\![\cdot]\!] : \Theta \to A$ by

$$\begin{aligned}
\mathcal{T}[\![\theta_0]\!] &= \lambda\mu \,.\, \{\mu\}, \\
\mathcal{T}[\![\alpha]\!] &= \mathcal{T}[\![\alpha \cdot \theta_0]\!] = \alpha, \\
\mathcal{T}[\![\alpha \cdot \theta]\!] &= \lambda\mu \,.\, \bigcup\{\mathcal{T}[\![\theta]\!](\mu') \mid \mu' \in \alpha(\mu)\} \quad \text{if } \theta \neq \theta_0, \\
\mathcal{T}[\![\theta \parallel \theta_0]\!] &= \mathcal{T}[\![\theta_0 \parallel \theta]\!] = \mathcal{T}[\![\theta]\!], \\
\mathcal{T}[\![\theta_1 \parallel \theta_2]\!] &= \mathcal{T}[\![\theta_1]\!] \mathbin{\hat{\|}} \mathcal{T}[\![\theta_2]\!],
\end{aligned}$$

where the operator $\hat{\|} : (A \times A) \to A$ is given by
$\alpha_1 \mathbin{\hat{\|}} \alpha_2 = \lambda\mu \,.\, (\{\mu_2 \mid \mu_1 \in \alpha_1(\mu), \mu_2 \in \alpha_2(\mu_1)\} \cup \{\mu_1 \mid \mu_2 \in \alpha_2(\mu), \mu_1 \in \alpha_1(\mu_2)\})$.

Clearly, $\hat{\|}$ is well-defined, i.e. $\theta_1 \,\hat{\|}\, \theta_2 \in A$ for any $\theta_1, \theta_2 \in A$. Well-definedness of $\mathcal{T}[\![\theta]\!]$ follows by an easy induction on the structure of θ.

In the following definitions, Θ is just a set endowed with the discrete metric. We define the domain $\mathbf{D}$ of *computations*, and the domain $\mathbf{K}$ of *continuations* by $(\varphi \in)\mathbf{D} = \mathbf{K} \xrightarrow{1} F$ and $(\kappa \in)\mathbf{K} = (\Theta \times \Theta) \to F$, where $F = \mathcal{P}_{finite}(L) \to Mb \to \mathbf{P}$. In the domain definitions, the sets Θ, $\mathcal{P}_{finite}(L)$ and Mb are endowed with discrete metrics, and $\mathbf{D}$ is a domain of *nonexpansive* functions [4].

Definition 1. *(Semantics of parallel composition)*

(a) We define $\|\colon (\mathbf{D} \times \mathbf{D}) \xrightarrow{1} \mathbf{D}$ *by*

$$\varphi_1 \parallel \varphi_2 = \lambda\kappa\,.\,\varphi_1(\lambda(\theta_1, \vartheta_1)\,.\,\varphi_2(\lambda(\theta_2, \vartheta_2)\,.\,\kappa(\theta_1 \,\hat{\|}\, \theta_2, \vartheta_1 \,\hat{\|}\, \vartheta_2)));$$

(b) For any $n \in \mathbb{N}$ *we define* $\|^n\,(\cdot) : \mathbf{D}^n \to \mathbf{D}$ *(*$\mathbf{D}^n = \mathbf{D} \times \cdots \times \mathbf{D}$ *n times,* $n \geq 1$*) by* $\|^1\,(\varphi) = \varphi$*, and* $\|^{n+1}\,(\varphi_1, \ldots, \varphi_{n+1}) = \varphi_1 \parallel (\|^n\,(\varphi_2, \ldots, \varphi_{n+1}))$*. For readability, we write* $\varphi_1 \parallel \cdots \parallel \varphi_n$ *instead of* $\|^n\,(\varphi_1, \ldots, \varphi_n)$*.*

Proposition 1. *The operator for parallel composition* $\|$ *is well-defined and nonexpansive in both arguments.*

The above property is not difficult to prove. Other properties, e.g., the associativity or the commutativity of $\|$, are more difficult to establish. A formal proof of the fact that $\|$ is associative employs the technique introduced in [9].

In order to define the semantics of $\mathcal{L}_{MC}$ statements we need some auxiliary operators on membranes. Given a membrane μ, $add(o, l', \mu)$ adds the object o to the multiset stored in the membrane region indicated by label l'. $new_M(M_{new}, l_{new}, l', \mu)$ creates a new membrane region with label l_{new} of the type M_{new} as an inner membrane (a child) of the membrane region with label l'. The operator $out(o, l', \mu)$ sends the object o from its membrane (instance) to the surrounding region. The operator $in(o, l', \mu)$ sends the object o from its membrane into a child membrane, nondeterministically chosen. The operator in is used to specify the fact that an object enters into a child membrane which is nondeterministically selected. If there is no child membrane, in is inoperative.

The operators are defined by induction on the structure of the membrane hierarchy, considering the fact that membranes (regions) are labelled in a one-to-one manner with labels from the given set L. All definitions are recursive. We only provide the definitions for some representative (non-recursive) cases, leaving the other cases to the reader. $\uplus$ is the multiset sum operation [1].

$$add : (O \times L \times Mb) \to Mb,$$
$$add(o, l', \langle M, l \mid w; \rangle) = \begin{cases} \langle M, l \mid [o] \uplus w; \rangle & \text{if } l' = l \\ \langle M, l \mid w; \rangle & \text{if } l' \neq l \end{cases}$$

$$new_M : (Mname \times L \times L \times Mb) \to Mb,$$
$$new_M(M_{new}, l_{new}, l', \langle M, l \mid w; \rangle) = \begin{cases} \langle M, l \mid w; \langle M_{new}, l_{new} \mid []; \rangle \rangle & \text{if } l' = l \\ \langle M, l \mid w; \rangle & \text{if } l' \neq l \end{cases}$$

$$
\begin{array}{l}
in : (O \times L \times Mb) \to \mathcal{P}_{finite}(Mb),\\
in(o, l, \langle M, l \mid w; \rangle) = \{\langle M, l \mid [o] \uplus w; \rangle\},\\
in(o, l, \langle M, l \mid w; \mu_1, \ldots, \langle M_i, l_i \mid w_i; \mu_{i1}, \ldots, \mu_{in_i}\rangle, \ldots, \mu_n\rangle) =\\
\quad \{\langle M, l \mid w; \mu_1, \ldots, \langle M_i, l_i \mid [o] \uplus w_i; \mu_{i1}, \ldots, \mu_{in_i}\rangle, \ldots, \mu_n\rangle \mid 1 \le i \le n\},\\
\\
out : (O \times L \times Mb) \to Mb,\\
out(o, l', \langle M, l \mid w; \mu_1, \ldots, \mu_n\rangle) = \langle M, l \mid [o] \uplus w; \mu_1, \ldots, \mu_n\rangle,\\
\quad \mathsf{if}\ l' \in \{label(\mu_1), \ldots, label(\mu_n)\}\ .
\end{array}
$$

Definition 2. *Denotational semantics* $[\![\cdot]\!] : X \to L \to \mathbf{D}$ *of* $\mathcal{L}_{MC}$ *statements is defined by*

$$
\begin{array}{rl}
[\![o]\!](l) &= \lambda\kappa\,.\,\kappa(\lambda\mu\,.\,\{\,add(o, l, \mu)\}, \theta_0),\\
[\![\,\mathsf{in}(o)]\!](l) &= \lambda\kappa\,.\,\kappa(\lambda\mu\,.\,in(o, l, \mu), \theta_0),\\
[\![\,\mathsf{out}(o)]\!](l) &= \lambda\kappa\,.\,\kappa(\lambda\mu\,.\,\{\,out(o, l, \mu)\}, \theta_0),\\
[\![\,\mathsf{new}(M, y)]\!](l) &= \lambda\kappa\,.\,\lambda\varsigma\,.\ \mathsf{let}\ l' = \nu(\varsigma); \varsigma' = \{l'\} \cup \varsigma\ \mathsf{in}\\
&\qquad [\![y]\!](l')(\lambda(\theta, \vartheta)\,.\,\kappa(\theta_0, (\lambda\mu\,.\,\{\,new_M(M, l', l, \mu)\}) \cdot (\theta \parallel \vartheta)))(\varsigma'),\\
[\![x_1 \parallel x_2]\!](l) &= ([\![x_1]\!](l)) \parallel ([\![x_2]\!](l)).
\end{array}
$$

Proposition 2. *The denotational mapping* $[\![\cdot]\!]$ *is well-defined.*
In particular, $[\![x]\!](l)$ *is non-expansive for any* $x \in X$, $l \in L$.

The denotational mapping $[\![x]\!](l)$ takes as parameters a statement $x \in X$ and a label $l \in L$. The parameter l is the membrane label where the statement x is executed. Each continuation takes as parameter a pair of actions (θ, ϑ). The action θ represents the behaviour of object and communication actions. The action ϑ represents the behaviour of membrane creation actions. Within each computation step the two actions θ and ϑ are executed in this sequence by the initial continuation κ_0: first the action θ, then the action ϑ. However, note that only one observable is produced by each computation step.

All equations are straightforward, except the equation handling membrane creation. In the equation defining the semantics of $\mathsf{new}(M, y)$, a new label l' is created. Next, the body y of the statement $\mathsf{new}(M, y)$ is executed. The membrane creation action $\lambda\mu\,.\,\{\,new_M(M, l', l, \mu)\})$ is executed before $\theta \parallel \vartheta$. Thus, the actions produced by $[\![y]\!]$ are executed in the newly created membrane with label l' only *after* the new membrane is created.

The mapping $appRules(r, w')$ computes a (finite) set of pairs, each pair consisting of a multiset of rules applicable to w' and an irreducible submultiset of w'. $\subseteq$ and $\setminus$ are operations for submultiset testing and multiset difference [1].

$$
\begin{array}{l}
appRules : (R \times W) \to \mathcal{P}_{finite}(R \times W),\\
appRules(r, w) = \ \mathsf{if}\ aux(r, w) = \emptyset\ \ \mathsf{then}\ \{(r_\epsilon, w)\}\ \ \mathsf{else}\\
\quad \{(\overline{w} \Rightarrow \overline{x} \,\square\, r', w'') \mid ((\overline{w}, \overline{x}), w') \in aux(r, w),\ (r', w'') \in\ appRules(r, w')\},\\
\text{where } aux : (R \times W) \to \mathcal{P}_{finite}((W \times X) \times W),\\
aux(r_\epsilon, w) = \emptyset,\\
aux(w' \Rightarrow x' \,\square\, r, w) =\\
\quad \mathsf{if}\ (w' \subseteq w)\ \ \mathsf{then}\ \{((w', x'), w \setminus w')\} \cup aux(r, w)\ \ \mathsf{else}\ aux(r, w).
\end{array}
$$

We define a scheduler mapping $sched : (Mb \times MDs) \to \mathcal{P}_{finite}(\mathbf{D} \times Mb)$ by induction on the structure of the membrane. The function $sched(\mu, D)$ takes as arguments a membrane μ and a list of membrane declarations D, and yields a finite set of pairs, each pair consisting of a computation and a corresponding membrane. It uses *appRules* to compute the applicable rules for each membrane region. We use *sched* in a fixed point construction required to define the semantics of parallel rewriting of multisets in a compositional manner.

$$
\begin{array}{l}
sched(\langle M, l \mid w; \rangle, D) = \\
\quad \{([\![x_1]\!](l) \parallel \cdots \parallel [\![x_n]\!](l) \parallel [\![o_1]\!](l) \parallel \cdots \parallel [\![o_m]\!](l), \langle M, l \mid []; \rangle) \\
\quad\ \mid (r', w') \in appRules(rules(D, M), w), \\
\quad\quad r' = w_1 \Rightarrow x_1; \ldots; w_n \Rightarrow x_n; r_\epsilon, w' = [o_1, \ldots, o_m]\}, \\
sched(\langle M, l \mid w; \mu_1, \ldots, \mu_k \rangle, D) = \\
\quad \{([\![x_1]\!](l) \parallel \cdots \parallel [\![x_n]\!](l) \parallel [\![o_1]\!](l) \parallel \cdots \parallel [\![o_m]\!](l) \parallel \\
\quad\quad \varphi_1 \parallel \cdots \parallel \varphi_k, \langle M, l \mid []; \mu'_1, \ldots, \mu'_k \rangle) \\
\quad\ \mid (r', w') \in appRules(rules(D, M), w), \\
\quad\quad r' = w_1 \Rightarrow x_1; \ldots; w_n \Rightarrow x_n; r_\epsilon, w' = [o_1, \ldots, o_m], \\
\quad\quad (\varphi_1, \mu'_1) \in sched(\mu_1, D), \ldots, (\varphi_k, \mu'_k) \in sched(\mu_k, D)\}.
\end{array}
$$

Given a list of membrane declarations, we also define a mapping $haltMb : (Mb \times MDs) \to Bool$ which decides whether the membrane system has reached a halting configuration. *haltMb* is defined with the aid of an auxiliary mapping $haltM : (Mname \times MDs \times W) \to Bool$, based on the mapping *appRules*.

$$
\begin{array}{l}
haltMb(\langle M, l \mid w; \rangle, D) = haltM(M, D, w), \\
haltMb(\langle M, l \mid w; \mu_1, \ldots, \mu_n \rangle, D) = \\
\quad haltM(M, D, w) \wedge haltMb(\mu_1, D) \wedge \cdots \wedge haltMb(\mu_n, D), \\
haltM(M, D, w) = (appRules(rules(D, M), w) = \{(r_\epsilon, w)\}).
\end{array}
$$

We define a mapping $\Psi : MDs \to \mathbf{K} \to \mathbf{K}$ by

$$
\begin{array}{l}
\Psi(D)(k)(\theta, \vartheta)(\varsigma)(\mu) = \\
\quad +\{\mu_2 \cdot (\ \mathsf{if}\ (haltMb(\mu, D))\ \mathsf{then}\ \{\epsilon\}\ \mathsf{else}\ \varphi(\kappa)(\varsigma)(\mu'_2)) \\
\quad\quad \mid \mu_1 \in \mathcal{T}[\![\theta]\!](\mu), \mu_2 \in \mathcal{T}[\![\vartheta]\!](\mu_1), (\varphi, \mu'_2) \in sched(\mu_2, D)\}.
\end{array}
$$

For any $D \in MDs$, we define the *initial continuation* $\kappa_0 \in \mathbf{K}$ by $\kappa_0 = fix(\Psi(D))$. This definition is justified by Proposition 3 which states that $\Psi(D)$ is $\frac{1}{2}$ contractive for any $D \in MDs$. According to Banach's Theorem, $\Psi(D)$ has a *unique* fixed point for any $D \in MDs$.

Proposition 3. *$\Psi(D) \in \mathbf{K} \xrightarrow{\frac{1}{2}} \mathbf{K}$ for any $D \in MDs$, i.e.*

$$d(\Psi(D)(\kappa_1)(\theta, \vartheta)(\varsigma)(\mu), \Psi(D)(\kappa_2)(\theta, \vartheta)(\varsigma)(\mu)) \leq \tfrac{1}{2} \cdot d(\kappa_1, \kappa_2)$$

for any $D \in MDs$, $\kappa_1, \kappa_2 \in \mathbf{K}$, $(\theta, \vartheta) \in \Theta \times \Theta$, $\varsigma \in \mathcal{P}_{finite}(A)$ and $\mu \in Mb$.

Proof. If $haltMb(\mu, D) = true$, then

$$d(\Psi(D)(\kappa_1)(\theta, \vartheta)(\varsigma)(\mu), \Psi(D)(\kappa_2)(\theta, \vartheta)(\varsigma)(\mu)) = 0 \leq \tfrac{1}{2} \cdot d(\kappa_1, \kappa_2).$$

Otherwise, if $haltMb(\mu, D) = false$, we have

$$\begin{aligned}
&d(\Psi(D)(\kappa_1)(\theta,\vartheta)(\varsigma)(\mu), \Psi(D)(\kappa_2)(\theta,\vartheta)(\varsigma)(\mu)) && [+ \text{ is nonexpansive}]\\
&\leq \tfrac{1}{2} \cdot max\{d(\varphi(\kappa_1)(\varsigma)(\mu_2'), \varphi(\kappa_2)(\varsigma)(\mu_2'))\\
&\qquad \mid \mu_1 \in \mathcal{T}[\![\theta]\!](\mu), \mu_2 \in \mathcal{T}[\![\vartheta]\!](\mu_1), (\varphi, \mu_2') \in sched(\mu_2, D)\}\\
&\qquad\qquad [\text{since } \varphi \in \mathbf{D}, \varphi \text{ is nonexpansive}]\\
&\leq \tfrac{1}{2} \cdot d(\kappa_1, \kappa_2).
\end{aligned}$$

Definition 3. *Semantics of $\mathcal{L}_{MC}$ programs is defined by $\mathcal{D}[\![\cdot]\!] : \mathcal{L}_{MC} \to \mathbf{P}$,*

$$\mathcal{D}[\![D; x]\!] = [\![x]\!](l_0)(\kappa_0)(\varsigma_0)(\mu_0),$$

where $D =$ membrane $M_0\,\{r_0\};\ \cdots\ ;$ membrane $M_m\,\{r_m\}$, $\kappa_0 = fix(\Psi(D))$, $l_0 = \nu(\emptyset)$, $\varsigma_0 = \{l_0\}$ and $\mu_0 = \langle M_0, l_0 \mid [\,]; \rangle$.

The function $\mathcal{D}[\![\rho]\!]$ defines the semantics of an $\mathcal{L}_{MC}$ program $\rho = D; x$, where $D \in MDs$ and $x \in X$. The execution of an $\mathcal{L}_{MC}$ program begins with the creation of the skin which is an instance of the first membrane type in the declaration list D. The label of the skin is $\nu(\emptyset)$; the statement x is executed in the skin membrane. In the spirit of denotational semantics, the initial continuation $\kappa_0 \in \mathbf{K}$ is defined as a fixed point of the higher-order mapping Ψ.

Example 1. We consider the $\mathcal{L}_{MC}$ program $\rho = D; x$, where D is:

membrane M_0 {
  $[o_1, o_4] \Rightarrow o_2 \parallel o_4$;
  $[o_2] \Rightarrow \mathsf{in}(o_5) \parallel \mathsf{new}(M_1, o_1 \parallel o_5)$;
  $[o_2] \Rightarrow o_4$;
  $[o_5] \Rightarrow \mathsf{in}(o_4)$;
  $[o_3] \Rightarrow \mathsf{in}(o_5)$ };
membrane M_1 {
  $[o_1] \Rightarrow o_2 \parallel \mathsf{out}(o_3)$;
  $[o_2] \Rightarrow o_3$ }

and $x = o_1 \parallel o_4$. Let $\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6 \in Mb$

$$\begin{aligned}
\mu_1 &= \langle M_0, l_0 \mid [o_1, o_4]; \rangle\\
\mu_2 &= \langle M_0, l_0 \mid [o_2, o_4]; \rangle\\
\mu_3 &= \langle M_0, l_0 \mid [o_4, o_4]; \rangle\\
\mu_4 &= \langle M_0, l_0 \mid [o_4, o_5]; \langle M_1, l_1 \mid [o_1, o_5]; \rangle \rangle\\
\mu_5 &= \langle M_0, l_0 \mid [o_3, o_4]; \langle M_1, l_1 \mid [o_2, o_4, o_5]; \rangle \rangle\\
\mu_6 &= \langle M_0, l_0 \mid [o_4]; \langle M_1, l_1 \mid [o_3, o_4, o_5, o_5]; \rangle \rangle
\end{aligned}$$

One can check that $\mathcal{D}[\![\rho]\!] = \{\mu_1\mu_2\mu_3, \mu_1\mu_2\mu_4\mu_5\mu_6\}$.

The denotational specification presented in this paper was developed following a prototyping approach; we used the functional language Haskell as a prototyping tool (and metalanguage) for this denotational semantics. A couple of $\mathcal{L}_{MC}$ programs (including the $\mathcal{L}_{MC}$ example discussed above) are provided, and can be tested by using the semantic interpreter available at

`http://users.utcluj.ro/~eneia/CMC17-semMC.hs` .

The semantic interpreter is a complete Haskell implementation of the denotational semantics presented in this paper.

4 Conclusion

In this paper we introduce a language $\mathcal{L}_{MC}$ in which computations are specified by means of multiset rewriting rules applied in a maximal parallel way into membrane-delimited compartments. In $\mathcal{L}_{MC}$ there is a notion of membrane declaration, and membranes are grouped into classes based on their rewriting rules. There exist primitives for parallel communication of objects between adjacent membranes, and a primitive for membrane creation (instantiation).

We present a denotational semantics designed with complete metric spaces for $\mathcal{L}_{MC}$. For illustration, we presented a small $\mathcal{L}_{MC}$ example program; other examples are available on our Haskell implementation webpage.

References

1. A. Alexandru, G. Ciobanu. Mathematics of multisets in the Fraenkel-Mostowski framework, *Bull. Math. Soc. Sci. Math. Roumanie*, Tome 58(106), pp. 318, 2015
2. A.W. Appel. *Compiling with Continuations*, Cambridge University Press, 2007.
3. O. Andrei, G. Ciobanu, D. Lucanu. A Rewriting Logic Framework for Operational Semantics of Membrane Systems, *Theoretical Computer Science*, vol. 373, pp. 163–181, 2007.
4. J.W. de Bakker, E.P. de Vink. *Control Flow Semantics*, MIT Press, 1996.
5. G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez. *Applications of Membrane Computing*, Natural Computing Series, Springer, 2006.
6. G. Ciobanu. Semantics of P Systems, *Handbook of Membrane Computing*, Oxford University Press, pp. 413-436, 2009.
7. G. Ciobanu, E.N. Todoran. Metric Denotational Semantics for Parallel Rewriting of Multisets, *Proceedings 13th Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2011)*, pp. 276–283, IEEE Computer Press, 2011.
8. G. Ciobanu, E.N. Todoran. Relating Two Metric Semantics for Parallel Rewriting of Multisets, *Proceedings 14th Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2012)*, pp.273–280, IEEE Computer Press, 2012.
9. G. Ciobanu, E.N. Todoran. Continuation Semantics for Asynchronous Concurrency, *Fundamenta Informaticae*, vol. 131(3-4), pp. 373–388, 2014.
10. G. Ciobanu, E.N. Todoran. Continuation semantics for dynamic hierarchical systems", *Proceedings of 17th Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2015)*, pp. 281-288, IEEE Computer Press, 2015.
11. Gh. Păun, *Membrane Computing. An Introduction.* Springer, 2002.
12. E.N. Todoran. Metric semantics for synchronous and asynchronous communication: a continuation-based approach, *Electronic Notes in Theoretical Computer Science*, vol.28, pp. 119–146, Elsevier, 2000.

Secure dispersion of robots in a swarm using P colonies

Andrei George Florea and Cătălin Buiu
{andrei.florea,catalin.buiu}@acse.pub.ro

Department of Automatic Control and Systems Engineering
Politehnica University of Bucharest

Abstract. Robotics has emerged as a challenging and fruitful application area for membrane computing. Since the introduction of membrane controllers based on numerical P systems for single robots, more models based on P colony automata and P swarms, an extension of P colonies, have been proposed. More, the focus has extended from the control of a single robot to the control of multiple, simple robots in a swarm. An overlooked issue in swarm robotics is the security of the swarm. While there are multiple potential security hazards in a robotic swarm, there have been no approaches to address both the control and security of a swarm. In this paper we report the development of an P colonies-based algorithm for secure dispersion of robots in a swarm. The overall concept and the software is tested on a swarm of simulated robots.

Keywords: membrane computing; P colonies; P swarms; swarm robotics; security; Kilobot.

1 Introduction

A swarm robotic system (SRS) is composed of many simple robots (agents) which exhibit a group (or swarm) intelligence that emerge as a result of local interactions between robots and interactions of the robots with the environment. In a SRS there is no centralized controller and the global behavior is the result of simple individual rules. Scalability, flexibility, and robustness are key features of SRSs [10]. A recent review of swarm robotics identifies a number of tasks relevant to SRSs: aggregation (gathering a number of agents in a well defined place), flocking (large groups of agents should move together to a well defined target location), foraging (finding and exploiting food sources), object clustering and sorting (finding and grouping objects together), navigation (reaching a target with the help of other agents), path formation (building a path between two locations in the environment), collaborative manipulation, dynamic task allocation, and deployment [1]. Dispersion is a common deployment algorithm in which the agents should position themselves away from each other. Stigmergic communication through the environment is often used in deployment algorithms [11]. Security challenges for SRSs have been identified in [5], such as:

resource constraints, physical capture, lack of centralized control, identity and authorization, key management, and intrusion detection.

Membrane computing models are parallel and distributed systems and have a number of similarities with SRSs. The P colonies have the same foundation ideas as swarm robotics: to use simple agents, placed in a shared environment, leading to emergent behaviors [7]. The basic rules in a P colony are the evolution rules of the form $a \rightarrow b$ (rewriting object a into object b) and the communication rules of the form $c \leftrightarrow d$ (exchanging the object c within the agent with the object d within the internal environment of the P colony). An XP colony is a P colony with exteroceptive rules of the form $c \Leftrightarrow d$ which checks the presence of an object d in the global environment (outside the internal environment of the P colony). If this is present, the object c will be exchanged with object d from the global environment. A P swarm is a colony of XP colonies, and this concept was introduced in [2] and extended in [4].

This paper proposes the use of P colonies for implementing a secure dispersion algorithm for a SRS. While dispersing the robots, the P colony model is also programmed for detection of non-self robots (robots that are not part of the original swarm) and consequently only the self robots (the original robots of the swarm) will be dispersed while ignoring the intruders.

This paper is organized as follows. The next section gives the control model in which the basic modules of a robot are abstracted as agents in a P colony. All of the P colonies abstracting the robots are interacting using messages exchanged by the robots and the sensed distance between neighbors in the swarm. The idea is validated on securely dispersing a swarm of simulated Kilobot robots (dispersion and intrusion detection simultaneously). The experiments are based on using Lulu, a Python simulator for P colonies and P swarms [4] that was ported to the C language, and Kilombo, a Kilobot robot simulator written in C which allows the fast simulation of swarms of up to 1000 robots [6]. Demonstration movies that support our ideas are available at [3]. The paper ends with conclusions and directions for further improvements.

2 Robot control model

The P colony based robot control model is presented in Figure 1. This model is an improved version of the original P colony controller described in [4] where only the command and motion agents were defined thus restraining the robot control model to a feed-forward type. The various input and output devices of the robot are abstracted as P colony agents that can interact through the P colony environment.

The model abstracts the sensors and effectors available on the Kilobot robot [9] that was used for testing the model. The only component that was not included is the light sensor.

From a generic perspective, the controller input is represented by the distance and symbolic id of any given neighbor robot and a new RGB led color and/or motion direction represent the controller output. Each agent has a specific set

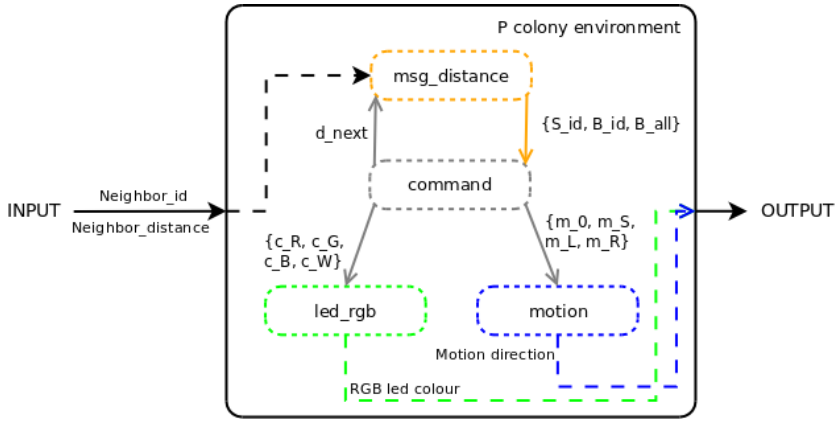


Fig. 1. The structure of the P colony based robot controller.

of objects that can be processed. These objects can be either request (input) or response (output) objects and form the basis of the communication mechanism used within the controller. The dashed lines that connect the Input and Output components of the model to the corresponding agents signify that background processing is needed in order to convert real values to symbolic objects.

The message_distance agent is used to model the infrared distance estimation function that can be executed after the successful reception of a message from a neighbor robot. Because of the fact that multiple messages can be received in a short amount of time, all messages are buffered internally by the controller application and, upon request (*d_next* object), the agent can respond with one of three objects (*S_id*, *B_id*, *B_all*). These values correspond to the comparison of the current distance from robot that has the unique symbolic number id with a threshold value and have the meaning of distance *Small* and distance *Big* respectively. If there is no neighbor in the infrared communication range of the robot or all robots are at distance *Big* then the *B_all* object is emitted in the environment.

Both of the output devices, the RGB LED and the movement direction, are controlled by agents that only receive request objects. These request objects are maintained in the respective agents for one simulation step before being consumed. Because of the limited set of possible colors (64) and movement directions (stop, straight, left, right), symbolic objects can be directly mapped to the Kilobot end effector states.

The command agent is responsible for coordinating the input and output agents and is where all of the application specific logic is implemented using P colony programs.

One similarly structured robot controller that uses P colonies is presented in [8]. The main difference between the models, apart from the particularities imposed by the different robots that were modelled, is the fact that the aforementioned model has an input filter that spreads coded input data that only the

specialized input agents can consume and transform into information that can be used by the command agent. The same procedure is done in reverse for the output agents. In [8] the term input filter refers to a function that continuously emits symbolic objects into the P colony environment that correspond to sensor readings. The same is done in reverse for the output filter. If these objects are not processed on time, i.e objects are emitted at a high rate then the model can react non-deterministically due to the fact that objects that represent low and high sensor readings could both be present at the same time. In this scenario, the P colony functioning principle of stochastically choosing only one of many executable programs is applied. The model presented in this paper limits the number of objects that can be exchanged at any one time by agents in an attempt to reduce the number of programs needed and maintains the deterministic execution by using the request-response interaction method.

3 Case study. The dispersion of a swarm of simulated Kilobot robots while ignoring intruders

Based on the previously discussed control model structure we now present a dispersion algorithm that includes an identity detection security method that uses a unique number assigned to each robot and was tested on simulated Kilobot robots. The basic task of the robots is to choose a random direction of movement whenever they sense other neighbor robots that are closer than a threshold distance, and so to disperse in the environment. The development process of new emerging technologies generally does not consider the application security as an explicit design objective but rather an afterthought [5]. Because of the defining characteristics of a robot swarm such as mobility, distributed control, autonomy, local communication and emergent behavior, the importance of a security method has been considered from the early stages of algorithm development. The formal definition of the P colony used to implement the dispersion algorithm that includes an ID based method of ignoring intruders is presented in Figure 2. We used the term Env to refer to the definition of the multiset that represents the initial convents of the P colony environment. This notation is different from the standard definition of a P colony [7] in that we specify the initial contents of the environment, other than only elementary (e) objects. In order to execute the P colony, it was converted to the input language used by the Lulu P colony / Pswarm simulator simulator. The entire source code of the algorithm along with demonstration videos are available at [3]. The details of the input language as well as the functioning principles of the simulator are given in [4]. In order the emphasize the control section of the algorithm, the code for the input/output agents has been discarded.

We used the Kilombo simulator to execute the algorithm on a swarm of Kilobot robots. The main advantage of this simulator is the fact that it is written in the C language and allows the PC simulation of the same code that runs on the robot, reducing the time spent on converting the code between the two platforms [6]. Other distinctive features are the ability to simulate a swarm of

$\Pi_1 = (A, e, f, Env, command, motion, msg\_distance, led\_rgb)$ where:
$A = \{m\_0, m\_S, m\_L, m\_R, c\_R, c\_G, c\_B, c\_W, c\_0, d\_next, B\_all, S\_*, B\_*, id\_*\}$
$Env = \{id\_0, id\_1, id\_2\}$
$command = (\{e, e, e, d\_next\},$
$< e \to e; e \to e; e \to e; d\_next \leftrightarrow e >)$

$< e \to c\_G; e \to m\_S; e \leftrightarrow id\_* / e \leftrightarrow e; e \leftrightarrow S\_* >$
$< e \to c\_G; e \to m\_S; e \leftrightarrow id\_* / e \leftrightarrow e; e \leftrightarrow S\_* >$
$< e \to c\_R; e \to m\_L; e \leftrightarrow id\_* / e \leftrightarrow e; e \leftrightarrow S\_* >$
$< e \to c\_B; e \to m\_R; e \leftrightarrow id\_* / e \leftrightarrow e; e \leftrightarrow S\_* >$

$< e \to e; e \to e; e \to e; e \leftrightarrow B\_* >$
$< e \to e; e \to e; e \to e; B\_* \to d\_next >$

$< e \to c\_W; e \to m\_0; e \to e; e \leftrightarrow B\_all >$

$< c\_G \to e; m\_S \to e; e \to e; S\_* \to d\_next >$
$< c\_R \to e; m\_L \to e; e \to e; S\_* \to d\_next >$
$< c\_B \to e; m\_R \to e; e \to e; S\_* \to d\_next >$

$< c\_G \leftrightarrow e; m\_S \leftrightarrow e; id\_* \leftrightarrow e; S\_* \to d\_next >$
$< c\_R \leftrightarrow e; m\_L \leftrightarrow e; id\_* \leftrightarrow e; S\_* \to d\_next >$
$< c\_B \leftrightarrow e; m\_R \leftrightarrow e; id\_* \leftrightarrow e; S\_* \to d\_next >$

$< c\_W \leftrightarrow e; m\_0 \leftrightarrow e; e \to e; B\_all \to d\_next >$
)

Fig. 2. Dispersion algorithm with self/non-self identification based on a unique symbolic id used by each swarm member

1000 robots at a peak speed of 100 times faster than real robots and the use of configurable sensor noise [6] that helps reduce the differences in runtime behavior between real and simulated Kilobots.

The initial assumption made by the algorithm is that any swarm robot is assigned a unique number that corresponds to a symbolic *id* object such as *id_2* for robot 2. In the case of Kilobot robots this unique number is stored in the non-volatile memory at calibration time and can be used by applications. In this implementation, all robots execute the same compiled code and determine their symbolic id at startup by subtracting the smallest robot id from the swarm (e.g. 60) from their unique robot id (e.g. 65). In this example, the symbolic $id = 65 - 60 = 5$. This conversion is necessary because it allows the P colony to be aware of what robot it is being executed on and consequently expand and execute its programs.

For the purpose of increased clarity and error avoidance we have implemented a simple wildcard expansion function within the Lulu P colony simulator, that allows the expansion of '*' wildcards with each value from a supplied suffix list.

We used this type of expansion in defining the alphabet of the P colony from line 2 in Figure 2, where, for example, the S_* object would be replaced by $\{S\_0, S\_1, S\_2, S\_3, S\_4\}$ if this P colony would be executed with a pre-specified swarm size of five robots. The same type of expansion is done for entire programs in an attempt to avoid repeated definitions.

The actual security measure is implemented in the programs between lines 7 to 10 and is based on a checking rule ($e \leftrightarrow id\_* / e \leftrightarrow e$) that verifies that the associated id object that corresponds to the received S_id object exists in the P colony environment. There are four different programs because in the event of a sensor perception of small distance from a given neighbor (S_id object in the environment) the robot should randomly choose one of the four programs if the associated id object is present in the environment (the neighbor is known). The random program selection is based on the fact that agents that have multiple executing programs have to choose only one for execution and in this case we defined four programs in order to make the robot chose with a probability of 50% to go straight and 25% left or right. For example, if the initially declared swarm size is 4 with robots 0, 1, 2, 3 and the environment has the contents listed in line 3 of Figure 2, then if robot 0 encounters robot 3, it will not move away from it because id_3 is not present in the P colony environment. After checking the id object, it is returned in the environment for later use. This security method is implemented using only P colony programs and does not rely on background security services such as cryptographic functions so as not to move away from the central concept of symbolic control of a robot using membrane computing models. An alternative security measure is proposed in the final section.

An example simulation of the dispersion of 9 robots and one intruder is illustrated in Figure 3 with and without the ID security method. In the example on the left side, the intruder (colored in violet) is moved close to one of the robots that starts to move away from the intruder. These behaviours are also illustrated by two demonstration videos [3].

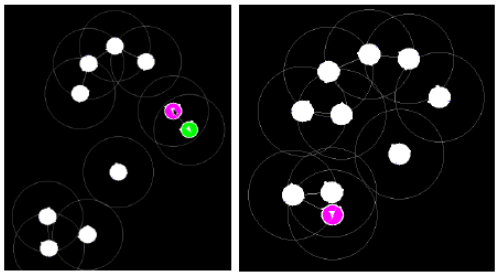


Fig. 3. Simulating the dispersion of 9 robots and the effects of the intruder robot (violet) on the final positions of the robots when running without (left) and with (right) the ID security feature.

4 Conclusions and further work

This paper presents the use of P colonies for dispersing a swarm of robots while ignoring intruders using a security method that is implemented symbolically inside the P colony programs. Future work will include the development and testing of models that abstract the communication between one or more neighbors through the exchange of symbolic objects from one P colony to another. On the security aspect, improvements can be made by not having to rely on a global ID that could be guessed by an intruder. Instead, we seek to develop a signature type authentication method, also implemented at a symbolic level, that would help locally validate that the emitter of a message is indeed a member of the swarm and would improve on the previously mentioned disadvantages.

Acknowledgement

This work was supported by the grant of the Ministry of National Education CNCS-UEFISCDI, project number PN-II-ID-PCE-2012-4-0239, Bioinspired techniques for robotic swarms security (Contract #2/30.08.2013).

References

1. Levent Bayındır. A review of swarm robotics tasks. *Neurocomputing*, 172:292–321, aug 2015.
2. Cătălin Buiu and Mihai Gansari. A new model for interactions between robots in a swarm. In *Electronics, Computers and Artificial Intelligence (ECAI), 2014 6th International Conference on*, pages 5–10, oct 2014.
3. Andrei George Florea and Cătălin Buiu. Demonstration video for the secure dispersion of robots in a swarm using P colonies. `http://dx.doi.org/10.17632/42by9d2f78.1`, 2016.
4. Andrei George Florea and Cătălin Buiu. Development of a software simulator for P colonies. Applications in robotics. *International Journal of Unconventional Computing*, 12(2-3):189–205, 2016.
5. Fiona Higgins, Allan Tomlinson, and Keith M. Martin. Threats to the Swarm : Security Considerations for Swarm Robotics. *International Journal on Advances in Security*, 2(2&3):288–297, 2009.
6. Fredrik Jansson, Matthew Hartley, Martin Hinsch, Ivica Slavkov, Noemí Carranza, Tjelvar S. G. Olsson, Roland M. Dries, Johanna H. Grönqvist, Athanasius F. M. Marée, James Sharpe, Jaap A. Kaandorp, and Verônica A. Grieneisen. Kilombo: a Kilobot simulator to enable effective research in swarm robotics. *arXiv preprint arXiv:1511.04285*, 2015.
7. J. Kelemen, A. Kelemenova, and Gheorghe Păun. Preview of P colonies: A biochemically inspired computing model. In *Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX), (M. Bedau et al., eds.)*, pages 82–86, Boston, Mass., 2004.
8. Miroslav Langer, Ludek Cienciala, Lucie Ciencialová, Michal Perdek, and Alice Kelemenová. An Application of the PCol Automata in Robot Control. In *11th Brainstorming Week on Membrane Computing*, pages 153–164, 2013.

9. Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot: A low cost robot with scalable operations designed for collective behaviors. *Robotics and Autonomous Systems*, 62(7):966–975, jul 2014.
10. Erol Sahin. Swarm Robotics: From Sources of Inspiration to Domains of Application. In *Proceedings of the 2004 International Conference on Swarm Robotics*, SAB'04, pages 10–20, Berlin, 2005. Springer-Verlag.
11. I.A. Wagner, M. Lindenbaum, and A.M. Bruckstein. Distributed covering by ant-robots using evaporating traces. *IEEE Transactions on Robotics and Automation*, 15(5):918–933, 1999.

Remarks on the Computational Power of Some Restricted Variants of P Systems with Active Membranes

Zsolt Gazdag, Gábor Kolonits

Department of Algorithms and their Applications
Faculty of Informatics
Eötvös Loránd University, Budapest, Hungary
{gazdagzs,kolomax}@inf.elte.hu

Abstract. In this paper we consider three restricted variants of P systems with active membranes: (1) P systems using send-out communication rules only, (2) P systems using elementary membrane division and dissolution rules only, and (3) polarizationless P systems using dissolution and unit evolution rules only. We show that every problem in **P** can be solved with uniform families of any of these variants using reasonably weak uniformity conditions. This, using known results on the upper bound of the computational power of variants (1) and (3) yields new characterizations of the class **P**. In the case of variant (2) we provide a further characterization of **P** by giving a semantic restriction on the computations of P systems of this variant.

Keywords: Membrane Computing, P systems with active membranes, computational complexity

1 Introduction

P systems with active membranes were introduced in [20]. These P systems have the possibility of dividing elementary (or even non-elementary) membranes. It was soon discovered that this feature (combined with maximal parallelism) makes this variant a rather powerful computational device, and efficient solutions of problems that are complete in **NP** [10,20,25,31] (or even in **PSPACE** [1,29]) were given. In order to establish the connection between classical complexity classes and P system families, recognizer P systems were introduced in [24]. Since then recognizer P systems are considered as the natural framework to study the computational power of various classes of P system families. Among the many research lines in Membrane Computing, one is to find efficient solutions of computationally hard problems by various types of recognizer P systems with active membranes (see e.g. [2,3,4,17,18,23]).

It is not too surprising that membrane division is necessary in these systems to solve computationally hard problems efficiently [31]. However, in [21] Păun conjectured that polarization is also necessary. More precisely, Păun's conjecture

(which is also known as the *P conjecture* in the literature) sounds as follows: polarizationless P systems with active membranes working in polynomial time can solve only problems in **P** if non-elementary membrane division rules are not allowed. Although the P conjecture has not been proven yet, there are some partial results. In [8] it was shown that without dissolution rules these systems can solve exactly the problems in **P**. The conjecture was also confirmed in the following cases: when dissolution rules are allowed, but the P systems can employ only restricted, so-called symmetric, division rules [12], and when the initial membrane structure is a linearly nested sequence of membranes, and the system can employ only dissolution and elementary membrane division rules [30].

It was observed in [13] that the **P** lower bound in the characterization of **P** in [8] comes from the polynomial uniformity of the examined P systems. In fact, according to [11] the used uniformity condition dominates the computational power of uniform families of polarizationless P systems with no dissolution rules. This initiated a sequence of papers where P systems with active membranes under reasonably tight uniformity conditions were examined [15,16]. Moreover, several solutions of problems in **P** with restricted classes of P systems under tight uniformity conditions were given [5,9,14,15].

In this paper we continue the work in this research line. First we show that uniform families of P systems with active membranes using send-out communication rules only can solve every problem in **P**. Then we show a similar result when the applicable rules are the elementary membrane division and the dissolution rules. The proofs are given by solving a restricted, but still **P**-complete variant of the well know HornSat problem, the satisfiability problem of Horn formulas.

Finally, we show that uniform families of polarizationless P systems with active membranes using dissolution and unit rules can simulate polynomial time Turing machines efficiently (a unit rule is such an evolution rule which introduces exactly one object during its application). This result is stronger than the one appearing in [6] since there communication and not restricted evolution rules were used too. In [15] a solution of a **P**-complete problem was given using dissolution and restricted evolution rules only, however the presented family of P systems was semi-uniform.

Using the **P** upper bound given in [31], our first and third results give new characterizations of **P** in terms of Membrane Computing techniques. In our second result we use P systems where the initial membrane structure is a linearly nested sequence of membranes, and during the computation the number of membranes on the deepest level is at most two. It can be seen that the set of those problems that can be solved by those P systems with active membranes which have this semantic restriction during their computations are in **P**. This yields another characterization of the complexity class **P**.

2 Preliminaries

Here we recall the necessary notions used later. Nevertheless, we assume that the reader is familiar with the basic concepts of formal language theory, propositional logic, and Membrane Computing techniques (for a comprehensive guide to these topics see e.g. [7,22,27], respectively). $\mathbb{N}$ denotes the set of natural numbers. For $n, m \in \mathbb{N}$, $n < m$, $[n, m]$ denotes the set $\{n, n+1, \ldots, m\}$. If $n = 1$, then $[n, m]$ is denoted by $[m]$.

Propositional formulas and the HornSat *problem.* A *propositional variable* is a variable whose value can be either *true* or *false*. If it is not confusing, we will often call propositional variables simply *variables*. We fix an infinite set $Var = \{v_1, v_2, v_3, \ldots\}$ of variables. For a number $n \in \mathbb{N}$, Var_n is the set $\{v_1, \ldots, v_n\}$. An *interpretation* of the variables in Var_n is a function $\mathcal{I} : Var_n \to \{true,\ false\}$.

The propositional variables and their *negations* are called *literals*. l is a *positive* (resp. *negative*) literal, if $l = x$ (resp. $l = \neg x$), for some $x \in Var$, where $\neg$ denotes the operation of *negation*. A *clause* $\mathcal{C}$ is a *disjunction* of finitely many pairwise different literals satisfying that there is no $x \in Var$ such that both x and $\neg x$ occur in $\mathcal{C}$. A clause $\mathcal{C}$ is a *positive unit clause* if it contains exactly one positive literal and no negative literals. A formula in *conjunctive normal form* (CNF) is a conjunction of finitely many clauses. Let φ be a formula in CNF. We will sometimes consider φ as a finite set of clauses, where the clauses are finite sets of literals. φ is satisfiable, if there is an interpretation under which φ evaluates to *true*. Moreover, φ is a *Horn formula* if every clause in φ contains at most one positive literal.

The HornSat problem sounds as follows: *given a Horn formula φ, decide if φ is satisfiable*. It is known that HornSat is **P**-complete (see e.g. [19]). Let Horn3Sat be that restriction of HornSat where every clause of the input formula can contain at most three literals. Moreover, let Horn3SatNorm be that restriction of Horn3Sat where the input formula is in the following normal form: every clause of the formula is either a positive unit clause or it contains exactly two negative literals and at most one positive literal. For example, $x \wedge (\neg x \vee y) \wedge (\neg y \vee \neg z \vee u)$ $(x, y, z, u \in Var)$ is an instance of Horn3Sat, but not of Horn3SatNorm, since $(\neg x \vee y)$ neither is a positive unite clause nor contains exactly two negative literals. However, $x \wedge (\neg x \vee \neg y) \wedge (\neg y \vee \neg z \vee u)$ is an instance of Horn3SatNorm.

Next we show that Horn3SatNorm is **P**-complete. The proof resembles the standard **NP**-completeness proof of the 3Sat problem (the 3Sat problem is the satisfiability problem of those formulas in CNF which can have only clauses with three literals, see e.g. [28]).

Proposition 1. Horn3SatNorm *is* **P***-complete.*

Proof. Since this problem is a restriction of HornSat, it is in **P**. Thus, it is enough to show that HornSat can be reduced using logarithmic space to Horn3SatNorm. First we show that HornSat reduces to Horn3Sat. Let

φ be a Horn formula over the variables in Var_n ($n \in \mathbb{N}$). We construct an instance φ' of Horn3Sat such that φ' is satisfiable if and only if φ is satisfiable. Let $\mathcal{C}$ be a clause in φ. If $\mathcal{C}$ has at most three literals, then let $\mathcal{C}$ be a clause of φ'. Otherwise, assume that $\mathcal{C} = x_1 \vee \neg x_2 \vee \cdots \vee \neg x_k$ for some $k \in [4, n]$ and $x_i \in Var_n$ ($i \in [k]$). It can be easily seen that $\mathcal{C}$ is satisfiable if and only if $(x_1 \vee \neg x_2 \vee \neg y) \wedge (y \vee \neg x_3 \vee \cdots \vee \neg x_k)$ is satisfiable, where y is a new variable, not included in Var_n. In this way we can construct the formula $(x_1 \vee \neg x_2 \vee \neg y_1) \wedge (y_1 \vee \neg x_3 \vee \neg y_2) \wedge \cdots \wedge (y_{k-3} \vee \neg x_{k-1} \vee \neg x_k)$, which is satisfiable (over $Var_n \cup \{y_1, \ldots, y_{k-3}\}$) if and only if $\mathcal{C}$ is satisfiable (over Var_n). To a clause with no positive literals one can give a very similar construction. Then we add these new clauses to φ'. Clearly, φ' is satisfiable if and only if φ is satisfiable, and the mapping $\varphi \mapsto \varphi'$ can be carried out by a deterministic Turing machine using logarithmic space in the size of φ.

Next we show that Horn3Sat reduces to Horn3SatNorm. To this end let φ be an instance of Horn3Sat with variables in Var_n. We construct an instance φ' of Horn3SatNorm such that φ' is satisfiable if and only if φ is satisfiable. For every clause $\mathcal{C}$ of φ, if $\mathcal{C}$ corresponds to the restrictions made on the instances of Horn3SatNorm, then let $\mathcal{C}$ be a clause of φ'. Otherwise we replace $\mathcal{C}$ with the set $\mathcal{C}'$ of clauses defined as follows:

- if $\mathcal{C} = \neg x$, then let $\mathcal{C}' = \{\neg x \vee \neg y, y\}$,
- if $\mathcal{C} = x_1 \vee \neg x_2$, then let $\mathcal{C}' = \{x_1 \vee \neg x_2 \vee \neg y, y\}$, and
- if $\mathcal{C} = \neg x_1 \vee \neg x_2 \vee \neg x_3$, then let $\mathcal{C}' = \{\neg x_1 \vee \neg x_2 \vee y, \neg y \vee \neg x_3\}$,

where $x, x_1, x_2, x_3 \in Var_n$ and y is always a new variable not used yet during the construction. Clearly the clauses in $\mathcal{C}'$ always have the desired forms, and φ' is satisfiable if and only if φ is satisfiable. Moreover, the described construction can be carried out by a logarithmic space Turing machine. Thus, since logarithmic space reductions are closed under composition, we have that HornSat can be efficiently reduced to Horn3SatNorm, which finishes the proof of the statement.

Turing machines. In this paper we will use that variant of Turing machines which appears, e.g., in [28]. A *(deterministic) Turing machine* is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where

- Q is the finite set of *states*,
- Σ is the *input alphabet*,
- Γ is the tape alphabet including Σ and a distinguished symbol $\sqcup \notin \Sigma$, called the *blank symbol*,
- $\delta : (Q - \{q_a, q_r\}) \times \Gamma \to Q \times \Gamma \times \{-1, 1\}$ is the *transition function*; the ith component of $\delta(q, X)$ ($i \in [1, 3], q \in Q - \{q_a, q_r\}, X \in \Gamma$) is denoted by $\mathrm{proj}_i(\delta(q, X))$,
- q_0, q_a, and q_r are the *initial*, *accepting*, and *rejecting states*, respectively.

M works on a single infinite tape that is closed on the left-hand side. During the computation of M, the tape contains only finitely many non-blank symbols,

and it is blank elsewhere. Let $w \in \Sigma^*$. The initial configuration of M on w is the configuration where w is placed at the beginning of the tape, the head points to the first letter of w, and the current state of M is q_0. A computation step performed by M can be described as follows. If M is in state p and the head of M reads the symbol X, then M changes its state to q and writes X' onto X if and only if $\delta(p, X) = (q, X', d)$, for some $d \in \{-1, 1\}$. Moreover, if $d = 1$ (resp. $d = -1$), then M moves its head one position to the right (resp. to the left) (by definition, M can never move the head off the left-hand end of the tape even if the head points to the first cell and $d = -1$). We say that M accepts (resp. rejects) w, if M can reach from the initial configuration on w the accepting state q_a (resp. the rejecting state q_r). We note here that M can stop only in these states. The language accepted by M is the set $L(M)$ consisting of those words in Σ^* that are accepted by M.

P systems with active membranes. In this paper we consider several restricted variants of P systems with active membranes. In general, a *P system* with active membranes [20] is a construct of the form $\Pi = (\Gamma, H, \mu, w_1, \ldots, w_m, R)$, where m is the initial *degree* of the system, Γ is the alphabet of *objects*, H is a finite set of *labels* of the membranes; μ is a *membrane structure* consisting of m membranes and labelled with elements of H; $w_1, \ldots, w_m \subseteq \Gamma^*$ are the *initial multisets of objects* placed in the m regions of μ; and R is a finite set of *rules* defined as follows:

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

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

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

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

(e) $[a]_h^{e_1} \to [b]_h^{e_2}[c]_h^{e_3}$, for $e_1, e_2, e_3 \in \{+, -, 0\}$, $h \in H$, $a, b, c \in \Gamma$
(*division* rules for elementary membranes; in reaction with an object, the membrane is divided into two membranes with possibly different polarizations; 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 it is usual in membrane computing, P systems with active membranes work in a *maximally parallel* manner: at each step the system first nondeterministically

assigns appropriate rules to the objects of the system such that the assigned multiset S of rules satisfies the following properties: (i) at most one rule from S is assigned to any object of the system, (ii) a membrane can be the subject of at most one rule in S, and (iii) S is maximal among the multisets of rules satisfying (i) and (ii).

We call an evolution rule $[a \to v]_h^e$ with $|v| = 1$ a *unit rule*. A *layer* is a non-branching membrane structure, that is a layer has the form $[\ldots[\ ]_{h_1}\ldots]_{h_n}$ ($n \geq 1, h_1, \ldots, h_n \in H$). For two layers $\mu_1 = [\ldots[\ ]_{h_1}\ldots]_{h_j}$ and $\mu_2 = [\ldots[\ ]_{g_1}\ldots]_{g_k}$ ($j, k \geq 1, h_1, \ldots, h_j, g_1, \ldots, g_k \in H$), the *composition* $\mu_1[\mu_2]$ of μ_1 and μ_2 is the layer $[\ldots[[\ldots[\ ]_{g_1}\ldots]_{g_k}]_{h_1}\ldots]_{h_j}$. For improving the readability of the paper, we will often call a composition of finitely many layers a *block*.

Recognizer P systems. A *recognizer P system* [24,26] is a P system Π with a designated *input* membrane and having the following properties. The alphabet Γ of objects has two designated elements *yes* and *no*. Every computation of Π halts and sends to the environment the same object which is either *yes* or *no*, and these objects are sent out in the last step of the computation (if the examined P system model does not have send-out communication rules, then the output of the systems appears in the skin membrane). The *input* of Π is a multiset over Γ, which is added to the input membrane of the system in the initial configuration.

Uniform families of P systems. A family $\mathbf{\Pi} = \{\Pi(i)\}_{i\in\mathbb{N}}$ of recognizer P systems *decides a problem* L if, for every instance x of L with length n, starting $\Pi(n)$ with an appropriate encoding of x in its input membrane leads to $\Pi(n)$ sending into the environment *yes* if and only if $x \in L$.

We will use uniform families of recognizer P systems to solve problems in $\mathbf{P}$. Clearly, we should use such a uniformity condition that is reasonably weak to work with in class $\mathbf{P}$. According to the widely believed fact that Turing machines using logarithmic space are strictly weaker than Turing machines working in polynomial time, we will use logarithmic space uniform families of P systems. We denote by $\mathbf{L}$ the family of functions that can be computed by Turing machines using logarithmic amount of space.

Assume that a family $\mathbf{\Pi} = \{\Pi(i)\}_{i\in\mathbb{N}}$ of recognizer P systems decides a problem L. $\mathbf{\Pi}$ is called **(L,L)**-*uniform* if and only if (i) there are functions $f, cod \in \mathbf{L}$ such that, for every $n \in \mathbb{N}$, $\Pi(n) = f(1^n)$ (i.e., the P system $\Pi(n)$ can be constructed by a logarithmic space Turing machine from the unary representation of n); (ii) for every instance x of L with size n, $cod(x)$ is a multiset encoding x over the alphabet of objects in $\Pi(n)$.

For a type $\mathcal{F}$ of recognizer P systems, we denote by $(\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{F}}$ the class of those problems that can be decided by $(\mathbf{L}, \mathbf{L})$-uniform families of P systems of type $\mathcal{F}$ working in polynomial time. $\mathcal{AM}_{+out}$ (resp. $\mathcal{AM}_{+e,+d}$) denotes the family of P systems with active membranes having send-out communication (resp. division and dissolution) rules only. Similarly, $\mathcal{AM}^0_{+u,+d}$ denotes the family of polarizationless P systems having unit rules and dissolution rules only.

3 Results

Here we show that recognizer P systems of type $\mathcal{AM}_{+out}$, $\mathcal{AM}_{+e,+d}$, or $\mathcal{AM}^0_{+u,+d}$ and working in polynomial time are capable to solve every problem in **P**. First we consider two solutions of HORN3SATNORM, then we give an efficient simulation of Turing machines.

3.1 The solution of HORN3SATNORM

By definition, if φ is an instance of HORN3SATNORM, then every clause of φ is either a positive unit clause or it has exactly two negative literals. In the rest of this section by a clause we mean a clause having this property. Using the well known equivalences of propositional logic, a clause having exactly two negative literals $\neg x$ and $\neg y$ can be written in the form $x \wedge y \to \downarrow$ or $x \wedge y \to z$, where z is a variable, $\to$ denotes the operation of implication and $\downarrow$ denotes a formula with constant *false* truth value. We will often use these expressions to denote the corresponding clauses of the input formula (in fact, we will often call these expressions clauses, although strictly speaking they are not clauses). Moreover, for the sake of simplicity, we will not indicate the sign $\wedge$ of conjunction in the left-hand side of these expressions.

Let φ be an instance of HORN3SATNORM with variables in Var_n ($n \geq 1$). Clearly, if φ is *true* in an interpretation $\mathcal{I}$, then $\mathcal{I}(x) = true$ must hold for every positive unite clause $\{x\}$ in φ. Assume now that $\mathcal{C} = xy \to z$ is a clause of φ, where x, y are variables and z is either a variable or $\downarrow$. We observe that if $\mathcal{I}(x) = \mathcal{I}(y) = true$, then $\mathcal{C}$ is *true* in $\mathcal{I}$ if and only if z is *true* too. That is, if $z = \downarrow$, then x, y, z cannot be all *true* in $\mathcal{I}$. We will use these observations in the following algorithm H3SN, which decides if φ is satisfiable or not. Let $\mathcal{N}(n)$ denote the set of those clauses over variables in Var_n which contain exactly two negative literals, and let $m = |\mathcal{N}(n)|$. In the rest of this section we assume a fixed enumeration $c_1, \ldots, c_m$ of clauses in $\mathcal{N}(n)$.

Algorithm H3SN

1. **input**: φ
2. $X := \{x \in Var_n \mid x \in \varphi\}$ // x is a positive unit clause in φ
3. **For** $i = 1 \ldots n$ do
4. **For** $j = 1 \ldots m$ **do**
5. **If** $c_j = xy \to u \in \varphi$ **and** $x, y \in X$ **then** $X := X \cup \{u\}$
6. **If** $\downarrow$ is in X **then return** *no*
7. **else return** *yes*

To demonstrate the work of H3SN consider the following example (see also the solution of HORNSAT in [19]). Let $\varphi = x \wedge y \wedge (xy \to z) \wedge (xz \to \downarrow)$, where $x, y, z \in Var_n$. Then, initially, $X = \{x, y\}$. Since $x, y \in X$ and $xy \to z \in \varphi$, X becomes $\{x, y, z\}$. Then, since $x, z \in X$ and $xz \to \downarrow \in \varphi$, X becomes $\{x, y, z, \downarrow\}$. After this the value of X remains the same until H3SN halts. Thus, since $\downarrow \in X$, H3SN outputs *no*. This is correct as φ is unsatisfiable.

In this section we give two families of P systems with rather restricted sets of applicable rules to solve the HORN3SATNORM problem in polynomial time. Both solutions are based on Algorithm H3SN. In these solutions the P systems cannot employ evolution and send-in communication rules. In addition, in the first solution dissolution and membrane division rules, while in the second solution send-out communication rules are also not allowed.

In both solutions the P systems, roughly, work as follows. Let φ be an instance of HORN3SATNORM with variables in Var_n ($n \geq 1$). The initial membrane structure consists of n blocks, and the innermost membrane contains $cod(\varphi)$ (that is, the encoding of φ). A block r_i corresponds to the ith round of the main loop in Algorithm H3SN.

For an arbitrary clause $\mathcal{C}$ with variables in Var_n, $cod(\varphi)$ contains an object $O_{\mathcal{C}}^{\exists}$ or $O_{\mathcal{C}}^{\nexists}$ (but not both) according to whether $\mathcal{C}$ occurs in φ or not. Moreover, for every clause of the form $xy \to u$ ($x, y \in Var_n, u \in Var_n \cup \{\downarrow\}$), r_i has a layer l whose membranes are indexed by this clause. The objects in the inner membrane of l go through l (either by send-out communication or by dissolution rules, according to the used model), and during this the system performs the following task. It first checks whether all the objects $O_{xy\to u}^{\exists}$, $O_x^{\exists}$, $O_y^{\exists}$, and $O_u^{\nexists}$ were present in the innermost membrane of l. If yes, then the system rewrites $O_u^{\nexists}$ to $O_u^{\exists}$. In this way the system can determine which variables of φ must be *true* in order to make φ *true* in an interpretation. After performing the above task in all layers of block r_n, the skin contains either $O_{\downarrow}^{\exists}$ or $O_{\downarrow}^{\nexists}$. If $O_{\downarrow}^{\exists}$ occurs in the skin, then φ cannot be satisfied and the system introduces object *no*, otherwise it introduces *yes*. Notice that while H3SN computes the set of those variables that must be *true* in order make φ *true* in an interpretation, the above described P systems consider these variables as they were positive unit clauses of the formula. However this behaviour is correct, since if we know that a variable x should be *true* in any interpretation that makes φ *true*, then $\varphi \wedge x$ is satisfiable if and only if φ is satisfiable.

Formally, we encode an instance φ of HORN3SATNORM with variables in Var_n as follows. First, let

$$\Sigma(n) = \{O^e \mid O \in V(n) \cup C(n), e \in \{\exists, \nexists\}\},$$

where $V(n) = \{V_u \mid u \in Var_n \cup \{\downarrow\}\}$ and $C(n) = \{C_{xy\to u} \mid x, y \in Var_n, u \in Var_n \cup \{\downarrow\}\}$. Then the encoding of φ is $cod(\varphi) = \{O_c^{\exists} \in \Sigma(n) \mid c \in \varphi\} \cup \{O_c^{\nexists} \in \Sigma(n) \mid c \notin \varphi\} \cup \{V_{\downarrow}^{\nexists}\}$. We note here that technically there is no need to distinguish in the notation between positive unite clauses and clauses having two negative literals. Nevertheless, we decided to do so to improve the readability of the constructions. Since the size of φ is clearly polynomial in n, it can be seen that *cod* is a function in **L**.

A solution using send-out communication rules only. Here we solve HORN3SATNORM with a family $\mathbf{\Pi_1} = \{\Pi_1(n)\}_{n\in\mathbb{N}}$ of recognizer P systems of type $\mathcal{AM}_{+out}$, where $\Pi_1(n) = (\Gamma(n), H(n), \mu(n), W(n), R(n))$ is defined as follows:

- $\Gamma(n) = \Sigma(n) \cup \{V_x^{\exists +} \mid x \in Var_n \cup \{\downarrow\}\} \cup \{yes, no\}$.
- $H = \{(xy \to u, \alpha) \mid x, y \in Var_n, u \in Var_n \cup \{\downarrow\}, \alpha \in \{a, b, c\}\} \cup \{s_k \mid k \in [m+n]\} \cup \{skin\}$.
- $\mu(n) = S[r_n[r_{n-1}[\ldots r_2[r_1]\ldots]]$, where $S = [[[\ ]_{s_1} \ldots]_{s_{m+n}}]_{skin}$ and, for every $i \in [n]$, r_i is a block defined as follows. $r_i = l_{c_m}[\ldots l_{c_2}[l_{c_1}]\ldots]$, where, for every $j \in [m]$, the layer l_{c_j} has the form $[[[\ ]_{(c_j,a)}]_{(c_j,b)}]_{(c_j,c)}$ (see Fig. 1).

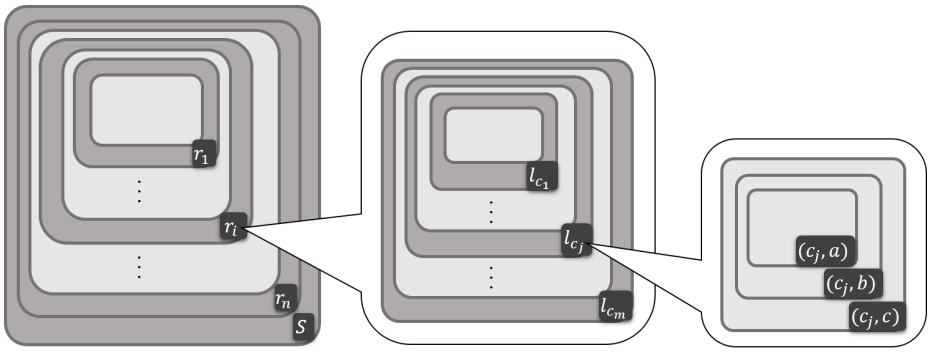


Fig. 1. The initial membrane structure of $\Pi_1(n)$

- The input membrane is the innermost membrane in the initial membrane structure.
- $W(n)$ is the sequence of empty initial multisets.
- R consists of the following subsets of rules, where $x, y \in Var_n$ and $u \in Var_n \cup \{\downarrow\}$:

  (1) $[C^{\exists}_{xy\to u}]^0_{(xy\to u,a)} \to [\ ]^+_{(xy\to u,a)} C^{\exists}_{xy\to u}$,
  $[C^{\exists}_{xy\to u}]^0_{(xy\to u,\beta)} \to [\ ]^0_{(xy\to u,\beta)} C^{\exists}_{xy\to u}$,
  $[C^{\nexists}_{xy\to u}]^0_{(xy\to u,\alpha)} \to [\ ]^-_{(xy\to u,\alpha)} C^{\nexists}_{xy\to u}$ $(\alpha \in \{a, b, c\}, \beta \in \{b, c\})$.
  These rules are used to initialize the layers in the following sense: an object representing a clause in φ sets the charge of the first membrane in the corresponding layer to positive, and keeps the neutral charges of the second and third membranes in that layer, while an object representing a clause not in φ sets the charges of all membranes in the corresponding layer to negative.

  (2) $[V^e_v]^-_{(xy\to u,\alpha)} \to [\ ]^-_{(xy\to u,\alpha)} V^e_v$,
  $[C^e_{rs\to v}]^-_{(xy\to u,\alpha)} \to [\ ]^-_{(xy\to u,\alpha)} C^e_{rs\to v}$
  $(e \in \{\exists, \nexists\}, r, s \in Var_n, v \in Var_n \cup \{\downarrow\}, \alpha \in \{a, b, c\})$.
  Every membrane with negative charge lets all of the objects pass through itself.

  (3) $[V^{\exists}_x]^+_{(xy\to u,a)} \to [\ ]^-_{(xy\to u,a)} V^{\exists +}_x$,
  $[V^{\exists +}_x]^0_{(xy\to u,b)} \to [\ ]^+_{(xy\to u,b)} V^{\exists}_x$.

If φ has a clause $xy \to u$, that is, the membrane with label $(xy \to u, a)$ has positive charge, and $V_x^{\exists}$ exists in this membrane, then these rules are used to store this information in the positive charge of the membrane with label $(xy \to u, b)$.

(4) $[V_y^{\exists}]^+_{(xy\to u,b)} \to [\,]^-_{(xy\to u,b)} V_y^{\exists +}$,
$[V_y^{\exists +}]^0_{(xy\to u,c)} \to [\,]^+_{(xy\to u,c)} V_y^{\exists}$.
If the membrane with label $(xy \to u, b)$ has positive charge and $V_y^{\exists}$ exists in this membrane, then these rules are used to store this information in the positive charge of the membrane with label $(xy \to u, c)$.

(5) $[V_u^{\nexists}]^+_{(xy\to u,c)} \to [\,]^-_{(xy\to u,c)} V_u^{\exists}$.
The positive charge of the membrane with label $(xy \to u, c)$ indicates that $xy \to u$ is a clause of the system and that both variables x and y has to be *true* in an interpretation in order to make φ *true*. Thus, with this rule the system rewrites $V_u^{\nexists}$ to $V_u^{\exists}$ indicating that u must be also *true* to make φ *true*.

(6) $[V_u^{\exists}]^p_{(xy\to u,\alpha)} \to [\,]^-_{(xy\to u,\alpha)} V_u^{\exists}$ $(p \in \{+, 0\}, \alpha \in \{a, b, c\})$.
If the system already knows that u must be *true* to make φ *true*, then the charges of the corresponding membranes are set to negative.

(7) $[V_x^{\nexists}]^p_{(xy\to u,\alpha)} \to [\,]^-_{(xy\to u,\alpha)} V_x^{\nexists}$,
$[V_y^{\nexists}]^p_{(xy\to u,\alpha)} \to [\,]^-_{(xy\to u,\alpha)} V_y^{\nexists}$ $(p \in \{+, 0\}, \alpha \in \{a, b, c\})$.
If any of the variables on the left-hand side of a clause $xy \to u$ is not considered to be *true* yet, then the charges of membranes of the corresponding layer are set to negative by these rules, and $V_u^{\exists}$ cannot be introduced by this layer.

(8) $[V_{\downarrow}^{e}]^0_{s_k} \to [\;\,]^0_{s_k} V_{\downarrow}^{e}$, $[V_{\downarrow}^{\exists}]^0_{skin} \to [\,]^0_{skin} yes$, $[V_{\downarrow}^{\nexists}]^0_{skin} \to [\,]^0_{skin} no$
$(k \in [m + n], e \in \{\exists, \nexists\})$.
The first rule is used to move object $V_{\downarrow}^{\exists}$ or $V_{\downarrow}^{\nexists}$ towards the skin membrane. When they arrive at the skin, the system sends to the environment the correct answer.

Correctness, running time, and $(\mathbf{L}, \mathbf{L})$*-uniformity.* First we observe that during the computation of $\Pi_1(n)$ the following holds.

1. If all the membranes in a layer l have negative charge, then l does not contribute to the computation, i.e. all objects pass through the membranes of l without any change.
2. For every $\mathcal{C} \in C(n)$, either $C_{\mathcal{C}}^{\exists}$ or $C_{\mathcal{C}}^{\nexists}$ (but not both) occurs in the system (the same object during the whole computation).
3. For every $x \in Var_n \cup \{\downarrow\}$, exactly one of $V_x^{\exists}$, $V_x^{\exists +}$ or $V_x^{\nexists}$ occurs in the system (except the last step, where $V_{\downarrow}^{\exists}$ (resp. $V_{\downarrow}^{\nexists}$) is rewritten to *no* (resp. *yes*)). Indeed, the rules that can change an object of this form are rules in (3)-(5) (not counting the rules that introduce *yes* or *no* at the last step of the computation). Rules in (3) (resp. in (4)) change $V_x^{\exists}$ to $V_x^{\exists +}$ (resp. $V_y^{\exists}$

to $V_y^{\exists+}$) and $V_x^{\exists+}$ to $V_x^{\exists}$ (resp. $V_y^{\exists+}$ to $V_y^{\exists}$). Rules in (5) remove $V_u^{\nexists}$ and introduce $V_u^{\exists}$. Thus the observation remains true after applying these rules.
4. If an object $V_x^{\exists}$ occurs in the systems, then $V_x^{\nexists}$ won't be introduced during the computation.

Now consider a layer $l_{xy\to u}$ $(x, y \in Var_n, u \in Var_n \cup \{\downarrow\})$. For every object in $\Gamma \setminus \{yes, no\}$ passing through the layer the following holds.

5. $C_{xy\to u}^{\exists}$, $C_{xy\to u}^{\nexists}$, $V_x^{\nexists}$, $V_y^{\nexists}$, and $V_u^{\exists}$ are unchanged (but they may change the polarizations of the membranes in the layer).
6. $V_x^{\exists}$ and $V_y^{\exists}$ can change only when the corresponding membrane's charge is positive (first rules in (3) and (4)). However, if they change, then the second rules in (3) and (4) will be applied in the next step.
7. $V_u^{\nexists}$ may be changed to $V_u^{\exists}$ by the rules in (5).
8. Any other objects are unchanged (and they don't change any of the polarizations in the layer). Moreover, they can only pass through membranes with negative polarization in this layer.

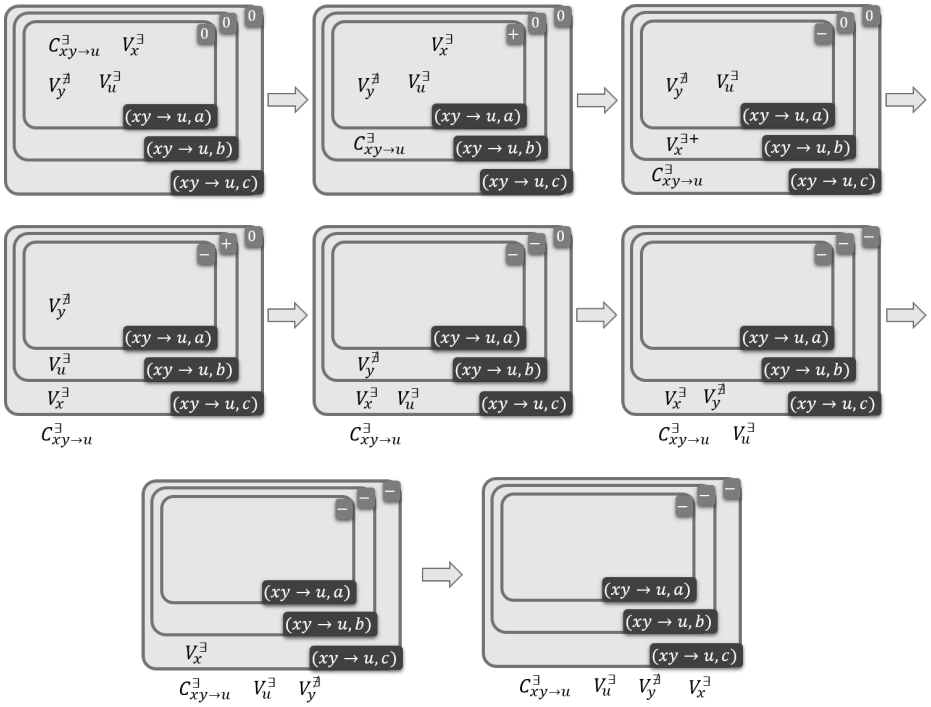


Fig. 2. One of the possible computations in a layer $l_{xy\to u}$ of $\Pi_1(n)$ when it contains $C_{xy\to u}^{\exists}$, $V_x^{\exists}$, $V_y^{\nexists}$, and $V_u^{\exists}$.

Using these observations and also the comments given after the rules, one can see that the following holds. At the beginning of the computation every membrane in $l_{xy\to u}$ has neutral charge. According to the objects that pass through this layer we can distinguish the following cases.

1. All of the objects $C^{\exists}_{xy\to u}$, $V^{\exists}_x$, $V^{\exists}_y$, and $V^{\nexists}_u$ pass through the membranes of $l_{xy\to u}$. Then the system rewrites the object $V^{\nexists}_u$ to $V^{\exists}_u$.
2. Any of the objects $C^{\nexists}_{xy\to u}$, $V^{\nexists}_x$, $V^{\nexists}_y$, or $V^{\exists}_u$ passes through the membranes of $l_{xy\to u}$. Then the charge of every membrane in $l_{xy\to u}$ is set to negative, and thus this layer cannot contribute to the computation. (Notice that in this case the computation is not deterministic but confluent, i.e., all the possible computations in the layer yield the same result. For an example of such a computation see Fig. 2.)

It follows that the objects passing through the layer $l_{xy\to u}$ simulate step 5 of Algorithm H3SN. Thus, sending objects through a block corresponds to performing steps $4-5$ of this algorithm. Since steps $4-5$ are performed n times by the algorithm, the work of the P system in the n blocks corresponds to the work of the algorithm. Thus, $V^{\exists}_{\downarrow}$ or $V^{\nexists}_{\downarrow}$ eventually appears in membrane s_1. In the next $m+n$ steps this object gets to the skin by rules in (8). There the system computes *yes* or *no* accordingly, which is then sent to the environment. It can be seen that during this computation all the other objects occurring in the system arrive to membrane s_1, and the computation halts.

This justifies the correctness of $\Pi_1(n)$. Since $\Pi_1(n)$ has polynomial number of objects in the initial configuration and no evolution rules are performed during its work, sending all the objects through a block takes polynomial steps. Thus the running time of $\Pi_1(n)$ is also polynomial.

It can be seen that all the ingredients of $\Pi_1(n)$ can be enumerated by a logarithmic space Turing machine. Thus, using that Horn3SatNorm is **P**-complete, we get the following result.

Theorem 1. $\mathbf{P} \subseteq (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}_{+out}}$.

A solution using elementary membrane division and dissolution rules only. In this subsection we solve Horn3SatNorm with a family $\mathbf{\Pi_2} = \{\Pi_2(n)\}_{n\in\mathbb{N}}$ of recognizer P systems of type $\mathcal{AM}_{+e,+d}$. The solution is similar to the one given in the previous subsection, however, there is a substantial difference: here the presence of the necessary objects to simulate step 5 of Algorithm H3SN are checked by the application of membrane division rules. Consequently, those objects that do not take part in the simulation are duplicated several times. In particular, at certain points of the computation the P system has multiple copies of objects of the form $V^{\nexists}_x$. However, the correctness of the computation requires that at the beginning of the work in a layer there is at most one copy of objects of this form. Therefore we will apply special layers that will remove those objects that could cause the system to give incorrect results. The following is the formal definition of $\Pi_2(n) = (\Gamma(n), H(n), \mu(n), W(n), R(n))$:

- $\Gamma(n) = \Sigma(n) \cup \{w, \overline{w}_1, \overline{w}_2, \#, \$\} \cup \{yes, no\}$.
- $H(n) = \{skin, s\} \cup \{(xy \to u, \alpha) \mid x, y \in Var_n, u \in Var_n \cup \{\downarrow\}, \alpha \in \{a, b, c, d\}\} \cup \{d_O \mid O \in V(n) \cup C(n) \cup \{w\}\}$.
- $\mu(n)$ is defined as follows (see also Fig. 3). Let $\mathcal{C} = xy \to u$ be a clause ($x, y \in Var_n, u \in Var_n \cup \{\downarrow\}$) and $l_{\mathcal{C}}$ be the layer $D_{\mathcal{C}}[M_{\mathcal{C}}]$, where $D_{\mathcal{C}}$ and $M_{\mathcal{C}}$ are defined as follows:

$$M_{\mathcal{C}} = [[[[[[\ ]_{(xy\to u,a)}]_{(xy\to u,b)}]_{d_w}]_{(xy\to u,c)}]_{d_w}]_{(xy\to u,d)}$$

and $D_{\mathcal{C}}$ is a layer containing, for every $O \in V(n) \cup C(n)$, the membrane $[\ ]_{d_O}$ fifteen times if $O \neq V_u$, and once, otherwise. Intuitively, $M_{\mathcal{C}}$ is that part of the layer which is responsible to simulate step 5 in Algorithm H3SN, and layer $D_{\mathcal{C}}$ is used (together with membranes with label d_w in $M_{\mathcal{C}}$) to remove those objects that are produced by the used division rules, but should be removed in order to keep the behaviour of the system correct.

To finish the construction, let $\mu(n) = S[r_n[r_{n-1}[\dots r_2[r_1]\dots]]$, where $S = [[\ ]_s]_{skin}$ and, for every $i \in [n]$, r_i is the block $l_{c_m}[\dots l_{c_2}[l_{c_1}]\dots]$.

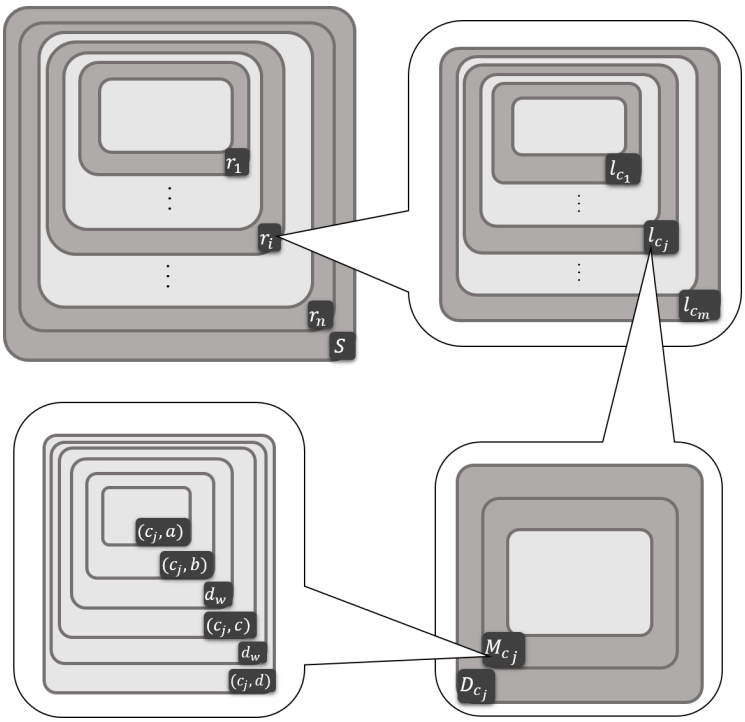


Fig. 3. The initial membrane structure of $\Pi_2(n)$

– The input membrane is the innermost membrane in the initial membrane structure.
– $W(n)$ is a sequence of empty initial multisets.
– R consists of the following subsets of rules, where $x, y \in Var_n$ and $u \in Var_n \cup \{\downarrow\}$:

(1) $[V_u^{\nexists}]^0_{(xy\to u,a)} \to [w]^-_{(xy\to u,a)}[\#]^-_{(xy\to u,a)}$,
$[V_u^{\exists}]^0_{(xy\to u,a)} \to [\overline{w}_1]^-_{(xy\to u,a)}[\#]^-_{(xy\to u,a)}$.
These rules are used to decide if $V_u^{\nexists}$ or $V_u^{\exists}$ is present in a membrane with label $(xy \to u, a)$. If $V_u^{\nexists}$ is present, then the system introduces w which indicates that the system should work further to decide if $V_u^{\exists}$ should be introduced or not. Object $\overline{w}_1$ indicates that $V_u^{\exists}$ is present in the system and thus it should not be introduced later. # indicates that the membrane containing it is not used effectively in the computation.

(2) $[w]^-_{(xy\to u,a)} \to w, [\overline{w}_1]^-_{(xy\to u,a)} \to \overline{w}_1$,
$[\#]^-_{(xy\to u,a)} \to \$$.
These rules pass the information computed by rules in (1) to the membrane labelled with $(xy \to u, b)$. \$ is a dummy object not used later.

(3) $[C^{\exists}_{xy\to u}]^0_{(xy\to u,b)} \to [C^{\exists}_{xy\to u}]^+_{(xy\to u,b)}[C^{\exists}_{xy\to u}]^+_{(xy\to u,b)}$,
$[C^{\nexists}_{xy\to u}]^0_{(xy\to u,b)} \to [C^{\nexists}_{xy\to u}]^-_{(xy\to u,b)}[C^{\nexists}_{xy\to u}]^-_{(xy\to u,b)}$.
These rules decide if object $C^{\exists}_{xy\to u}$ or $C^{\nexists}_{xy\to u}$ exists in the system. The result is stored in the polarizations of the new membranes.

(4) $[w]^+_{(xy\to u,b)} \to w, [w]^-_{(xy\to u,b)} \to \overline{w}_2$,
$[\overline{w}_1]^+_{(xy\to u,b)} \to \overline{w}_1, [\overline{w}_1]^-_{(xy\to u,b)} \to \overline{w}_1$.
These rules introduce objects that will control the computation according to the information computed by the previous subsets of rules. For example, if w and $C^{\nexists}_{xy\to u}$ is present in the inner membrane, then $\overline{w}_2$ is introduced. In this case $V_u^{\exists}$ will not be introduced at the end of the computation in this layer (see rules in (8)).

(5) $[V_y^{\exists}]^0_{(xy\to u,c)} \to [V_y^{\exists}]^+_{(xy\to u,c)}[V_y^{\exists}]^+_{(xy\to u,c)}$,
$[V_y^{\nexists}]^0_{(xy\to u,c)} \to [V_y^{\nexists}]^-_{(xy\to u,c)}[V_y^{\nexists}]^-_{(xy\to u,c)}$.
These rules decide if object $V_y^{\exists}$ or $V_y^{\nexists}$ exists in the system. The result is stored in the polarizations of the new membranes.

(6) $[w]^+_{(xy\to u,c)} \to w, [w]^-_{(xy\to u,c)} \to \overline{w}_2$,
$[\overline{w}_1]^+_{(xy\to u,c)} \to \overline{w}_1, [\overline{w}_1]^-_{(xy\to u,c)} \to \overline{w}_1$,
$[\overline{w}_2]^+_{(xy\to u,c)} \to \overline{w}_2, [\overline{w}_2]^-_{(xy\to u,c)} \to \overline{w}_2$.
These rules introduce objects that will control the computation according to the information computed by the previous subset of rules.

(7) $[V_x^{\exists}]^0_{(xy\to u,d)} \to [V_x^{\exists}]^+_{(xy\to u,d)}[V_x^{\exists}]^+_{(xy\to u,d)}$,
$[V_x^{\nexists}]^0_{(xy\to u,d)} \to [V_x^{\nexists}]^-_{(xy\to u,d)}[V_x^{\nexists}]^-_{(xy\to u,d)}$.

These rules decide if object $V_x^{\exists}$ or $V_x^{\nexists}$ exists in the system. The result is stored in the polarizations of the new membranes.

(8) $[w]^+_{(xy\to u,d)} \to V_u^{\exists}, [w]^-_{(xy\to u,d)} \to V_u^{\nexists}$,
$[\overline{w}_1]^+_{(xy\to u,d)} \to V_u^{\exists}, [\overline{w}_1]^-_{(xy\to u,d)} \to V_u^{\exists}$
$[\overline{w}_2]^+_{(xy\to u,d)} \to V_u^{\nexists}, [\overline{w}_2]^-_{(xy\to u,d)} \to V_u^{\nexists}$.
These rules are used to handle the different cases of possible computations in a layer. For example, w indicates that at the beginning of the computation in a layer the system contained objects $V_u^{\nexists}$, $C^{\exists}_{xy\to u}$, and $V_y^{\exists}$.

(9) $[O^e]^0_{d_O} \to \$, [w]^0_{d_w} \to \$, [\overline{w}_i]^0_{d_w} \to \$$
$(O \in V(n) \cup C(n), e \in \{\exists, \nexists\}, i \in [2])$.
These rules are used to remove certain objects from the system.

(10) $[V_\downarrow^{\nexists}]^0_s \to [no]^-_s[\$]^-_s$, $[V_\downarrow^{\exists}]^0_s \to [yes]^-_s[\$]^-_s$, $[\kappa]^-_s \to \kappa \quad (\kappa \in \{yes, no\})$.
These rules are used to send out the computed answer to the environment.

Correctness, running time, and $(\mathbf{L}, \mathbf{L})$*-uniformity.* First we observe that during the computation of $\Pi_2(n)$ the following holds.

1. The membrane structure has the form $[\ldots[M]_{h_1}\ldots]_{h_k}$ $(h_1, \ldots, h_k \in H(n))$, where M is either a membrane or it is of the form $[\ ]_{g_1}[\ ]_{g_2}$ $(g_1, g_2 \in H(n))$, and
2. objects occur only in the innermost membranes.

The correctness of the system follows from the following lemma.

Lemma 1. *Let* $\mathcal{C} = xy \to u$ $(x, y \in Var_n, u \in Var_n \cup \{\downarrow\})$ *and consider the layer* $l_{\mathcal{C}} = D_{\mathcal{C}}[M_{\mathcal{C}}]$. *Assume that, for every* $O \in C(n) \cup V(n)$, *either one copy of* $O^{\exists}$ *or one copy of* $O^{\nexists}$ *occurs in* $l_{\mathcal{C}}$. *Let* O *be an object in* $l_{\mathcal{C}}$. *Depending on* O *the following holds:*

1. *If* $O \in \Sigma(n) - \{V_u^{\nexists}\}$, *then after dissolving all the membranes in* $l_{\mathcal{C}}$, $\Pi_2(n)$ *contains exactly one copy of* O.
2. *If* $O = V_u^{\nexists}$ *and* $l_{\mathcal{C}}$ *contains all of the objects* $C_{\mathcal{C}}^{\exists}$, $V_x^{\exists}$, *and* $V_y^{\exists}$, *then after dissolving all the membranes in* $l_{\mathcal{C}}$, $\Pi_2(n)$ *contains no* $V_u^{\nexists}$ *and exactly one copy of* $V_u^{\exists}$.
3. *If* $O = V_u^{\nexists}$ *and* l_c *contains* $C_{\mathcal{C}}^{\nexists}$, $V_x^{\nexists}$, *or* $V_y^{\nexists}$, *then after the work in* $l_{\mathcal{C}}$ $\Pi_2(n)$ *contains exactly one copy of* $V_u^{\nexists}$.

Proof. By assumption, $l_{\mathcal{C}}$ contains exactly one copy of O. Then Statement 1 can be seen by distinguishing the following two sub-cases:
Case 1. $O \neq V_u^{\exists}$. Then during the work in $M_{\mathcal{C}}$, O is duplicated by the corresponding rules rules in (1), (3), (5), and (7), and the other rules are not applied to O in $M_{\mathcal{C}}$. This yields sixteen copies of O in $D_{\mathcal{C}}$. Out of these copies fifteen ones are removed during the computation in $D_{\mathcal{C}}$.

Case 2. $O = V_u^{\exists}$. Then the second rule in (1) removes first $V_u^{\exists}$ and introduces one copy of $\overline{w}_1$. After this, membrane $(\mathcal{C}, a)$ is dissolved using rules in (2). In the next two steps, $\overline{w}_1$ is duplicated first due the division of membrane $(\mathcal{C}, b)$ by rules in (3), then the yielded membranes are dissolved by rules in (4). Thus, at this point of the computation two copies of $\overline{w}_1$ are in membrane d_w. However, in the next step one copy is removed due to the corresponding rule in (9). After this, membrane $(\mathcal{C}, c)$ is divided (rules in (5)) and the new membranes are dissolved (rules in (6)). At this point, two copies of $\overline{w}_1$ are in membrane d_w, and one copy is removed by the corresponding rule in (9). Finally, $\overline{w}_1$ is duplicated by rules in (7), and then the two copies of $\overline{w}_1$ introduce two copies of $V_u^{\exists}$. During the dissolution of membranes in $D_{\mathcal{C}}$ one copy of $V_u^{\exists}$ is removed which proves the statement.

Statement 2 can be seen as follows. The computation starts with removing the object $V_u^{\nexists}$ and introducing one w (first rule in (1)). Then the new membranes with label $(\mathcal{C}, a)$ are dissolved by the corresponding rules in (2). In membrane $(\mathcal{C}, b)$ the first rule of (3) is applied and thus w is duplicated. At this point membranes with label $(\mathcal{C}, b)$ have positive charges, thus only the first rule in (4) can be applied. After this the corresponding rule in (9) removes one copy of w. During the next step the first rule in (5) is applied, and then only the first rule in (6) can be used. Again, one copy of w is removed by the corresponding rule in (9). Then the first rule in (7) divides membrane $(\mathcal{C}, d)$, w is again duplicated, and by the first rule in (8) each w introduces one copy of $V_u^{\exists}$. During the work in $D_{\mathcal{C}}$, one copy of $V_u^{\exists}$ is removed.

The system has several different computations in the case of Statement 3. We discuss here only one of them, the remaining ones can be treated similarly. Assume for example that $l_{\mathcal{C}}$ contains $C_{\mathcal{C}}^{\exists}$ and $V_y^{\nexists}$. Then the computation goes in the same way as in the case of Statement 2 until the application of the corresponding dissolution rules in (4). But now the second rule in (5) is applied, and thus, in the next step, only the second rule in (6) can be applied. Therefore here two copies of $\overline{w}_2$ are introduced. Then the computation continues similarly as in Case 2 in the proof of Statement 1. However here, when rules from (8) are applied the system has two copies of $\overline{w}_2$, and thus two copies of $V_u^{\nexists}$ are introduced by the fifth and sixth rules in (8). One of these copies is eliminated during the work in $D_{\mathcal{C}}$.

Clearly, the initial configuration of $\Pi_2(n)$ satisfies the conditions of Lemma 1. Then, by the iterated application of Lemma 1, we get that the computation in a layer $l_{\mathcal{C}}$ in a block r_k ($k \in [n]$) corresponds to the step 5 of Algorithm H3SN when $i = k$ and $c_j = \mathcal{C}$. Therefore, the the whole computation of $\Pi_2(n)$ corresponds to the complete work of this algorithm. This justifies the correctness of $\Pi_2(n)$.

Since $\Pi_2(n)$ has polynomial number of membranes in layer $l_{\mathcal{C}}$, and in $l_{\mathcal{C}}$ the number of the applied division rules is constant, we have that dissolving all the membranes in $l_{\mathcal{C}}$ takes polynomial time. As in the initial configuration there are n blocks and each block has polynomial number of layers, it follows that the running time of $\Pi_2(n)$ is also polynomial.

Finally, it can be seen that all the ingredients of $\Pi_2(n)$ can be enumerated by a logarithmic space Turing machine. This, using the **P**-completeness of HORN3SATNORM yields the following theorem.

Theorem 2. $\mathbf{P} \subseteq (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}_{+e,+d}}$.

3.2 Simulating Turing Machines

Here we show that, for every polynomial time Turing machine M, an $(\mathbf{L}, \mathbf{L})$-uniform family $\mathbf{\Pi_3}$ of polarizationless recognizer P systems can be constructed such that the members of $\mathbf{\Pi_3}$ can simulate the work of M efficiently using only dissolution and unit rules.

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ be an $f(n)$-time Turing machine, for some polynomial $f(n)$. Notice that M can use at most $f(n)$ cells of its tape during its computations. Let $k = |Q|$ and $m = |\Gamma|$. Assume that $Q = \{s_1, \ldots, s_k\}$, where $s_1 = q_0, s_{k-1} = q_a$ and $s_k = q_r$. Likewise, assume that $\Gamma = \{X_1, \ldots, X_m\}$, where $X_m = \sqcup$. The idea of the simulation is the following. The initial membrane structure μ is a composition of $f(n)+1$ blocks (see Fig. 4). The input membrane is the innermost membrane. During the simulation of the tth step of M, the objects in the innermost membrane will dissolve all the membranes in the tth block as follows. Assume that after $t-1$ steps M is in state s_i ($i \in [k-2]$), the position of the head is p, and the head scans X_j. Then the innermost membrane of the tth block contains an object O that represents s_i and p, and another object O' representing X_j on the pth position of the tape. The blocks are composed of $k \cdot m \cdot f(n)$ membranes (that is, in every block, for every state–tape symbol–position triple there is a corresponding membrane). During the simulation of the tth step of M, O dissolves all the membranes that correspond to a state $s_{i'}$ with $i' < i$ or a position $p' < p$. Meanwhile O' evolves using a counter and at the appropriate time step it starts to dissolve all the membranes corresponding to s_i, p, and tape symbol $X_{j'}$ with $j' < j$. After this the simulation of one step of M is performed using the value $\delta(s_i, X_j)$. Then the remaining membranes in the tth block are dissolved, and the system continues with the simulation of the next step of M.

Construction of the P system. The uniform family of P systems that will perform the above described simulation is defined as follows. Let $w = a_1 \ldots a_n$ be an input of M ($a_1, \ldots, a_n \in \Sigma$) and $N = f(n) \cdot k \cdot m$. Let $cod(w)$ be a multiset over the alphabet

$$\Sigma(n) = \{(X_j, p, t)^{(c)}, (s_i, p, t)^{(c')} \mid j \in [m], i \in [k], p \in [f(n)], t \in [0, f(n)], c \in [0, N], c' \in [0, N+m]\}$$

defined as follows: $cod(w) = \{(a_1, 1, 0)^{(0)}, \ldots, (a_n, n, 0)^{(0)}\} \cup \{(\sqcup, n+1, 0)^{(0)}, \ldots, (\sqcup, f(n), 0)^{(0)}\} \cup (s_1, 1, 0)^{(0)}$. Intuitively, an object $(X_j, p, t)^{(c)}$ in $\Sigma(n)$ represents the fact that after t steps M has X_j on the pth position of its tape. We call these objects *position objects*. Similarly, an object $(s_i, p, t)^{(c')}$ represents the fact that

after t steps M is in state s_i and the head points to the pth position of the tape. We call these objects *state objects*. The indexes c, c' are counters used for technical reasons. It can be seen that $cod \in \mathbf{L}$.

Let $\mathbf{\Pi_3} = \{\Pi_3(n)\}_{n\in\mathbb{N}}$ be a uniform family of P systems, where $\Pi_3(n) = (\Gamma(n), H(n), \mu(n), W(n), R(n))$ is defined as follows:

- $\Gamma(n) = \Sigma(n) \cup \{yes, no\}$.
- $H(n) = \{(s_i, p, X_j, t) \mid i \in [k], p, t \in [f(n)], j \in [m]\}$.
  Intuitively, a label (s_i, p, X_j, t) corresponds to the following configuration of M after t steps on w: the current state is s_i, the position of the head is p, and the scanned symbol is X_j. We will often call s_i, p, and t the *state*, *position*, and *time* labels of the corresponding membrane, respectively.
- $\mu(n)$ is a composition $S[r_{f(n)}[\dots[r_1]]]$ of blocks, where $S = [\ ]_{skin}$, and a block r_t ($t \in [f(n)]$) is a composition of layers defined as follows. For every $i \in [k]$ and $p \in [f(n)]$, let $l_{s_i,p,t} = [\dots[\ ]_{(s_i,p,X_1,t)}\dots]_{(s_i,p,X_m,t)}$, and let $r_t = l_{s_k,f(n),t}[\dots[l_{s_k,1,t}[\dots[l_{s_1,f(n),t}[\dots[l_{s_1,1,t}]\dots]]\dots]]\dots]$.

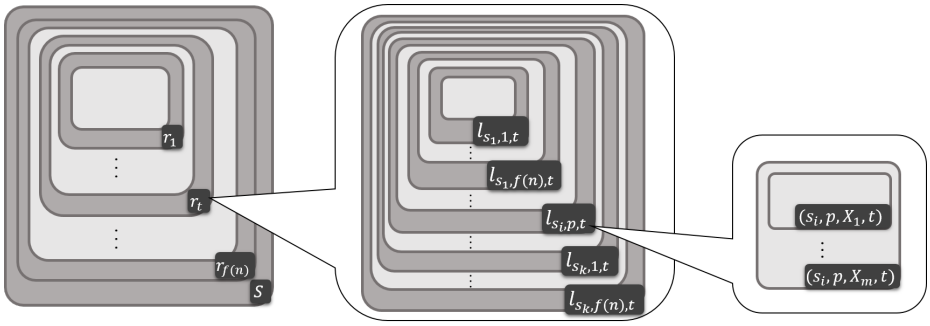


Fig. 4. The initial membrane structure of $\Pi_3(n)$

- The input membrane is the innermost membrane in $\mu(n)$.
- $W(n)$ is a sequence of empty initial multisets.
- R consists of the following sets of rules:
  (1) $[(s_i, p, t)^{(0)}]_{(s_{i'}, p', X_j, t+1)} \to (s_i, p, t)^{(0)}$
  ($j \in [m]$, $i \in [k-2]$, $i' \in [k]$, $p, p' \in [f(n)]$, $t \in [0, f(n)-1]$, and $i' < i$ or $p' < p$).
  These rules are used to find the first such membrane whose state and position labels correspond to the state and position stored in the state object.

  (2) $[(X_j, p, t)^{(c)} \to (X_j, p, t)^{(c+1)}]_{(s_i, p', X_{j'}, t+1)}$
  ($j, j' \in [m]$, $i \in [k]$, $p, p' \in [f(n)]$, $t \in [0, f(n)-1]$, $c \in [0, N-1]$).
  These rules are used to increment the counter c in the position objects. When this counter equals to N, the system can start to use rules in (3).

(3) $[(X_j,p,t)^{(N)}]_{(s_i,p,X_l,t+1)} \to (X_j,p,t)^{(N)}$,
$[(X_j,p,t)^{(N)} \to (X_{j'},p,t+1)^{(0)}]_{(s_i,p,X_j,t+1)}$,
$[(X_j,p',t)^{(N)} \to (X_j,p',t+1)^{(0)}]_{(s_i,p,X_1,t+1)}$
$(j,l \in [m],\ l<j,\ i \in [k-2],\ p,p' \in [f(n)],\ p \neq p',\ t \in [0,f(n)-1]$, and $X_{j'} = \mathrm{proj}_2(\delta(s_i,X_j)))$.
If the position stored in an object $(X_j,p,t)^{(N)}$ corresponds to the position label of the current membrane, then this object starts to dissolve the membranes until a membrane whose label stores X_j is found. When this membrane is found, $(X_j,p,t)^{(N)}$ evolves according to the value of $\delta(s_i,X_j)$, its counter is reset, and its component t is incremented. Those position objects that store a different position than the position label of the current membrane evolve immediately such that their counter is reset and their component t is incremented. Notice that after performing the computations by these rules, the position objects have no impact on the computation in block r_{t+1}.

(4) $[(s_i,p,t)^{(c)} \to (s_i,p,t)^{(c+1)}]_{(s_i,p,X_l,t+1)}$,
$[(s_i,p,t)^{(N+m)} \to (s_{i'},p',t+1)^{(0)}]_{(s_i,p,X_l,t+1)}$
$(i \in [k-2], i' \in [k],\ p \in [f(n)],\ t \in [0,f(n)-1],\ c \in [N+m-1],\ l \in [m]$, $s_{i'} = \mathrm{proj}_1(\delta(s_i,X_l))$, $p' = \max\{p + \mathrm{proj}_3(\delta(s_i,X_l)),1\})$.
The counter of the state object is incremented using the first rule. Until the counter reaches $N+m$, the appropriate position object can find the corresponding membrane using rules in (3). Then the state object evolves according to the value of the transition function of M. Moreover, its counter is reset and its component t is incremented.

(5) $[(s_i,p,t+1)^{(0)}]_{(s_{i'},p',X_j,t+1)} \to (s_i,p,t+1)^{(0)}$
$(i \in [k-2], i' \in [k], p,p' \in [f(n)], j \in [m], t \in [0,f(n)-1])$.
After simulating a step of M using rules in (1)-(4), the remaining membranes in block r_{t+1} are dissolved by these rules.

(6) $[(s_{k-1},p,t)^{(0)} \to yes]_h$, $[(s_k,p,t)^{(0)} \to no]_h$,
$[yes]_{h'} \to yes$, and $[no]_{h'} \to no$
$(p,t \in [f(n)], h \in H(n), h' \in H(n) - \{skin\})$.
These rules are used to produce the answer of $\Pi_3(n)$ according to which halting state is reached by M on the input.

Correctness, running time, and $(\mathbf{L},\mathbf{L})$*-uniformity.* Let $w = a_1 \dots a_n$ be an input of M ($a_1,\dots,a_n \in \Sigma$). We show that $\Pi_3(n)$ produces *yes* started with $cod(w)$ in its input membrane if and only if $w \in L(M)$. The work of $\Pi_3(n)$ can be described as follows. Initially, the object $(s_1,1,0)^{(0)}$ (representing that M starts its work in its initial state and the head is positioned to the first letter of the input) is in the innermost membrane of block r_1. Now assume that $\Pi_3(n)$ has already simulated t steps of M, that is, the innermost membrane of $\Pi_3(n)$ is the most deeply nested membrane of block r_{t+1}, and this membrane contains an object $(s_i,p,t)^{(0)}$, for some $i \in [k]$ and $p \in [f(n)]$ (see Fig. 5). If $i \in [k-1,k]$, i.e., M has reached one of its halting states, then $\Pi_3(n)$, using rules in (6) computes

the answer of the system *yes* or *no* accordingly. Otherwise, rules from (1) are applied until a membrane with label $(s_i, p, X_1, t+1)$ is reached. Meanwhile, the counter c in position objects is incremented using rules in (2). By the time this counter becomes N, the corresponding membrane is reached by the rules in (1). Now those position objects that store different positions than p evolve by the third rule in (3) to such objects that will be used next time only in the next block r_{t+2} (i.e., in the simulation of the next step of M).

Concerning the position object storing p, assume that this object is $(X_j, p, t)^{(N)}$. Then $(X_j, p, t)^{(N)}$ is used to find that membrane in layer $l_{s_i,p,t+1}$ whose label contains X_j. When this membrane is found, $(X_j, p, t)^{(N)}$ evolves according to the transition function of M. Moreover, its counter is reset and its time component is incremented. Thus this object is not used any more in this block.

Meanwhile, rules in (4) are used to increment the counter c of $(s_i, p, t)^{(c)}$. By the time this counter becomes $N + m$, the position object $(X_j, p, t)^{(N)}$ has reached the membrane it searched for. Now the second rule in (4) is used to produce object $(s_{i'}, p', t+1)^{(0)}$ where $s_{i'}$ and p' are calculated according to the transition function of M. Finally, $(s_{i'}, p', t+1)^{(0)}$ is used to dissolve the remaining membranes of r_{t+1}. If this is done, the system is ready to simulate the next step of M. With this we have seen that $\Pi_3(n)$ simulates correctly the computation of M on w.

It can be seen that dissolving a block in the membrane structure takes $O(N)$ steps and $N = O(f(n))$. Moreover, $\Pi_3(n)$ has $f(n)$ blocks. Thus the running time of the system is $O(f^2(n))$, that is, polynomial in n. The $(\mathbf{L}, \mathbf{L})$-uniformity of $\mathbf{\Pi_3}$ follows from the observation that the size of $\Pi_3(n)$ is also polynomial in n. Thus we have the following result.

Theorem 3. $\mathbf{P} \subseteq (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}^0_{+u,+d}}$.

As we have observed on page 151, our solution of HORN3SATNORM by P systems of type $\mathcal{AM}_{+e,+d}$ is such that the number of membranes occurring on the same level in the membrane structure is at most two during the whole computation of the system. Let $k \geq 1$. We say that a P system Π is k-bounded, if the number of membranes occurring on the same level in the membrane structure is at most k in every configuration of each computation of Π. For a type $\mathcal{F}$ of P systems, denote $(\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{F}_{\leq k}}$ the set of those problems that can be decided in polynomial time by such $(\mathbf{L}, \mathbf{L})$-uniform families of P systems of type $\mathcal{F}$ which have k-bounded members only. Denote $\mathcal{AM}_{-e}$ those P systems with active membranes that do not employ membrane division rules. It was shown in [31] that $\mathbf{PMC}_{\mathcal{AM}_{-e}} \subseteq \mathbf{P}$. It can be seen by the generalization of the proof of this result that $(\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}_{\leq 2}} \subseteq \mathbf{P}$ also holds. Using these and the results obtained in the paper we can give the following new characterizations of $\mathbf{P}$.

Corollary 1. $\mathbf{P} = (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}_{+out}} = (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}_{+e,+d,\leq 2}} = (\mathbf{L}, \mathbf{L}) - \mathbf{PMC}_{\mathcal{AM}^0_{+u,+d}}$.

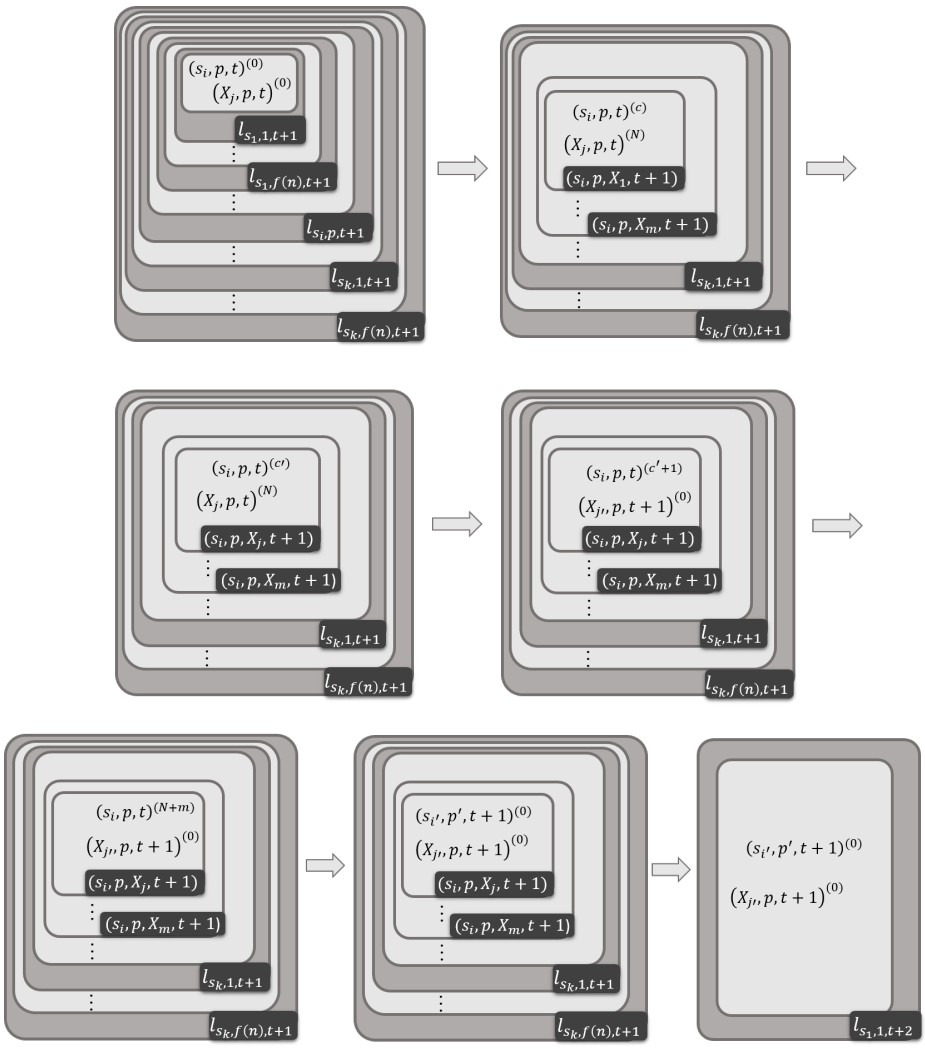


Fig. 5. Simulating the $(t+1)$th step of a TM by $\Pi_3(n)$ (c and c' are appropriate counter in $[N+m]$)

4 Conclusions

In this paper we have shown that uniform families of the following restricted variants of P systems with active membranes can solve all problems in **P**: (1) P systems where only send-out communication rules are used, (2) P systems where only elementary membrane division and dissolution rules are used, and (3) polarizationless P systems where only dissolution and unit rules are used. Using the obtained results concerning variants (1) and (3), and known results about the upper bound on the power of these variants we could give new characterizations of **P** in terms of Membrane Computing techniques.

It remained an open question if the variant (2) could solve problems outside of **P**. It is known that without polarizations of the membranes this is not possible [30]. It is also an open question if these systems can solve all problems in **P** when polarizations of the membranes are not allowed. Nevertheless, we could give another characterization of **P** using variant (2) where we made a simple semantic restriction on the computations of this variant.

References

1. Alhazov, A., Martín-Vide, C., Pan, L.: Solving a PSPACE-Complete Problem by P Systems with Restricted Active Membranes. Fundamenta Informaticae **58** (2003) 67–77
2. Alhazov, A., Pan, L., Păun, Gh.: Trading polarizations for labels in P systems with active membranes. Acta Inf. **41**(2-3) (2004) 111–144
3. Gazdag, Z.,: Solving SAT by P Systems with Active Membranes in Linear Time in the Number of Variables. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing: 14th International Conference, LNCS vol. 8340 (2014) 189–205
4. Gazdag, Z., 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, G. (eds.) Membrane Computing: 13th International Conference, LNCS vol. 7762 (2013) 195–207
5. Gazdag, Z., Gutiérrez-Naranjo, M.A.: Solving the ST-connectivity problem with pure membrane computing techniques. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing: 15th International Conference, LNCS vol. 8961 (2014) 215–228
6. Gazdag, Z., Kolonits, G., Gutiérrez-Naranjo, M.A.: Simulating Turing machines with polarizationless P systems with active membranes. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing: 15th International Conference, LNCS vol. 8961 (2014) 229–240
7. Gensler, H.J.: Introduction to Logic, Routledge, London (2002)
8. Gutierrez-Naranjo, M.A., Perez-Jimenez, M.J., Riscos-Núñez, A., Romero-Campero, F.J.: On the Power of Dissolution in P Systems with Active Membranes. In: Freund, R., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing: 6th International Workshop, LNCS vol. 3850 (2006) 224–240
9. Kolonits, G.: A Solution of Horn-SAT with P Systems Using Antimatter. In: Membrane Computing: 16th International Conference, LNCS vol. 9504 (2015) 236–250

10. Krishna, S.N., Rama, R.: A variant of P systems with active membranes: Solving NP-complete problems. Romanian J. of Information Science and Technology, 2, 4 (1999) 357–367
11. Murphy, N.: Uniformity conditions for membrane systems: uncovering complexity below P. Ph.D. thesis, National University of Ireland, Maynooth (2010)
12. Murphy, N., Woods, D.: Active Membrane Systems Without Charges and Using Only Symmetric Elementary Division Characterise P. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing: 8th International Workshop, LNCS vol. 4860 (2007) 367–384
13. Murphy, N., Woods, D.: A Characterisation of NL Using Membrane Systems without Charges and Dissolution. In: Calude, C.S., da Costa, J.F.G., Freund, R., Oswald, M., Rozenberg, G. (eds.) Unconventional Computing: 7th International Conference, LNCS vol. 5204 (2008) 164–176
14. Murphy, N., Woods, D.: On acceptance conditions for membrane systems: characterisations of **L** and **NL**. In Neary, T., Woods, D., Seda, T., Murphy, N., eds.: Proceedings International Workshop on The Complexity of Simple Programs, Cork, Ireland, Volume 1 of Electronic Proceedings in Theoretical Computer Science., Open Publishing Association (2009) 172–184
15. Murphy, N., Woods, D.: The computational power of membrane systems under tight uniformity conditions. Natural Computing **10**(1) (2011) 613–632
16. Murphy, N., Woods, D.: Uniformity is weaker than semi-uniformity for some membrane systems. Fundam. Inf. **134**(1-2) (2014) 129–152
17. Pan, L., Alhazov, A., Ishdorj, T.-O.: Further remarks on P systems with active membranes, separation, merging, and release rules. Soft Computing **9**(9) (2004) 686–690
18. Pan, L., Ishdorj, T.-O.: P systems with active membranes and separation rules. Journal of Universal Computer Science 10(5) (2004) 630–649
19. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley Publishing Company, Inc. (1994)
20. Păun, Gh.: P Systems with Active Membranes: Attacking NP-Complete Problems. Journal of Automata, Languages and Combinatorics **6**(1) (2001) 75–90
21. Păun, Gh.: Further twenty six open problems in membrane computing. In: Third Brainstorming Week on Membrane Computing. Fénix Editora, Sevilla (2005) 249–262
22. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Oxford, England (2010)
23. Pérez-Jiménez, M.J., Romero-Campero, F.J.: Trading Polarization for Bi-stable Catalysts in P Systems with Active Membranes. In: Mauri, G., Păun, Gh., Pérez-Jiménez, M.J., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing: 5th International Workshop, LNCS vol. 3365 (2005) 373–388
24. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: A polynomial complexity class in P systems using membrane division. In: Csuhaj-Varjú, E., Kintala, C., Wotschke, D., Vaszil, G. (eds.) Proceeding of the 5th Workshop on Descriptional Complexity of Formal Systems. DCFS 2003 (2003) 284–294
25. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. Natural Computing **2**(3) (2003) 265–285
26. Pérez-Jiménez, M.J., Romero-Jiménez, Á., Sancho-Caparrini, F.: A polynomial complexity class in P systems using membrane division. Journal of Automata, Languages and Combinatorics **11**(4) (2006) 423–434

27. Salomaa, A.: Formal Languages. Academic Press, New York, London (1973)
28. Sipser, M.: Introduction to the Theory of Computation. 3rd edn. Cengage Learning (2012)
29. Sosík, P.: The computational power of cell division in P systems. Nat. Comput. **2**(3) (2003) 287–298
30. Woods, D., Murphy, N., Pérez-Jiménez, M.J., Riscos-Núñez, A.: Membrane Dissolution and Division in P. In: Calude, C.S., da Costa, J.F.G., Dershowitz, N., Freire, E., Rozenberg, G. (eds.) Unconventional Computation: 8th International Conference, LNCS vol. 5715 (2009) 262–276
31. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-Complete Problems Using P Systems with Active Membranes. In: Unconventional Models of Computation, UMC'2K: Proceedings of the Second International Conference on Unconventional Models of Computation. Springer London, London (2001) 289–301

Kernel P Systems Modelling, Testing and Verification - Sorting Case Study

Marian Gheorghe[1], Rodica Ceterchi[2], Florentin Ipate[2,3] and Savas Konur[1]

[1] School of Electrical Engineering and Computer Science, University of Bradford
Bradford BD7 1DP, UK
{m.gheorghe, s.konur}@bradford.ac.uk
[2] Department of Computer Science, University of Bucharest
Str. Academiei nr. 14, 010014, Bucharest, Romania
florentin.ipate@ifsoft.ro, rceterchi@gmail.com
[3] Department of Computer Science, University of Piteşti
Str Targul din Vale, nr 1, 110040, Argeş, Romania

Abstract. A kernel P system (kP system, for short) integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and, consequently, it provides a framework for analyzing these models. In this paper, we illustrate the modelling capacity of kernel P systems by providing a number of kP system models for sorting algorithms. Furthermore, the problem of testing systems modelled as kP systems is also discussed and a test generation method based on automata is proposed. We also demonstrate how formal verification can be used to validate that the given models work as desired.

1 Introduction

Membrane systems were introduced in [28] as a new natural computing paradigm inspired by the structure and distribution of the compartments of living cells, as well as by the main biochemical interactions occurring within compartments and at the inter-cellular level. They were later also called *P systems*. An account of the basic fundamental results can be found in [29] and a comprehensive description of the main research developments in this area is provided in [30]. The key challenges of the membrane systems area and a discussion on some future research directions, are available in a more recent survey paper [21].

In recent years, significant progress has been made in using P systems to model and simulate systems and problems from various areas. However, in order to facilitate the modelling, in many cases various features have been added in an ad-hoc manner to these classes of P systems. This has led to a multitude of P systems variants, without a coherent integrating view. There have been investigations aiming to produce unifying approaches for several variants of P systems [14, 13], looking mainly at the computational aspects, syntax and semantics. The concept of *kernel P systems (kP systems)* [18, 19] integrates in a coherent and elegant manner many of the P system features most successfully used for modelling various applications and provides a generic framework for specifying and analyzing these models. Furthermore, the expressive power of these systems has been illustrated by a number of representative case studies [20, 19]. The kP system model is supported by a modelling language, called kP-Lingua, capable of mapping a kP system specification into a machine readable representation. Furthermore, kP systems are supported by a software framework, kPWorkbench [22], which integrates a set of related simulation and verification tools and techniques.

Another complementary method to simulation and verification is testing, a major activity in the lifecycle of software systems. In practice, software products are almost always validated through

testing. Testing has been discussed for cell-like P systems and various strategies, such as rule coverage based and automata based techniques have been proposed [17, 25]. Until now, however, testing has not been discussed in the context of kP systems.

In this paper we further illustrate the modelling capacity of kP systems by providing a number of kP system models for sorting algorithms. Furthermore, the problem of testing and formally verifying systems modelled as kP systems is also discussed.

The key contributions of the paper are: (a) illustrate the modelling capability of kP systems by implementing a number of sorting methods - the method presented in Section 3.1 is an extension of the approach introduced in [18, 19] which includes a stopping condition, whereas the other sorting methods are new; (b) integrate the kP systems with a test generation method based on automata; and (c) formally verifying the sorting problems using the kPWorkbench environment.

2 kP Systems - Main Concepts and Definitions

We consider that standard P system concepts such as strings, multisets, rewriting rules, and computation are well-known and refer to [29] for their formal notations and precise definitions. The kP system concepts and definitions introduced below are from [18, 19]; some are slightly changed and this will be mentioned.

Definition 1. *T is a* set of compartment types*, $T = \{t_1, \ldots, t_s\}$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq s$, consists of a set of rules, R_i, and an execution strategy, σ_i, defined over $Lab(R_i)$, the labels of the rules of R_i.*

Remark 1. The compartments that appear in the definition of the kP systems will be instantiated from these compartment types. The types of rules and the execution strategies will be discussed later.

Definition 2. *A* kernel P (kP) system *of degree n is a tuple*

$$k\Pi = (A, \mu, C_1, \ldots, C_n, i_0),$$

where A is a finite set of elements called objects*; μ defines the* initial 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 t_i from T and an* initial multiset*, w_i over A; i_o is the* output compartment *where the result is obtained.*

2.1 kP System Rules

Each rule r may have a **guard** g and its generic form is $r\ \{g\}$. The guards are constructed using multisets over A, as operands, and relational and Boolean operators. Let us first introduce some notations.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w. Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, and a^n a multiset. We first introduce an *abstract relational expression.*

Definition 3. *If g is an* abstract relational expression *γa^n and w a multiset, then g applied to w denotes the* relational expression *$|w|_a \gamma n$; g is true with respect to the multiset w, if $|w|_a \gamma n$ is true.*

One can consider the Boolean operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction). An *abstract Boolean expression* is either an abstract relational operator or if g and h are abstract Boolean expressions then $\neg g$, $g \wedge h$ and $g \vee h$ are abstract Boolean expressions. The concept of a guard is a generalisation of the promotor and inhibitor concepts utilised by some variants of P systems.

Definition 4. *If g is an* abstract Boolean expression *containing g_i, $1 \leq i \leq n$, abstract relational expressions and w a multiset, then g applied to w, denoted gw, means the* Boolean expression *obtained from g by applying g_i, $1 \leq i \leq n$, to w. The guard g is true with respect to the multiset w, if the Boolean expression gw is true.*

Example 1. If g is the guard $\geq a^5 \wedge \geq b^3 \vee \neg > c$ and w a multiset, then gw is true if it has at least 5 a's and 3 b's or no more than one c.

If r $\{g\}$ denotes a rule then its guard, g, is defined by an abstract Boolean expression. A rule may or may not have a guard. A rule r $\{g\}$ is applicable to a multiset w when the left-hand side of r is contained into w and gw is true.

Definition 5. *A rule from a compartment $C_{l_i} = (t_{l_i}, w_{l_i})$ will have one of the following types:*

- *(a)* **rewriting and communication** *rule: $x \to y$ $\{g\}$,*
  where $x \in A^+$ and y has the form $y = (a_1, t_1) \ldots (a_h, t_h)$, $h \geq 0$, $a_j \in A$ and t_j indicates a compartment type from T – see Definition 2 – with instance compartments linked to the current compartment; t_j might also indicate the type of the current compartment, t_{l_i}, (in this case it is not present on the right hand side of the rule); if a link does not exist (i.e., there is no link between the two compartments 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 C_{l_i}, then one of them will be non-deterministically chosen;
- *(b)* **structure changing rules***; the following types of rules are considered:*
  - *(b1)* **membrane division** *rule: $[x]_{t_{l_i}} \to [y_1]_{t_{i_1}} \ldots [y_p]_{t_{i_p}}$ $\{g\}$,*
    where $x \in A^+$ and $y_j \in A^*$, $1 \leq j \leq p$; the compartment C_{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 C_{l_i}, but x, which will be replaced by y_j; all the links of C_{l_i} are inherited by each of the newly created compartments;
  - *(b2)* **membrane dissolution** *rule: $[]_{t_{l_i}} \to \lambda$ $\{g\}$;*
    the compartment C_{l_i} will be destroyed together with its links;
  - *(b3)* **link creation** *rule: $[x]_{t_{l_i}}; []_{t_{l_j}} \to [y]_{t_{l_i}} - []_{t_{l_j}}$ $\{g\}$;*
    the current compartment is linked to a compartment of type t_{l_j} and x is transformed into y; if more than one instance of the compartment type t_{l_j} not yet linked to t_{l_i} exist 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_{l_1};
  - *(b4)* **link destruction** *rule: $[x]_{t_{l_i}} - []_{t_{l_j}} \to [y]_{t_{l_i}}; []_{t_{l_j}}$ $\{g\}$;*
    is the opposite of link creation and means that the compartments are disconnected.

The membrane division is defined slightly differently here compared to [18, 19], where each y_j, $1 \leq j \leq p$, is composed of objects with target compartments.

2.2 kP System Execution Strategies

In kP systems the way in which rules are executed is defined for each compartment type t from T – see Definition 1 and Remark 1. As in Definition 1, $Lab(R)$ is the set of labels of the rules R.

Definition 6. *For a compartment type $t = (R, \sigma)$ from T and $r \in Lab(R)$, $r_1, \ldots, r_s \in Lab(R)$, the* execution strategy*, σ, is defined by the following*

- *$\sigma = \lambda$, means no rule from the current compartment will be executed;*
- *$\sigma = \{r\}$ – the rule r is executed once;*
- *$\sigma = \{r_1, \ldots, r_s\}$ – one of the rules labelled $r_1, \ldots, r_s$ will be chosen non-deterministically and executed; if none is applicable then none is executed; this is called* alternative *or* choice*;*

- $\sigma = \{r_1, \ldots, r_s\}^*$ – *the rules are applied an arbitrary number of times (*arbitrary parallelism*);*
- $\sigma = \{r_1, \ldots, r_s\}^\top$ – *the rules are executed according to* maximal parallelism *strategy;*
- $\sigma = \sigma_1 \& \ldots \& \sigma_s$*, means executing sequentially* $\sigma_1, \ldots, \sigma_s$*, where* σ_i*,* $1 \leq i \leq s$*, describes any of the above cases, namely* λ*, one rule, a choice, arbitrary parallelism or maximal parallelism; if one of* σ_i *fails to be executed then the rest is no longer executed;*
- *for any of the above* σ *strategy only one single structure changing rule is allowed.*

Arbitrary parallelism and maximal parallelism for rewriting and communication rules, as well as for structure changing rules (cell division, dissolution), are discussed in [30].

Remark 2. In certain cases the operator & will be ignored and the sequential execution will be denoted as $\sigma = \sigma_1 \ldots \sigma_s$.

3 Sorting with kP Systems

Sorting is a central topic in Computer Science (see [26]). A variety of approaches to sorting have been investigated, for different algorithms, and with different P system models. A first approach was [3], in which a BeadSort algorithm was implemented with tissue P systems. Another approach was [6], in which algorithms inspired from sorting networks were implemented using P systems with communication. Other papers ([1], [31]) use different types of P systems, and refine the sorting problem to sorting by ranking. A first overview of sorting algorithms implemented with P systems was [2]. A dynamic sorting algorithm was proposed in [7]. The bitonic sort was implemented with P systems [8], spiking neural P systems were used for sorting [10], other network algorithms were implemented using P systems [9]. Another overview of sorting algorithms implemented with P systems is provided by [11]. First implementations of sorting with kP systems were proposed in [18, 19].

The problem can be stated as follows: suppose we want to sort $x_1, \cdots, x_n$, $n \geq 1$, in ascending order, where x_i, $1 \leq i \leq n$, are positive integer values. Each such number, x_i, $1 \leq i \leq n$, will be represented as a multiset $a_i^{x_i}$, $1 \leq i \leq n$, where a_i is an object from a given set. In the next sections we will present two sorting algorithms using different representations of the sequence of positive integer numbers. More precisely, we start with an algorithm already studied in several other papers, [6, 2] for various types of P systems. Here we implement it using kP systems, by representing each element x_i by a^{x_i}, $1 \leq i \leq n$. The multisets a^{x_i}, $1 \leq i \leq n$, are stored in separate compartments, C_i, $1 \leq i \leq n$ (Section 3.1). In Section 3.2 these positive integer numbers are represented by $a_i^{x_i}$, $1 \leq i \leq n$, and stored in one compartment C_1; an additional one, C_2, is used for implementation purposes. In Section 3.3 is used again the representation $a_i^{x_i}$, $1 \leq i \leq n$, but a more complex structure of compartments is provided in order to maximise the parallel behaviour of the system implementing the sorting algorithm. The algorithm used in Section 3.1 and Section 3.2 makes comparisons of adjacent compartments by employing a two stage process. In the first stage all pairs "odd-even" are compared (C_{2i-1} with C_{2i}, $i \geq 1$) and in the second stage all pairs "even-odd" are involved (C_{2i} with C_{2i+1}, $i \geq 1$).

3.1 Sorting Using kP Systems with an Element per Compartment

The approach presented below follows [18, 19], but stopping conditions have been also considered and the sequence of numbers is obtained in ascending order.

Let us consider a kP system, $k\Pi_1$, having n compartments $C_i = (t_i, w_{i,0})$, where $t_i = (R_i, \sigma_i)$, $1 \leq i \leq n$, and a set of objects $A = \{a, b, c, p, p'\}$. In each compartment, C_i, the initial multiset, $w_{i,0}$, $1 \leq i \leq n$, includes the representation of the positive integer number x_i, i.e., a^{x_i}, the multiset $c^{2(n-1)}$ and the object p for all odd index values, when n is an even number, and for all odd index

values, but the last, when n is odd. The objects p stored initially in compartments indexed by odd values indicate that one starts with stage one, whereby "odd-even" compartment pairs are compared first. The multiset $c^{2(n-1)}$ will be used in a counting process, in each of the compartments, that will help stopping the algorithm when the sorting is complete.

Let us consider for $n = 6$ the sequence 3, 6, 9, 5, 7, 8. Then the initial multisets are: $w_{1,0} = a^3c^{10}p; w_{2,0} = a^6c^{10}; w_{3,0} = a^9c^{10}p; w_{4,0} = a^5c^{10}; w_{5,0} = a^7c^{10}p; w_{6,0} = a^8c^{10}$. As n is even, p appears in all compartments indexed by odd values, i.e., C_1, C_3, and C_5.

In each compartment C_i, t_i contains the following set of rules, denoted R_i, $1 \leq i \leq n$,
$r_{1,i} : a \to (b, i+1)$ $\{\geq p\}$, $i < n$;
$r_{2,i} : p \to p'$;
$r_{3,i} : p' \to (p, i+1)$, for i odd and $i < n$;
$r'_{3,i} : p' \to (p, i-1)$, for i even and $i > 1$;
$r_{4,i} : ab \to a(a, i-1)$, $i > 1$;
$r_{5,i} : b \to a$, $i > 1$;
$r : c \to \lambda$.

The rule r is used for implementing the counting process mentioned above. By using the two stage process of comparing "odd-even" pairs of compartments and then "even-odd" ones, one needs at most $n-1$ stages to complete the sorting. As will be explained below, each stage will involve two steps and consequently after $2(n-1)$ steps one expects to stop the sorting process.

In each compartment C_i, the execution strategy is given by $\sigma_i = \{r\}\{r_{1,i}, r_{2,i}, r_{3,i}, r_{4,i}\}^\top \{r_{5,i}\}^\top$, if i is odd; for even values of i, $r_{3,i}$ is replaced by $r'_{3,i}$. The execution strategy, σ_i, tells us that a sequence of three sets of rules are executed in each step. The first one indicates that one single rule is applied and then two sets of rules are used, each of them applied in a maximal parallel manner.

We assume that any two compartments, C_i, C_{i+1}, $1 \leq i < n$, are connected.

In the first step, of the "odd-even" stage, in every compartment one c is removed by applying $r : c \to \lambda$; then the only applicable rules are $r_{1,i}, r_{2,i}$ in all compartments indexed by an odd value. Given the presence of p in these compartments, rules $r_{1,i}$ move all objects a from each compartment with an odd index value, i, $i < n$, to the compartment C_{i+1} by transforming them into bs and rules $r_{2,i}$ transforming p into p'. In the next step, another c is removed from every compartment and rules $r_{3,i}, r_{4,i}, r_{5,i}$ are then applied. The rules $r_{3,i}$ are applied in compartments with an odd index value and $r_{4,i}$ are applied in compartments with an even index value, this means p' is moved as p from each C_i, i an odd value and $i < n$, to compartment C_{i+1} and every ab, in each C_j, j an even value and $j > 1$, is transformed into an a kept in the compartment and another a moved to C_{j-1}. At the end of the step, in each compartment C_j, j an even value and $j > 1$, and in accordance with the execution strategy, the remaining b objects, if any, are transformed into as. These two steps implement comparators between two adjacent compartments, in this case "odd-even" pairs. If a^{x_i} from C_i and $a^{x_{i+1}}$ from C_{i+1}, $i < n$, are such that $x_i > x_{i+1}$ then the multiset a^{x_i} is moved to C_{i+1} and $a^{x_{i+1}}$ to C_i. In the next step, the first of the second stage, ps appear in even compartments and the comparators are now acting between pairs of compartments C_i, C_{i+1}, where i is even and $i < n$.

Given that the algorithm must stop in maximum $2(n-1)$ steps, one can notice that in step $2(n-1)$ the counter, c, disappears, i.e., becomes λ, and the first rule from the execution strategy, r, is no longer applicable and then the next sets of rules are not executed either. Hence, the process stops with the multisets codifying the positive integer values in ascending order.

The table below presents the first four steps of the sorting process.

Compartments - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	$a^3c^{10}p$	a^6c^{10}	$a^9c^{10}p$	a^5c^{10}	$a^7c^{10}p$	a^8c^{10}
1	c^9p'	$a^6b^3c^9$	c^9p'	$a^5b^9c^9$	c^9p'	$a^8b^7c^9$
2	a^3c^8	a^6c^8p	a^5c^8	a^9c^8p	a^7c^8	a^8c^8p
3	a^3c^7	c^7p'	$a^5b^6c^7$	c^7p'	$a^7b^9c^7$	a^8c^7p'
4	a^3c^6p	a^5c^6	a^6c^6p	a^7c^6	a^9c^6p	a^8c^6

Now, one can state the result of the algorithm presented above and the number of steps involved.

Proposition 1. *The above algorithm sorts in ascending order a sequence of $n, n \geq 1$, positive integer numbers in $2(n-1)$ steps.*

3.2 Sorting Using kP Systems with Two Compartments

In this section we use a representation of the positive integer numbers $x_1, \cdots, x_n$ as multisets $a_1^{x_1}, \cdots, a_n^{x_n}$, where $a_1, \cdots, a_n$ are from a given set of distinct objects. We consider a kP system, $k\Pi_2$, with two compartments $C_j = (t_j, w_{j,0})$, $1 \leq j \leq 2$, which are linked and $A = \{a_1, \ldots, a_n, c\}$. The initial multisets are $w_{1,0} = a_1^{x_1} \cdots a_n^{x_n} c^{n-1}$ and $w_{2,0} = c^{n-1}$.

Finally, the kP system $k\Pi_2$ will lead to a multiset $a_1^{x_{i_1}} \cdots a_n^{x_{i_n}}$ in compartment C_1, such that $x_{i_1} \leq \cdots \leq x_{i_n}$.

In compartment C_1 the rules are
$R_{1,1} = \{a_i a_{i+1} \to (a_i, 2)(a_{i+1}, 2) \mid 1 \leq i < n \ \wedge \ i \text{ is odd}\}$;
$R_{2,1} = \{a_i \to (a_{i+1}, 2) \mid 1 \leq i < n \ \wedge \ i \text{ is odd}\}$;
$R_{3,1} = \{a_i \to (a_i, 2) \mid 1 \leq i \leq n\}$.

We also consider the rule $r : c \to \lambda$, like in the previous section.

Compartment C_2 has the rules
$R_{1,2} = \{a_i a_{i+1} \to (a_i, 1)(a_{i+1}, 1) \mid 1 \leq i < n \ \wedge \ i \text{ is even}\}$;
$R_{2,2} = \{a_i \to (a_{i+1}, 1) \mid 1 \leq i < n \ \wedge \ i \text{ is even}\}$;
$R_{3,2} = \{a_i \to (a_i, 1) \mid 1 \leq i \leq n\}$;
and the rule r defined above.

The execution strategies of these compartments are $\sigma_j = \{r\}Lab(R_{1,j})^\top Lab(R_{2,j})^\top Lab(R_{3,j})^\top$, $j = 1, 2$.

In compartment C_1 one implements "odd-even" comparison steps and in C_2 "even-odd" steps. The process starts with compartment C_1. The execution strategy in each compartment starts by decrementing the counter (using r), then the comparators are implemented by executing first $R_{1,j}$ and then $R_{2,j}$, $j = 1, 2$, both in maximally parallel manner. After that all the pairs a_i, a_{i+1} are sent to the other compartment and when $a_i^{x_i}$ and $a_{i+1}^{x_{i+1}}$ are such that $x_i > x_{i+1}$ then a_i is transformed into a_{i+1} and sent to the other compartment, i.e., a_i and a_{i+1} are swapped and sent to the other compartment. In the last part, are moved to the other compartment all the objects a_i, $1 \leq i \leq n$, that remained there after comparisons. This is the case when a pair a_i and a_{i+1} has its objects with their multiplicities, x_i and x_{i+1}, respectively, in the right order, i.e., $x_i \leq x_{i+1}$.

Clearly after at most $n-1$ steps the objects $a_1, \cdots, a_n$ have their multiplicities in the ascending order and the sorting process stops at step $n-1$ as r is no longer applicable and the execution strategy is not applicable any more.

Proposition 2. *The above algorithm sorts in ascending order a sequence of $n, n \geq 1$, positive integer numbers in $n-1$ steps.*

One can produce a similar implementation whereby the comparison of two neighbours is made more directly and with simpler rules, but with more complex guards.

In this case we extend the definition of a guard, by allowing θa^n to be of the form $\theta a^{f(z)}$, where $f(z)$ is a function over the multisets of objects returning a positive integer value. For the current multiset z, one can define, for instance, $f_b(z) = |z|_b$, Then a rule $a \to b\{> a^{f_b(\cdot)}\}$ is applicable to z if the guard is true, i.e., $|z|_a > |z|_b$.

The *extended definition of the guard* allows us to implement a comparator with simpler rules than in the previous case. We have the pair of integers x_1, x_2 represented as ${a_1}^{x_1}, {a_2}^{x_2}$. Consider the pair of guarded rewriting rules

$$a_1 \to a_2\{> {a_1}^{f_{a_2}(\cdot)}\} \text{ and } a_2 \to a_1\{< {a_2}^{f_{a_1}(\cdot)}\}$$

where $f_{a_2}(w) = |w|_{a_2}$ and $f_{a_1}(w) = |w|_{a_1}$. Then both guards codify the condition $x_1 > x_2$.

If $x_1 \leq x_2$ the rules are not applicable, while if $x_1 > x_2$, then the x_1 copies of a_1 are rewritten as a_2, and x_2 copies of a_2 are rewritten as a_1, interchanging the values and achieving eventually $x_1 \leq x_2$.

A kP system, $k\Pi_3$, is defined now for sorting the sequence of $n, n \geq 1$, positive integer numbers. It consists of two compartments C_1 and C_2 which are linked. They have the same initial multisets like $k\Pi_2$. The sets of rules associated with these compartments are

- R_1 consisting of the following subsets of rules (R_1 is responsible for "odd-even" stages):
  - $\{c \to \lambda\}$;
  - $R_{1,1} = \{a_i \to (a_{i+1}, 2)\{> a_i^{f_{a_{i+1}}(\cdot)}\} \mid i = 1, 3 \cdots \wedge\ i \leq n\}$;
  - $R_{2,1} = \{a_{i+1} \to (a_i, 2)\{< a_{i+1}^{f_{a_i}(\cdot)}\} \mid i = 1, 3 \cdots \wedge\ i \leq n\}$;
  - $R_{3,1} = \{a_i \to (a_i, 2) \mid i = 1, \cdots, n\}$.

The function f_{a_i} is defined $f_{a_i}(z) = |z|_{a_i}$, $1 \leq i \leq n$, for any multiset z.

Similarly, one defines R_2 in compartment C_2, which is used to implement the "even-odd" stage. The execution strategy is given by $\sigma_j = \{r\}Lab(R_{1,j} \cup R_{2,j})^\top Lab(R_{3,j})^\top, j = 1, 2$.

Proposition 3. *The above algorithm sorts in ascending order a sequence of $n, n \geq 1$, positive integer numbers in $n - 1$ steps.*

Remark 3. 1. The kP system $k\Pi_3$ has simpler rules (non-cooperative) than $k\Pi_2$ (cooperative rules), but the guards of the rules in $k\Pi_2$ are simpler than those belonging to $k\Pi_3$.
2. The number of rules applied in each step to interchange $a_i^{x_i}$ and $a_{i+1}^{x_{i+1}}$ is $max\{x_i, x_{i+1}\}$ for $k\Pi_2$ and $x_i + x_{i+1}$ for $k\Pi_3$. Hence, $k\Pi_2$ uses less rules than $k\Pi_3$ in each one of the $n - 1$ steps.

3.3 A kP System for Sorting in Constant Time

We present two sorting methods exploiting more the massive parallelism of the P systems in general and of the kP systems in particular. In both cases we consider the integers to be sorted $x_1, \cdots x_n$ distinct. In the first case one uses $n^2 + 2n$ compartments with specific initial multisets and a special arrangement of links amongst them. In the second case the n^2 compartments are replaced by n compartments with simpler initial multisets.

For the first sorting method we consider $n^2 + 2n$ compartments:

- $C_{i,j}$, $1 \leq i, j \leq n$, where each $C_{i,j}$ will be responsible for a comparison;
- C_i, $1 \leq i \leq 2n$, where each C_i, $1 \leq i \leq n$, will collect the results of comparing x_i to the rest; and C_i, $n + 1 \leq i \leq 2n$, will collect the sorted result.

The connections between compartments are given by the set of edges

$$E = \cup_{i=1}^{n} E_i$$

where

$$E_i = \{(C_i, C_{i,j}) \mid 1 \leq j \leq n\} \cup \{(C_i, C_k) \mid n+1 \leq k \leq 2n\}, 1 \leq i \leq n.$$

Each $C_{i,j}$, $1 \leq i, j \leq n$, will contain the initial multiset $w_{i,j,0} = {a_i}^{x_i} {a_j}^{x_j} a$ and the rules
$r'_{i,j} : a_i \to a_j F\{> {a_i}^{f_j(\cdot)}\}$; $r''_{i,j} : a_j \to a_i\{< {a_j}^{f_i(\cdot)}\}$; $r'''_{i,j} : a \to a'$;
$r_{i,j} : a' \to (F, i)\{\geq F\}$,
where $f_i(z) = |z|_{a_i}$ and $f_j(z) = |z|_{a_j}$.
The execution strategy is $\sigma_{i,j} = \{r'_{i,j}, r''_{i,j}, r'''_{i,j}, r_{i,j}\}^{\top}$.

Note that the rules $r'_{i,j}, r''_{i,j}$ implement a comparator between x_i and x_j, similar to the one of the previous section. The modified comparator produces also a symbol F (False) when $x_i > x_j$, signifying that $x_i \leq x_j$ is false. If the rewriting rules $r'_{i,j}, r''_{i,j}$ and $r'''_{i,j}$ have acted, then a single F will be sent to compartment C_i (by using the rule $r_{i,j}$).

In compartment $C_i, 1 \leq i \leq n$, we have the initial multiset $w_{i,0} = {a_i}^{x_i} a$ and the rules
$r'_i : a \to a'$; $r''_i : a' \to a''$;
$r_{i,0} : a_i \to (a, n+1)\{< F \wedge = a''\}$; $r_{i,k} : a_i \to (a, n+k+1)\{= F^k \wedge = a''\}$, $1 \leq k \leq n-1$.
The execution strategy is $\sigma_i = \{r'_i, r''_i, r_{i,0}, \cdots, r_{i,n-1}\}^{\top}$.
Compartments $C_i, n+1 \leq i \leq 2n$, are initially empty and contain no rules.

The functioning of the system is as follows. Initially, in compartments $C_{i,j}$, $1 \leq i, j \leq n$, the rules $r'_{i,j}$, $r''_{i,j}$, and $r'''_{i,j}$ act. If $x_i > x_j$ the values will be interchanged and some Fs will be produced (rules $r'_{i,j}$, $r''_{i,j}$ are used), signifying that $x_i \leq x_j$ is false. Also $r'''_{i,j}$ is used to transform a in a'. If at least one F is produced in $C_{i,j}$, then a single F will be sent to C_i, using rule $r_{i,j}$. In parallel, in each compartment C_i, $1 \leq i \leq n$, in the first two steps the rules r'_i and r''_i are applied.

After these two steps, no rules are applicable in $C_{i,j}$, $1 \leq i, j \leq n$, and in C_i, $1 \leq i \leq n$, the rules $r_{i,k}$, $0 \leq k \leq n-1$, might be applicable, depending on the number of Fs collected.The number of Fs tells us how many comparisons $x_i \leq x_j$, $1 \leq j \leq n$, are false. If we have k such Fs in C_i, it means that x_i is greater than exactly k other values, which means that in the sorted order it must be the $(k+1)$-th component. This is accomplished by sending a^{x_i} in C_{n+k+1}. The maximum number of Fs in C_i is $n-1$ because $C_{i,i}$ will never produce an F. If there are no Fs in C_i, this means that x_i is the minimum, and a^{x_i} will be sent to C_{n+1}. Compartments C_{n+i}, $1 \leq i \leq n$, collect the result of sorting. Each such C_{n+i} will contain at the end of the computation the string $a^{x_{k_i}}$, x_{k_i} being the i-th value in the sorted order. The computation has three steps, the first two ones in which $C_{i,j}$, $1 \leq i, j \leq n$, work, and a third one in which C_i, $1 \leq i \leq n$, work.

Proposition 4. *The above kP system sorts n integers in 3 steps.*

Remark 4. This algorithm makes extensive use of the n^2 additional compartments, $C_{i,j}$, $1 \leq i, j \leq n$, their contents, $a_i^{x_i}$ and $a_j^{x_j}$, and links with the first n components, C_i, $1 \leq i \leq n$. This solution, although computationally efficient, requires an initial, quite complex, setting, i.e., some precomputed resources of size $O(n^2)$. This can be simplified as shown by the next sorting method.

In the second sorting method we consider $2n$ compartments C_i, $1 \leq i \leq 2n$, as above and replace the n^2 compartments, $C_{i,j}$, $1 \leq i, j \leq n$, by a much simpler set of n compartments from which $C_{i,j}$, $1 \leq i, j \leq n$, compartments are obtained by using membrane division rules. Let us say that the new set of compartments are C_k, $2n+1 \leq k \leq 3n$, and each C_k is connected to a C_i, $1 \leq i \leq n$, such that $k - 2n = i$. Each compartment C_k, $2n+1 \leq k \leq 3n$, contains the initial multiset $w_{k,0} = s$, where s is a new object. Membrane division rules can be used to transform each C_k, $2n+1 \leq k \leq 3n$, in one step, into n compartments $C_{i,j}$, $1 \leq j \leq n$ and $k - 2n = i$, $1 \leq i \leq n$. We could use the rules

$$[s]_k \to [{a_1}^{x_1} {a_i}^{x_i} a]_{i,1} \cdots [{a_j}^{x_j} {a_i}^{x_i} a]_{i,j} \cdots [{a_n}^{x_n} {a_i}^{x_i} a]_{i,n}, \;\; 2n+1 \leq k \leq 3n, k - 2n = i.$$

Hence one additional step will be added to the algorithm. The rules of C_i will be modified to account for this additional step. We can now formulate the following result.

Proposition 5. *The above kP system sorts n integers in 4 steps.*

Remark 5. This algorithm requires $3n$ compartments, the first $2n$ have similar structure to the first $2n$ compartments presented in the first sorting method, hence the entire initial settings is much simpler.

4 Simulating and Verifying kP Systems

In Section 3, we have illustrated that kP systems provide a coherent and expressive language that allows us to model various systems that were originally implemented by different P system variants. In addition to the modelling aspect, there has been a significant progress on analysing kP systems using various simulation and verification methodologies. The methods and tools developed in this respect have been integrated into a software platform, called kPWorkbench, to support the modelling and analysis of kP systems.

The ability of simulating kernel P systems is an important feature of this tool. Currently, there are two different simulation approaches, kPWorkbench Simulator and Flame (Flexible Large-Scale Agent Modelling Environment). Both simulators receive as input a kP system model written in kP–Lingua and output a trace of the execution, which is mainly used for checking the evolution of a system and for extracting various results out of the simulation. The simulators provide traces of execution for a kP system model, and an interface displaying the current configuration (the content of each compartment) at each step. It is useful for checking the temporal evolution of a kP system and for inferring various information from the simulation results.

Another important analysis method that kPWorkbench features is formal verification, requiring an exhaustive analysis of system models against some queries to be verified. The automatic verification of kP systems brings in some challenges as they feature a dynamic structure by preserving the structure changing rules such as membrane division, dissolution and link creation/destruction. kPWorkbench employs different verification strategies to alleviate these issues. The framework supports both *Linear Temporal Logic (LTL)* and *Computation Tree Logic (CTL)* properties by making use of the Spin [23] and NuSMV [15] model checkers.

In order to facilitate the formal specification, kPWorkbench features a property language, called *kP-Queries*, comprising a list of natural language statements representing formal property patterns, from which the formal syntax of the Spin and NuSMV formulas are automatically generated. The property language editor interacts with the kP-Lingua model in question and allows users to directly access the native elements in the model, which results in less verbose and shorter state expressions, and hence more comprehensible formulas. *kP-Queries* also features a grammar for the most common property patterns. These features and the natural language like syntax of the language make the property construction much easier.

Some of the commonly used patterns are “next”, “existence”, “absence”, “universality”, “recurrence”, “steady-state”, “until”, “response” and “precedence”. The details can be found in [22].

We now illustrate the usage of the query patters on the sorting algorithm given in Section 3.1. The other algorithms can be considered in a similar manner. In order to verify that the algorithm works as desired, we have constructed a set of properties specified in kP-Queries, listed in Table 1. The applied pattern types are given in the second column of the table. For each property we provide the following information; **(i)** informal description of each kP-Query, and **(ii)** the formal kP-Query using the patterns. The queries given in Table 1 capture that the algorithm given in Section 3.1 works as desired.

We note that both the kP–Lingua model and the queries are automatically converted into the languages required by the corresponding model checkers. So, the verification process in kPWorkbench is carried out in an automatic manner.

Prop.	Pattern	(i) Informal query, (ii) Formal query using patterns
1	Existence	**(i)** *The numbers will be eventually sorted, i.e. the multisets representing the numbers will be in ascending order in the compartments*
		(ii) eventually (c_1.a <= c_2.a & c_2.a <= c_3.a & c_3.a <= c_4.a & c_4.a <= c_5.a & c_5.a <= c_6.a)
2	Universality	**(i)** *Counters in different compartments are always sync'ed*
		(ii) always (c_1.c = c_2.c & c_2.c = c_3.c & c_3.c = c_4.c & c_4.c = c_5.c & c_5.c = c_6.c)
3	Steady-state	**(i)** *In the steady-state, the numbers are sorted*
		(ii) steady-state (c_1.a <= c_2.a & c_2.a <= c_3.a & c_3.a <= c_4.a & c_4.a <= c_5.a & c_5.a <= c_6.a)
4	Existence	**(i)** *The algorithm will eventually stop*
		(ii) eventually (c_i.c = 0)
5	Response	**(i)** *An unsorted state of two adjacent compartments will always be followed by a sorted one*
		(ii) (c_i.a >c_{i+1}.a) **followed-by** (c_i.a <= c_{i+1}.a)

Table 1: List of properties derived from the property language and their representations in different formats.

5 Testing kP Systems Using Automata Based Techniques

In this section we outline how the kP systems obtained in the previous sections can be tested using automata based testing methods. The approach presented here follows the blueprint presented in [25] and [17] for cell-like P systems. We illustrate our approach on $k\Pi_1$, the application of our approach on the other kP system modelling sorting algorithms is similar.

Naturally, in order to apply an automata based testing method to a kP model, a finite automata needs to be obtained first. In general, the computation of a kP system cannot be fully modelled by a finite automaton and so an *approximate* automaton will be sought. The problem will be addressed in two steps.

- Firstly, the computation tree of a P system will be represented as a deterministic finite automaton. In order to guarantee the finiteness of this process, an upper bound k on the length of any computation will be set and only computations of maximum k transitions will be considered at a time.
- Secondly, a *minimal* model, that preserves the required behaviour, will be defined on the basis of the aforementioned derivation tree.

Let $M_k = (A_k, Q_k, q_{0,k}, F_k, h_k)$ be the finite automaton representation of the computation tree, where A_k is the finite input alphabet, Q_k is the finite set of states, $q_{0,k} \in Q_k$ is the initial state, $F_k \subseteq Q_k$ is the set of final states, and $h_k : Q_k \times A_k \longrightarrow Q_k$ is the next-state function. A_k is composed of the tuples of multisets that label the transition of the computation tree. The states of T_k correspond to the nodes of the tree. For testing purposes we will consider all the states as final. It is implicitly assumed that a non-final "sink" state q_{sink} that receives all "rejected" transitions, also exists.

Consider $k\Pi_1$, the kP system in section 3.1, $n = 6$ and the sequence to be sorted 3, 6, 9, 5, 7, 8. Then the initial multisets are:
$w_{1,0} = a^3c^{10}p; w_{2,0} = a^6c^{10}; w_{3,0} = a^9c^{10}p; w_{4,0} = a^5c^{10}; w_{5,0} = a^7c^{10}p; w_{6,0} = a^8c^{10}$. As $k\Pi_1$ is a deterministic kP system, there is no ramification in the computation tree. For $k = 3$, this is represented below.

Compartments - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	$rr^3_{1,1}r_{2,1}$	r	$rr^9_{1,3}r_{2,3}$	r	$rr^7_{1,5}r_{2,5}$	r
1	$rr_{3,1}$	$rr^3_{4,2}$	$rr_{3,3}$	$rr^5_{4,4}r^4_{5,4}$	$rr_{3,5}$	$rr^7_{4,6}$
2	r	$rr^6_{1,2}r_{2,2}$	r	$rr^9_{1,4}r_{2,4}$	r	$rr_{2,6}$
3	r	$rr'_{3,2}$	$rr^5_{1,3}r_{5,3}$	$rr'_{3,4}$	$rr^7_{1,5}r^2_{5,5}$	$rr'_{3,6}$

Let us denote
$\alpha_1 = (rr_{1,1}^3 r_{2,1}, r, rr_{1,3}^9 r_{2,3}, r, rr_{1,5}^7 r_{2,5}, r)$,
$\alpha_2 = (rr_{3,1}, rr_{4,2}^3, rr_{3,3}, rr_{4,4}^5 r_{5,4}^4, rr_{3,5}, rr_{4,6}^7)$,
$\alpha_3 = (r, rr_{1,2}^6 r_{2,2}, r, rr_{1,4}^9 r_{2,4}, r, rr_{2,6})$,
$\alpha_4 = (r, rr'_{3,2}, rr_{1,3}^5 r_{5,3}, rr'_{3,4}, rr_{1,5}^7 r_{5,5}^2, rr'_{3,6})$.

Then, for $k = 3$, $M_k = (A_k, Q_k, q_{0,k}, F_k, h_k)$, where
$A_k = \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\}$, $Q_k = \{q_{0,k}, q_{1,k}, q_{2,k}, q_{3,k}, q_{4,k}\}$, $F_k = Q_k$, and h_k, the next-state function, is defined by: $h_k(q_{i-1,k}, \alpha_i) = q_{i,k}$, $1 \leq i \leq 4$.

As M_k is a deterministic finite automaton over A_k, one can find the minimal deterministic finite automaton that accepts *exactly* the language defined by M_k. However, as only sequences of at most k transitions are considered, it is irrelevant how the constructed automaton will behave for longer sequences. Consequently, a deterministic finite cover automaton of the language defined by M_k will be sufficient.

A *deterministic finite cover automaton* (*DFCA*) of a finite language U is a deterministic finite automaton that accepts all sequences in U and possibly other sequences that are longer than any sequence in U [4], [5]. A *minimal DFCA* of U is a DFCA of U having the least possible states. A minimal DFCA may not be unique (up to a renaming of its states). The great advantage of using a minimal DFCA instead of the minimal deterministic automaton that accepts precisely the language U is that the size (number of states) of the minimal DFCA may be much less than that of the minimal deterministic automaton that accepts U. Several algorithms for constructing a minimal DFCA (starting from the deterministic automaton that accepts the language U) exist, the best known algorithm [27] requires $O(n\ log\ n)$ time, where n denotes the number of states of the original automaton. For details about the construction of a minimal DFCA we refer the reader to [25] and [27].

A minimal DFCA of the language defined by M_k, $k = 3$, is $M = (A, Q, q_0, F, h)$, where $A = A_k$, $Q = \{q_0, q_1, q_2, q_3\}$, $F = Q$ and h defined by: $h(q_{i-1}, \alpha_i) = q_i$, $1 \leq i \leq 3$ and $h(q_3, \alpha_4) = q_0$.

Now, suppose we have a finite state model (automaton) of the system we want to test. In *conformance testing* one constructs a finite set of input sequences, called *test suite*, such that the implementation passes all tests in the test suite if and only if it behaves identically as the specification on any input sequence. Naturally, the implementation under test can also be modelled by an unknown deterministic finite automaton, say M'. This is not known, but one can make assumptions about it (e.g. that may have a number of incorrect transitions, missing or extra states). One of the least restrictive assumptions refers to its size (number of states). The *W-method* [12] assumes that the difference between the number of states of the implementation model and that of the specification has to be at most β, a non-negative integer estimated by the tester. The W-method involves the selection of two sets of input sequences, a state cover S and a characterization set W [12].

In our case, we have constructed a DFCA model of the system and we are only interested in the behavior of the system for sequences of length up to an upper bound k. Then, the set suite will only contain sequences of up to length k and its successful application to the implementation under test will establish that the implementation will behave identically to the specification for any sequence of length less then or equal to k. This situation is called conformance testing for bounded sequences. Recently, it was shown that the underlying idea of the W-method can also be applied in the case of bounded sequences, provided that the sets S and W used in the construction of the test suite satisfy some further requirements; these are called a proper state cover and strong characterization set, respectively [24]. In what follows we informally define these two concepts and illustrate them on our working example. For formal definitions we refer the reader to [24] or [25].

A *proper state cover* of a deterministic finite automaton $M = (A, Q, q_0, F, h)$ is a set of sequences $S \subseteq A^*$ such that for every state $q \in Q$, S contains a sequence of *minimum length* that reaches q. Consider M the DFCA in our example. Then λ is the sequence of minimum length that reaches

q_0, σ_1 is a sequence of minimum length that reaches q_1, $\alpha_1\alpha_2$ is a sequence of minimum length that reaches q_2, $\alpha_1\alpha_2\alpha_3$ is a sequence of minimum length that reaches q_3. Furthermore, we can use any input symbol in $A \setminus \{\alpha_1\}$ to reach the (implicit) "sink" state, for example α_2. Thus, $S = \{\lambda, \alpha_1, \alpha_1\alpha_2, \alpha_1\alpha_2\alpha_3, \alpha_2\}$ is a proper state cover of M.

A *strong characterization set* of a minimal deterministic finite automaton $M = (A, Q, q_0, F, h)$ is a set of sequences $W \subseteq A^*$ such that for every two distinct states $q_1, q_2 \in Q$, W contains a sequence of minimum length that distinguishes between q_1 and q_2. Consider again our running example. λ distinguishes between the (non-final) "sink" state and all the other (final) states. A transition labelled α_1 is defined from q_0, but not from q_1, q_2 or q_3, so α_1 is a sequence of minimum length that distinguishes q_0 from q_1, q_2 and q_3. Similarly, α_2 is a sequence of minimum length that distinguishes q_1 from q_2 and q_3 and α_3 is a sequence of minimum length that distinguishes between q_2 and q_3. Thus $W = \{\lambda, \alpha_1, \alpha_2, \alpha_3\}$ is a strong characterization set of M,

Once we have established the sets S and W and the maximum number β of extra states that the implementation under test may have, a test suite is constructed by extracting all sequences of length up to k from the set

$$S(A^0 \cup A^1 \cup \cdots \cup A^\beta)W,$$

where A^i denotes the set of input sequences of length $i \geq 0$.

Note that some test sequences may be accepted by the DFCA model - these are called *positive tests* - but some others may not be accepted (they end up in the (non-final) "sink" state) - these are called *negative tests*.

6 Conclusions

In this paper, we have illustrated the modelling power of kernel P systems by providing a number of kP system models for sorting algorithms.These prove that the kP systems approach provides a coherent and expressive language that allows us to model various systems that were originally implemented by different P system variants. We have also discussed the problem of testing systems modelled as kernel P systems and proposed a test generation method based on automata. Namely, we have outlined how the kP systems can be tested using automata based testing methods. Furthermore, we have demonstrated how formal verification can be used to validate that the given models work as desired.

In our future work we aim to show how other problems can be solved, tested and verified by using kP systems and also to prove how existing classes of P systems can be expressed with this formalism.

Acknowledgements

The authors are indebted to the anonymous reviewers for carefully reading and providing comments allowing us to improve the content and presentation of the paper. MG and SK acknowledge the support provided for synthetic biology research by EPSRC ROADBLOCK (project number: EP/I031812/1). The work of FI and MG was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PN-II-ID-PCE-2011-3-0688).

References

1. A. Alhazov, D. Sburlan, Static Sorting Algorithms for P Systems, *Pre-Proc. 4^{th} Workshop on Membrane Computing* (A. Alhazov et al., eds.), *GRLMC Rep.* 28/03, Tarragona, 17 – 40, 2003.
2. A. Alhazov, D. Sburlan, Static Sorting P Systems. In [16], 215 – 252, 2006.

3. J.J. Arulanandham, Implementing Bead-Sort with P Systems, *Unconventional Models of Computation* (C.S. Calude et al., eds.), *Lecture Notes in Computer Science*, 2509, 115–125, 2002.
4. C. Câmpeanu, N. Santean, S. Yu, Minimal Cover-Automata for Finite Languages, *Workshop on Implementing Automata* (J.-M. Champarnaud et al., eds.), *Lecture Notes in Computer Science*, 1660, 43 – 56, 1998.
5. C. Câmpeanu, N. Santean, S. Yu, Minimal Cover-Automata for Finite Languages. *Theoretical Computer Science*, 267(1-2), 3 – 16, 2001.
6. R. Ceterchi, C. Martín-Vide, P Systems with Communication for Static Sorting. In *Pre-Proc. 1^{st} Brainstorming Week on Membrane Computing* (M. Cavaliere et al., eds.), *Technical Report* no 26, Rovira i Virgili Univ., Tarragona, 101 – 117, 2003.
7. R. Ceterchi , C. Martín-Vide, Dynamic P Systems, *Proc. 4^{th} Workshop on Membrane Computing* (Gh. Păun et al., eds.), *Lecture Notes in Computer Science*, 2597, 146 – 186, 2003.
8. R. Ceterchi, M.J. Pérez-Jiménez, A.I. Tomescu, Simulating the Bitonic Sort Using P Systems, *Proc. 8^{th} Workshop on Membrane Computing* (G. Eleftherakis et al., eds.), *Lecture Notes in Computer Science*, 4860, 172 – 192, 2007.
9. R. Ceterchi, M.J. Pérez-Jiménez, A.I. Tomescu, Sorting Omega Networks Simulated With P Systems: Optimal Data Layouts, (D. Diaz-Pernil et al., eds.), *Proc. 6^{th} Brainstorming Week on Membrane Computing*, *RGNC Rep.* 01/08, Fénix Editora, pp. 79 – 92, 2008.
10. R. Ceterchi, A. I. Tomescu, Implementing Sorting Networks with Spiking Neural P Systems, *Fundamenta Informaticae*, 87(1), 35 – 48, 2008.
11. R. Ceterchi, D. Sburlan, Membrane Computing and Computer Science, Chapter 22 of [30], 553–583, 2010.
12. T. S. Chow, Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3), 178 – 187, 1978.
13. 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*, 90(4), 801 – 815, 2013.
14. R. Freund, S. Verlan, A Formal Framework for Static (Tissue) P Systems, In Membrane Computing, emphProc. 8^{th} Workshop on Membrane Computing (G. Eleftherakis et al., eds.), *Lecture Notes in Computer Science*, 4860, 271 – 284, 2007.
15. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV Version 2: An Open Source Tool for Symbolic Model Checking, *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, (W.A. Hunt, Jr and F. Somenzi, eds.), *Lecture Notes in Computer Science*, 2404, 359 – 364, 2002.
16. G. Ciobanu, Gh. Păun, M. J. Pérez-Jiménez, eds., *Applications of Membrane Computing*, Springer, 2006.
17. M. Gheorghe, F. Ipate. On Testing P Systems, *Proc. 9^{th} Workshop on Membrane Computing*, (D.W. Corne et al., eds.), *Lecture Notes in Computer Science*, 5391, 204 – 216, 2009.
18. M. Gheorghe, F. Ipate, C. Dragomir, Kernel P Systems, *Proc. 10^{th} Brainstorming Week on Membrane Computing*, (M. A. Martínez-del-Amor et al., eds.), Fénix Editora, Universidad de Sevilla, 153 – 170, 2012.
19. M. Gheorghe, F. Ipate, C. Dragomir, L. Mierlă, L. Valencia-Cabrera, M. García-Quismondo, M.J. Pérez-Jiménez, Kernel P Systems – Version 1, *Proc. 11th Brainstorming Week on Membrane Computing*, (L. Valencia-Cabrera et al., eds.), Fénix Editora, Universidad de Sevilla, 97 – 124, 2013.
20. M. Gheorghe, F. Ipate, R. Lefticaru, M.J. Pérez-Jiménez, A. Ţurcanu, L. Valencia-Cabrera, M. García-Quismondo, L. Mierlă, 3-COL Problem Modelling Using Simple kernel P Systems, *International Journal of Computer Mathematics*, 90(4), 816 – 830, 2013.
21. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg, Research Frontiers of Membrane Computing: Open Problems and Research Topics, *International Journal of Foundations of Computer Scienve*, 24, 547 – 624, 2013.
22. M. Gheorghe, S. Konur, F. Ipate, L. Mierlă, M. E. Bakir, M. Stannett, An Integrated Model Checking Toolset for Kernel P Systems, *Proc. 16^{th} Conference on Membrane Computing*, (G. Rozenberg et al., eds.), *Lecture Notes in Computer Science*, 9504, 153 – 170, 2015.

23. G. J. Holzmann, The Model Checker SPIN, *IEEE Transactions on Software Engineering*, 23(5), 275 – 295, 1997.
24. F. Ipate, Bounded Sequence Testing from Deterministic Finite State Machines, *Theoretical Computer Science*, 411(16-18), 1770 – 1784, 2010.
25. F. Ipate, M. Gheorghe, Finite State Based Testing of P Systems, *Natural Computing*, 8(4), 833 – 846, 2009.
26. D.E. Knuth, *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Addison-Wesley, 1973.
27. H. Körner, On Minimizing Cover Automata for Finite Languages in O(n log n) Time, *Proc. 7^{th} Conference on Implementation and Application of Automata*, (J.-M. Champarnaud and D. Morel, eds.), *Lecture Notes in Computer Science*, 2608, 117 – 127, 2002.
28. Gh. Păun, Computing with Membranes, *Journal of Computer and System Sciences*, 61(1), 108 – 143, 2000.
29. Gh. Păun, Membrane Computing - An Introduction, Springer, 2002.
30. Gh. Păun, G. Rozenberg, A. Salomaa, eds., *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
31. D. Sburlan, A Static Sorting Algorithm for P Systems with Mobile Catalysts, *Analele Ştiinţifice Universitatea Ovidius Constanţa*, 11(1), 195 – 205, 2003.

Walking Membranes: Grid-exploring P Systems with Artificial Evolution for Multi-purpose Topological Optimisation of Cascaded Processes

Thomas Hinze[1,2], Lea Louise Weber[1], and Uwe Hatnik[3]

[1]Brandenburg University of Technology
Institute of Computer Science
Postfach 10 13 44, D-03013 Cottbus, Germany

[2]Friedrich Schiller University Jena
Ernst-Abbe-Platz 1–4, D-07743 Jena, Germany

[3]Fraunhofer Institute for Integrated Circuits IIS, Design Automation Division EAS,
Zeunerstraße 38, D-01069 Dresden, Germany

thomas.hinze@b-tu.de, weberlea@b-tu.de, uwe.hatnik@eas.iis.fraunhofer.de

Abstract. The capability of *self-organisation* belongs to the most fascinating features of many living organisms. It results in formation and continuous adjustment of dedicated *spatial structures* which in turn can sustain a high fitness and efficient use of resources even if environmental conditions or internal factors tend to vary. Spatial structures in this context might for instance incorporate *topological arrangements* of cellular compartments and filaments towards fast and effective signal transduction. Due to its discrete nature, the P systems approach represents an ideal candidate in order to capture emergence and evolution of topologies composed of membranes passable by molecular particles. We introduce *grid-exploring P systems* in which generalised membranes form the grid elements keeping the grid structure variable. Particles initially placed at different positions of the grid's boundary individually run through the grid visiting a sequence of designated membranes in which they become successively processed. Using artificial evolution, the arrangement of membranes within the grid becomes optimised for shortening the total time duration necessary for complete passage and processing of all particles. Interestingly, the corresponding framework comprises numerous practical applications beyond modelling of biological self-organisation. When replacing membranes by queue-based treads, tools, or processing units and particles by customers, workpieces, or raw products, we obtain a multi-purpose optimisation strategy along with a simulation framework. Three case studies from cell signalling, retail industry, and manufacturing demonstrate various benefits from the concept.

1 Introduction and Background

Living organisms appear almost perfectly adapted to environmental conditions. A plethora of elaborated survival strategies in concert with highly *optimised*

form and function enables maintenance of individuals (*autopoiesis*) as well as long-term persistence of entire biological species or colonies. Impressive examples range from nanoscaled intracellular pathway mechanisms [10, 15] via shapes of tissues or seeds [3, 7, 18] up to complex behavioural patterns of ant colonies, construction of insects' nests, or positioning of foxholes within a bumpy landscape [4, 5, 19]. All these phenomena have in common that spatial structures and arrangements follow the best possible way to fulfil a useful function in helping the organism to cope with realities.

Sometimes, a more or less static formation of a structure is sufficient. Let us consider for instance the common sunflower (*Helianthus annuus*). Its florets in the head are arranged in a way to assure a maximum exploitation of light energy since the individual florets have small offsets to each other. This in turn gives two advantages: Firstly, the shadow induced by a floret cannot significantly cover another floret. Secondly, the total number of florets within the sunflower's head reaches its maximum by close packaging [21]. To this end, the florets form centered spiral structures (Fermat's spirals) whose rotation angle resembles the golden ratio expressed by Fibonacci numbers, see Figure 1 left part.

The scenario becomes more complicated when considering temporally *dynamic control* of spatial structures in an adaptively continuous manner instead of a one-time static formation. Resulting systems turn out to be highly robust against perturbations and damages. In addition, they might be able to restore themselves up to a certain degree of damage. Moreover, corresponding organisms undergo a permanent "self-assembly", "self-optimisation", and "self-healing" aimed at providing the best possible survival conditions [8]. An illustrative example in this context can be seen in the wide cardiovascular network of human blood vessels [22, 27]. The existence and reliability of this circulatory system is essential for supply of each single cell with nutrients, metabolites, hormones, and other messengers. Substances released by a cell are also transported via the blood stream. The underlying topology of the cardiovascular network obviously follows its function taking into account a minimum need of material resources and mechanical energy [23]. Mostly, this becomes evident by the placement of branches and junctions successively dividing arteries and capillaries towards more and more fine-grained spatial structures sketched in Figure 1 right part. Number and spatial positions of junctions keep appropriately balanced [9]. Interestingly, the cardiovascular network topology re-organises all the time [16]. Along with human ontogenesis from embryonic state and during childhood, the network initially grows. After maturation, the network further adapts to the individual lifestyle. If a capillar blood vessel becomes locked due to plugs, neighboured vessels can extend and re-branch in order to compensate the damage.

Both aforementioned biological examples – arrangement of florets in sunflower heads as well as the cardiovascular network topology of human blood vessels – demonstrate the capability of *self-organisation*. There are much more examples in many facets of biology and medicine. In all cases, spatial structures exhibit a certain flexibility which has been permanently utilised to find out the best possible topology to achieve adaptively under present constraints. Modi-

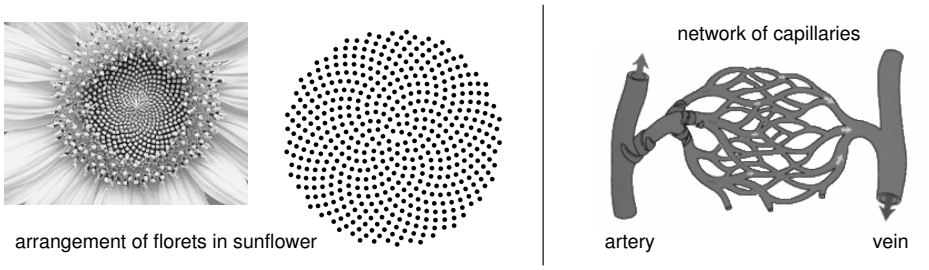


Fig. 1. Biological examples of spatial structures emerged from self-organising formation and maintenance. **Left:** Close two-dimensional packaging of florets in a sunflower head forming Fermat's spirals [21]. **Right:** Schematic representation of a cardiovascular network structure composed of blood vessels by self-organisation.

fications affecting the topology have been initiated endogenously which means without any external control. Inspired by its potential in nature, we are going to develop an abstract desriptive framework for exploration of self-organisation *in-silico* on a two-dimensional grid along with a corresponding software for system's configuration and behavioural simulation. The P systems approach in general provides an ideal candidate to formalise this framework since it can directly cope with dynamical structures due to its employent of algebraic elements and flexible hierarchically nested compositions from that.

What stands out after study of diverse systems equipped with self-organisation is that in a majority of cases the optimised topology has been passed by *particles*. Typically, the particles carry a dedicated *meaning* for the underlying system such as providing information, energy supply, or messages. For instance, hormones as particles within the blood stream should fast and safely reach their specific destination cells coupled to the blood vessels. Photons assumed as particles should intensively and homogeneously penetrate the sunflower florets instead of getting lost aside. In other words, the entirety of particles passing through the topology defines the *fitness* of the topology on its own. During passage, a particle is allowed to consult a predetermined sequence of destinations within the topology in terms of a signalling cascade. Here, each destination acts as a processing unit for particles which in turn successively proceed and finally leave the system or get consumed. In the end, the whole amount of time necessary to completely process a given initial setting of particles measures the quality of the underlying topology under study. Slight *modifications* of the topology can lead to a better, worse, or unchanged quality. Those producing a better quality are retained. Modifications of the topology characterise an adaptive self-organisation able to manage varying settings of particles over time.

Potential applications of self-organising topologies are not restricted to modelling of pure biological phenomena. Particularly, the fourth industrial revolution (*"Industry 4.0"*) comes along with an increased need of so-called self-X properties [14]. Formerly large-volume fabrication of uniform products has been more and more transformed into assembly of highly individualised products. To this

end, instead of mass-products, a huge variety of customised goods emerges. In order to successfully manage the underlying production processes, a continuous self-configuration, self-optimisation, and self-organisation of involved machines, processing units, and manufacturing facilities becomes essential. So, the placement and arrangement of moveable machines within a factory building could reorganise according to the current production order. In this way, groups or clusters of machines placed close to each other enable a cascaded form of production with short transportation distances of raw products. Here, a self-organised topology of machines aims at fastest passage and successive processing of raw products towards final products. In this scenario, the raw products act as particles while the underlying two-dimensional grid is composed of regions ("membranes"), each of them covered by a machine or forming a paved area for transportation. Having in mind that several types of machines can exist and raw materials prior to raw products may enter the grid at different entries, a non-trivial optimisation problem occurs in which the best possible topology needs to be found. Whenever there is a change of the production order, the optimisation problem arises again. Coping with dynamical structures within a formal framework turns out to be a final clue which opens a new field of applications for the P systems approach.

Another non-biological example of a self-organising two-dimensional grid of membranes passed by particles comes from retail industry when considering a typical supermarket. Nowadays, the arrangement of products at the two-dimensional ground follows a sophisticated scheme resulting from modern sales psychology. Customers should stay for a long time in the supermarket discovering more and more attractive products alongside their route from the entry to the cashpoints. In contrast, let us imagine an alternative form called *"Supermarket 2.0"* reflecting the habit of educated customers: They know in advance what products they are looking for. In addition, they permanently suffer from lack of time. This type of customers is interested in finding all desired products as fast as possible walking across the supermarket at the shortest possible path. Scanners in concert with contact-free automatic payment could further accelerate the shopping. While passing a supermarket's exit, the customer confirms the price to pay simply by pressing a button and having the debit card in the pocket. This concept has been already under test study [6, 24]. It avoids the queue in front of conventional cashpoints. In this context, an optimal placement of products adapted to the preferences of educated customers defines an appealing scenario for a self-organisation framework in which customers represent the particles.

Motivated by these and many further application scenarios, we introduce grid-exploring P systems for topological optimisation of cascaded processes. Here, self-organisation towards fastest passage and processing of particles is carried out using artificial evolution which in turn cares for variation of grid elements. Since grid elements constitute membranes able to be entered and left by particles, the metaphor of *walking membranes* depicts the central idea traced throughout this paper. Principles of self-organisation have been addressed in the field of membrane computing from time to time. Tissue P systems [17] reflect the idea of a membrane grid. By grid-exploring P systems, we extend the notion of

tissue P systems by dedicated transmembranous instructions (sequences of processing units) to be executed by each particle. Modelling of swarm-based multi-agent systems succeeded using population P systems [26]. In [2], self-assembly by consecutive membrane division in population P systems was modelled. Self-adaptive and reconfigurable distributed computing systems were introduced in [1]. Here, self-organisation is employed to minimise failures in network partitioning. Each node in the network stands for a uniform type of processor. When looking at self-modifiable sequences of instructions able to compose functional chemical units (modules) on the fly, we refer the reader to [13]. In [11], evaluation, accumulation, and categorised counting of particles initially positioned at a planar two-dimensional surface was considered for image analysis using blotting P systems. Some approaches in membrane computing are directed at modelling of dynamical structures in various facets. For instance, variable molecular structures expressed by modifiable character strings became apparent in [12] while active membranes flank the notion of P systems almost from its beginning [20].

In the following section, we familiarise the reader with the general concept of grid-exploring P systems which also sheds light on the assumptions made towards an abstract, flexible, and nevertheless widely practicable framework. After that, Section 3 is dedicated to define the underlying formalisms and algorithms comprising the framework of grid-exploring P systems and their behaviour over time. Our model comes along with three case studies revealing the overall capability of explorative two-dimensional grid optimisation in different application scenarios. First, we address biological cell signalling by taking into consideration several pathways (Section 4.1). Placement of receptors in conjunction with downstream signalling cascade destinations can induce a broad spectrum of latencies prior to cell response depending on the type (urgency) of stimulus signal. Beyond biology, Section 4.2 is focused on a production scenario in manufacturing. Here, an optimised placement of processing units for cascaded production of several goods is exemplified by a cabinet maker's workshop. The advantage of specific islands of machine tools over conventional production lines becomes visible. In Section 4.3, we develop an experimental idea of a possible alternative supermarket in the future whose arrangement of products follows the needs of educated customers in hurry. Even if a bit visionary, this example could sketch a new prototype of supermarket dominated by groups of frequently chosen product combinations. Final remarks discuss the concept of grid-exploring P systems regarding its extensibility for further work.

2 General Concept

Basic prerequisite for the topological optimisation is a configurable initial $n \times m$ grid composed of *membranes* as grid elements. In this context, a membrane represents a square-shaped region or area able to be entered, passed, and left by particles. In addition, membranes can also process or consume particles during passage. According to its functional purpose, we assume different predefined types of membranes:

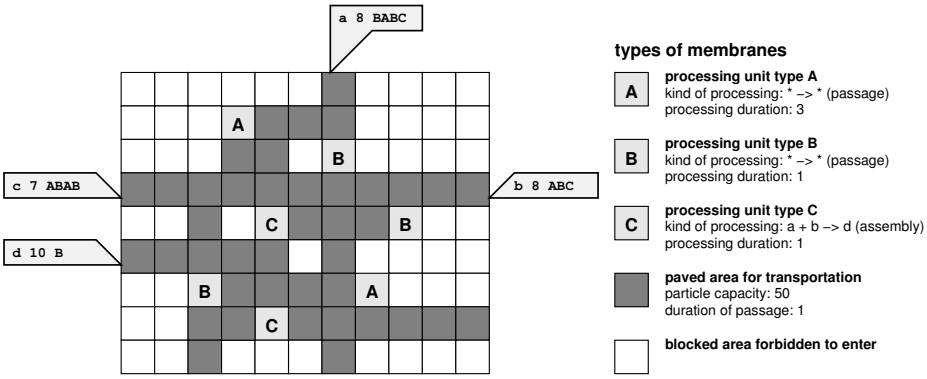


Fig. 2. Example of a predefined 9×11 grid with 7 entries/exits and 7 processing units marked by capitalised letters. From each processing unit, any processing unit of different type can be reached via paved areas. Initially, particles are placed in front of entries. There are four categories of particles called `a`, `b`, `c`, and `d`. Each particle is obliged to consult the corresponding sequence of processing units. By doing so, it can either be consumed within a processing unit (like in unit `C` which assembles $\texttt{a} + \texttt{b} \rightarrow \texttt{d}$ by consumption of particles `a` and `b`) or finally runs to the nearest entry/exit after completion of processing.

- A **paved area** allows particle transportation. The corresponding membrane can be passed by particles in all directions without any processing or modification. Each paved area comes with two individual parameters: Its *capacity* defines the maximum number of particles permitted to stay inside the membrane at the same time. In case of exhausted capacity, no further particles can enter. Another parameter is the *duration of passage*, expressed by a natural number > 0. Its value marks the minimum number of time steps each particle must reside inside the membrane. Afterwards, it can leave the membrane by entering an adjacent membrane if possible. In its entirety, paved areas make accessible the grid for particles. When placed adjacent to each other, sequenced paved areas develop pathes throughout the grid to be trodden by particles. Paved areas placed at the outer boundary of the grid act as combined entries and exits. Each grid must have at least one entry/exit.
- A **blocked area** is a permanently empty membrane forbidden to become entered by particles. Using blocked areas, fixed zones can be excluded from any topological consideration and variation. This enables incorporation of specific immutable features of the underlying landscape or ground.
- A **processing unit** specifies a membrane able to affect particles which are permitted to enter from all adjacent paved areas. Since processing of particles can happen in a varied manner, we distinguish different types of processing units. The number of processing unit types can be arbitrarily chosen but at least one is mandatory. For simplicity, we assign a capitalised letter starting from `A` for each processing unit type. The grid example shown in Figure 2 utilises three types denoted by `A`, `B`, and `C`. Addressed by its name, each

processing unit type comprises two attributes for its behavioural specification. The *processing duration*, a natural number > 0, indicates the number of time steps necessary to carry out the processing. Furthermore, the *kind of processing* is given. Here, either the mode *passage* or *assembly* are available.

- **Passage**, marked by $\texttt{*} \rightarrow \texttt{*}$, leaves intact each entering particle which migrates to an adjacent membrane after processing as soon as possible. At the same time, at most one particle is allowed to be present in the processing unit.
- **Assembly** emulates a *composition*, *incorporation*, or *unification* of two particles which results in a particle again. Let particles `a`, `b`, and `c` be present within the grid, assembly can be of the form $\texttt{a} + \texttt{b} \rightarrow \texttt{c}$ (composition) but also $\texttt{a} + \texttt{b} \rightarrow \texttt{a}$ (incorporation) or $\texttt{a} + \texttt{a} \rightarrow \texttt{b}$ (unification). In the assembly mode, exactly two particles of the processed form are permitted to reside simultaneously within the processing unit. Along with assembly, these two particles are consumed releasing the corresponding product particle to leave the membrane as soon as possible.

Several instances of processing units of each type might be placed at the initial grid. Each processing unit must be reachable via at least one paved area. Adjacent positioning of processing units is forbidden in order to imply a transportation phase between subsequent processing steps.

Figure 2 illustrates an example of an initial grid configuration complemented by the placement of particles before passing the grid. For system's setup, we distinguish an arbitrarily chosen but final number of *particle categories*, for simplicity named by lower-case letters beginning from `a`. Each particle category comprises the total number of individual particles together with a uniform final sequence of processing unit identifiers to be passed. All particles from the same category are collectively placed in front of an arbitrary entry/exit of the grid. In the example shown in Figure 2, there are four categories of particles (`a`, `b`, `c`, and `d`). Within this example setting, a total amount of 8 particles from category `a` are situated on top of the only entry/exit at the upper bound of the grid. Each particle from `a` must migrate to the nearest processing unit from type `B`, then `A`, `B` again, and finally `C`. Since `C` consumes particles of category `a`, their passage is finished there. Otherwise, particles after all steps of processing run to the next entry/exit to leave the grid.

Having the initial grid with its membranes and placement of particles at hand, a behavioural simulation traces the passage of the particles through the grid over time until all particles from all categories left the grid or got consumed. The corresponding total number of times steps taken from a global clock marks the *fitness* of the grid under study. The fitness measure reflects the suitability of the grid for processing all particles in the desired manner. In the example given in Figure 2, we obtain a fitness of 98 time steps. Please note that a particle must wait inside a membrane if the subsequent membrane on its route cannot be entered since it is fully occupied. In this way, the passage could get delayed. By variation of the topological arrangement of processing units within the accessible part of the grid, the corresponding fitness values might deviate from each other.

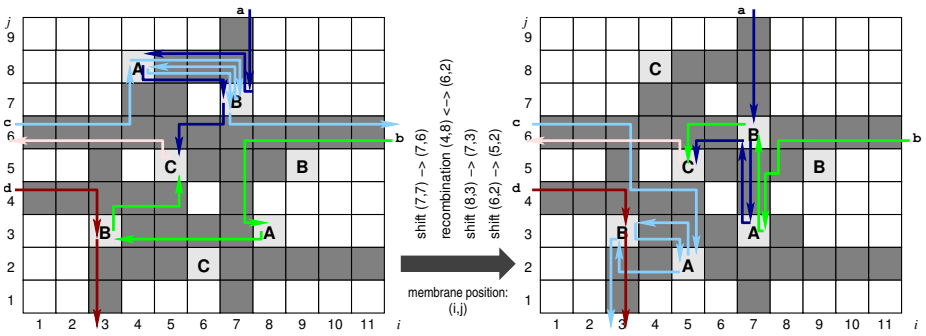


Fig. 3. Four evolutionary cycles can improve the fitness from 98 time steps (initial grid, left part) to 75 time steps (final grid, right part). Placement of particles and parameter setting of processing units and paved areas remain unchanged. The underlying artificial evolution carries out a self-organising optimisation of the grid topology in order to diminish the overall time of passage for all particles from the initial placement.

The faster a grid can process all particles, the higher its fitness. It may happen that a grid cannot completely process all particles due to two reasons: In case of a deadlock (circular path of membranes whose capacities are exhausted) or in case of persisting particles unable to become assembled due to lack of their counterparts, the overall duration of passage (execution time) is set to ∞ which implies worst possible fitness.

Out of the initial grid, a variety ("population") of grids is generated using *artificial evolution*. To this end, we introduce two evolutionary operators called *recombination* and *shift* able to modify the grid topology:

- **Recombination** randomly selects two processing units out of the whole grid. Both processing units get exchanged with each other. In case that either processing units are from the same type, the recombination has no effect, and the original grid is reproduced.
- **Shift** randomly picks one processing unit which swaps its place with one of the adjacent paved areas which in turn have been also identified by random in equipartition.

The initialisation phase of the artificial evolution creates a population of grids. Here, a number of copies ("individuals") from the initial grid is produced. Each of them undergoes either a recombination or a shift in which the occurrence of both operators is kept in parity. All grids emerged in this way are checked for validity. Invalid grids become removed from the population and replaced by additional ones until the desired population size is reached. A typical population size in our case studies comprises 50 grids. An individual fitness evaluation reveals the qualities of all grids in the population. The ascending order of fitness values identifies at its end a number of worst grids (20% of the population size) to be eliminated from the population. From the surviving grids, an appropriate number of copies is made and each of them tackled once by an evolutionary

operator in order to fill up the population. So, the new generation of the grid population consists of a mixture of parents and offsprings in which the original initial grid always persists independently of its fitness in order to ensure a revitalisation of the population if necessary. We run the evolution loop until there is no improval of the best fitness over 100 generations. An optimisation result of the initial grid exemplified in Figure 2 is shown in Figure 3.

3 Grid-exploring P Systems: Definitions and Formalisms

3.1 Algebraic and string-operational prerequisites

Let A and B be arbitrary sets, $\emptyset$ the empty set, $\mathbb{N}$ the set of natural numbers including zero. The term $|A|$ denotes the number of elements in A (cardinality). The Cartesian product of A and B is written by $A \times B$. A multiset over A is a mapping $F : A \longrightarrow \mathbb{N}$. A multiset can also be specified by unordered enumeration of multiple elements like for instance $\{a, a, b, a, b\}$ instead of $\{(a, 3), (b, 2)\}$. The support $\mathrm{supp}(F) \subseteq A$ of F is defined by $\mathrm{supp}(F) = \{a \in A \mid F(a) > 0\}$. A multiset F over A is said to be empty iff $\forall a \in A : F(a) = 0$. The cardinality $|F|$ of F over A is $|F| = \sum_{a \in A} F(a)$. Let A be a set, $a \in A$, and $F : A \longrightarrow \mathbb{N}$ a multiset. We define the removal $F \backslash \{a\} \Leftrightarrow F(a) := F(a) - 1$ and the incorporation $F \cup \{a\} \Leftrightarrow F(a) := F(a) + 1$, respectively.

Let Σ be an alphabet, ε the empty word, and $w \in \Sigma^*$ a word over Σ. The symbol $x \in \Sigma$ is called *prefix*(w) iff $w = xy$ and $y \in \Sigma^*$. The symbol $z \in \Sigma$ is called *suffix*(w) iff $w = yz$ and $y \in \Sigma^*$. Let $u, v \in \Sigma^*$ words over the same alphabet. Concatenation $u \oplus v := uv$ appends v to u. For removal of the leftmost symbol x from a word w, we define $w \ominus x := y$ with $w = xy$.

3.2 Definition of system components

A grid exploring P system $\Pi_\square$ is a construct

$$\Pi_\square = (m, n, \Sigma_F, \Sigma_P, G_{\mathrm{mbrns}}, G_{\mathrm{capac}}, G_{\mathrm{durat}}, F, P)$$

with its components

$m \in \mathbb{N} \setminus \{0\}$ ..... number of grid columns

$n \in \mathbb{N} \setminus \{0\}$ ..... number of grid rows

Σ_F ..... alphabet of processing unit types

Σ_P ..... alphabet of particle categories

$G_{\mathrm{mbrns}} : \{1, \ldots, m\} \times \{1, \ldots, n\} \longrightarrow \Sigma_F \cup \{\#\} \cup \{\perp\}$
grid of membranes, denoted by a matrix and represented by a function whose arguments identify column and row. Assigned function values provide the type of membrane at the corresponding grid position. Available typs are processing units ($\in \Sigma_F$), paved areas ($\#$), and blocked areas ($\perp$).

$G_{\text{capac}} : \{1,\ldots,m\} \times \{1,\ldots,n\} \longrightarrow \mathbb{N}$
capacities of grid elements which define the maximum number of particles allowed to be present at the same grid membrane simultaneously. Blocked areas are assumed to have a capacity of 0. All other membranes should constitute individual capacities > 0.

$G_{\text{durat}} : \{1,\ldots,m\} \times \{1,\ldots,n\} \longrightarrow \mathbb{N} \setminus \{0\}$
durations necessary for particle passage or processing individually assigned to each membrane within the grid. Each duration is expressed by a number of time steps.

$F : \Sigma_F \longrightarrow \{\star\} \cup \Sigma_P^3$. mode (kind of processing) for each processing unit type

$P : \{((i,j),p,f) \mid i \in \{1,\ldots,m\} \wedge j \in \{1,\ldots,n\} \wedge p \in \Sigma_p \wedge f \in \Sigma_F^*\} \longrightarrow \mathbb{N}$
finite multiset of particles. Each particle comes with individual attributes such as its position (i,j) at the grid, its category p, and a finite sequence (word) f of processing unit types to be consecutively passed through.

3.3 Auxiliary data to be obtained prior to system's evolution

Undirected graph (V, E) of routes through the grid of membranes

$$V = \left\{ \underline{(a,b)} \mid G_{\text{mbrns}}(a,b) \in \Sigma_F \cup \{\#\} \right\} \text{ and } E \subseteq V \times V \text{ with}$$

$$E = \Big\{ \Big(\underline{(a,b)}, \underline{(c,d)} \Big), \Big(\underline{(c,d)}, \underline{(a,b)} \Big) \mid G_{\text{mbrns}}(a,b) \in \Sigma_F \cup \{\#\} \ \wedge$$
$$G_{\text{mbrns}}(c,d) \in \Sigma_F \cup \{\#\} \ \wedge \ (((|a-c| = 1) \wedge (b = d)) \vee ((|b-d| = 1) \wedge (a = c))) \Big\}$$

The graph (V, E) identifies the adjacence structure of grid membranes. Each accessible membrane of the underlying grid results in a node while adjacent membranes get bidirectionally connected by edges.

Shortest routes through graph (V, E)

Using the *Floyd-Warshall* algorithm [25], the shortest path from each node to each other node along with its length is calculated by filling two matrices.

matrix of shortest routes $H_{\text{route}} : V \times V \longrightarrow V^*$
matrix of shortest distances $H_{\text{dist}} : V \times V \longrightarrow \mathbb{N} \cup \{\infty\}$

```
for (i,j) ∈ V × V:
  H_route(i,j) := ε
  H_dist(i,j) := { 1  iff (i,j) ∈ E
                 { ∞  otherwise
for (k,l) ∈ V:
  for ((o,p),(q,r)) ∈ E:
    dist := H_dist((o,p),(k,l)) + H_dist((k,l),(q,r))
    if (dist < H_dist((o,p),(q,r))):
      H_dist((o,p),(q,r)) := dist
      H_route((o,p),(q,r)) := H_route((o,p),(k,l)) ⊕ (k,l) ⊕ H_route((k,l),(q,r))
```

Check for validity of the grid of membranes
Since the grid must be passable by the particles in the desired sequence of processing units, a validity check of the grid prior to its fitness evaluation and optimisation is required. Invalid grids cannot be handled. We formulate the validity check by a number of constraints which have to be met in conjunction.
`validity_check` : $(m, n, G_{\text{mbrns}}, H_{\text{dist}}, H_{\text{route}}) \mapsto \{true, false\}$

• at least one entry/exit available

$$\exists (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\}. \\ (G_{\text{mbrns}}(1,b) = \# \vee G_{\text{mbrns}}(a,1) = \# \vee G_{\text{mbrns}}(m,b) = \# \vee G_{\text{mbrns}}(a,n) = \#)$$

• no processing unit placed at the outer grid boundaries

$$\nexists (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\}. \\ (G_{\text{mbrns}}(1,b) \in \Sigma_F \vee G_{\text{mbrns}}(a,1) \in \Sigma_F \vee G_{\text{mbrns}}(m,b) \in \Sigma_F \vee G_{\text{mbrns}}(a,n) \in \Sigma_F)$$

• no adjacent processing units

$$\nexists (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; (G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge G_{\text{mbrns}}(a,b+1) \in \Sigma_F \vee \\ G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge G_{\text{mbrns}}(a+1,b) \in \Sigma_F)$$

• each processing unit type reachable from each other processing unit

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge G_{\text{mbrns}}(c,d) \in \Sigma_F) \;\Rightarrow \\ (H_{\text{dist}}(\underline{(a,b)}, \underline{(c,d)}) \in \mathbb{N} \vee H_{\text{route}}(\underline{(a,b)}, \underline{(c,d)}) = \varepsilon)$$

• route from an any processing unit to another one exclusively via paved areas

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge G_{\text{mbrns}}(c,d) \in \Sigma_F \wedge a \neq c \wedge b \neq d) \;\Rightarrow \\ (\forall x \in V \;.\; H_{\text{route}}(\underline{(a,b)}, \underline{(c,d)}) = wxy \wedge w \in V^* \wedge y \in V^* \wedge G_{\text{mbrns}}(x) = \#)$$

• from any processing unit at least one entry/exit reachable

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ G_{\text{mbrns}}(a,b) \in \Sigma_F \Rightarrow \exists H_{\text{route}}(\underline{(a,b)}, \underline{(c,d)}) \in V^* \;.\; (c = 1 \vee c = m \vee d = 1 \vee d = n)$$

• from each entry/exit each processing unit type reachable

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(1,b) = \#) \Rightarrow \forall f \in \Sigma_F \;.\; \exists H_{\text{route}}(\underline{(1,b)}, \underline{(c,d)}) \in V^* \;.\; G_{\text{mbrns}}(c,d) = f$$

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(m,b) = \#) \Rightarrow \forall f \in \Sigma_F \;.\; \exists H_{\text{route}}(\underline{(m,b)}, \underline{(c,d)}) \in V^* \;.\; G_{\text{mbrns}}(c,d) = f$$

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(a,1) = \#) \Rightarrow \forall f \in \Sigma_F \;.\; \exists H_{\text{route}}(\underline{(a,1)}, \underline{(c,d)}) \in V^* \;.\; G_{\text{mbrns}}(c,d) = f$$

$$\forall (a,b) \in \{1,\dots,m\} \times \{1,\dots,n\} \;.\; \forall (c,d) \in \{1,\dots,m\} \times \{1,\dots,n\} \;. \\ (G_{\text{mbrns}}(a,n) = \#) \Rightarrow \forall f \in \Sigma_F \;.\; \exists H_{\text{route}}(\underline{(a,n)}, \underline{(c,d)}) \in V^* \;.\; G_{\text{mbrns}}(c,d) = f$$

If and only if all constraints are fulfilled, the validity check returns *true*.

Auxiliary function estimating next membrane towards nearest exit
After a particle has passed all processing units, it runs to the nearest exit. To do so, we provide an auxiliary function which detects for all accessible grid membranes (for all nodes in V) the next membrane to be entered in order

to reach the nearest exit. We implement `succ_to_exit` : $V \longrightarrow V$ called by `succ_to_exit`$(\underline{i,j})$ in pseudocode.

$min := \infty$
$x := \varepsilon$
`if` $(i = 1 \vee i = m \vee j = 1 \vee j = n)$`:`
  `succ_to_exit`$(\underline{i,j}) := \underline{(i,j)}$
`else`
  `for` $a \in \{1, \ldots, m\}$`:`
    `if` $(H_{\text{dist}}(\underline{(i,j)}, \underline{(a,1)}) < min)$`:`
      $min := H_{\text{dist}}(\underline{(i,j)}, \underline{(a,1)})$
      $x := H_{\text{route}}(\underline{(i,j)}, \underline{(a,1)})$
    `if` $(H_{\text{dist}}(\underline{(i,j)}, \underline{(a,n)}) < min)$`:`
      $min := H_{\text{dist}}(\underline{(i,j)}, \underline{(a,n)})$
      $x := H_{\text{route}}(\underline{(i,j)}, \underline{(a,n)})$
  `for` $b \in \{1, \ldots, n\}$`:`
    `if` $(H_{\text{dist}}(\underline{(i,j)}, \underline{(1,b)}) < min)$`:`
      $min := H_{\text{dist}}(\underline{(i,j)}, \underline{(1,b)})$
      $x := H_{\text{route}}(\underline{(i,j)}, \underline{(1,b)})$
    `if` $(H_{\text{dist}}(\underline{(i,j)}, \underline{(m,b)}) < min)$`:`
      $min := H_{\text{dist}}(\underline{(i,j)}, \underline{(m,b)})$
      $x := H_{\text{route}}(\underline{(i,j)}, \underline{(m,b)})$
  `if` $(x = \varepsilon)$`:`
    `succ_to_exit`$(\underline{i,j}) := \underline{(i,j)}$
  `else`
    `succ_to_exit`$(\underline{i,j}) := \mathit{prefix}(x)$

Function estimating next membrane towards next processing unit
Analogously, we make available an auxiliary function `succ_to_proc_unit` : $V \times \Sigma_F \longrightarrow V$ which detects for all accessible grid membranes (for all nodes in V) the next membrane to be entered in order to reach the nearest processing unit from type $f \in F$. The function is called by `succ_to_proc_unit`$(\underline{(i,j)}, f)$.

$min := \infty$
$x := \varepsilon$
`if` $(G_{\text{mbrns}}(i,j) = f)$`:`
  `succ_to_proc_unit`$(\underline{(i,j)}, f) := \underline{(i,j)}$
`else`
  `for` $a \in \{1, \ldots, m\}$`:`
    `for` $b \in \{1, \ldots, n\}$`:`
      `if` $((G_{\text{mbrns}}(a,b) = f) \wedge (H_{\text{dist}}(\underline{(i,j)}, \underline{(a,b)}) < min))$`:`
        $min := H_{\text{dist}}(\underline{(i,j)}, \underline{(a,b)})$
        $x := H_{\text{route}}(\underline{(i,j)}, \underline{(a,b)})$
  `if` $(x = \varepsilon)$`:`
    `succ_to_proc_unit`$(\underline{(i,j)}, f) := \underline{(i,j)}$
  `else`
    `succ_to_proc_unit`$(\underline{(i,j)}, f) := \mathit{prefix}(x)$

3.4 Passing the particles throughout the grid for estimation of total overall execution time

The fitness of a grid together with its initial placement of particles is expressed by the overall execution time which means the number of time steps necessary to process all particles and run them outwards the grid. The formal description of the fitness evaluation is based on a global clock counting the number of time steps on the one hand and the successive progression of the P system's *configuration* on the other. Let $Q = \{((i,j),p,f) \mid i \in \{1,\ldots,m\} \wedge j \in \{1,\ldots,n\} \wedge p \in \Sigma_p \wedge f \in \Sigma_F^*\} \longrightarrow \mathbb{N}$ be an arbitrary multiset of particles located within the membrane at grid position (i,j). We capture a configuration of $\Pi_\square$ at time t by a triple $(Q_{\text{processing}}, Q_{\text{migratable}}, t)$ in which $t \in \mathbb{N} \cup \{\infty\}$ marks a point in time. $Q_{\text{processing}} : V \longrightarrow (\text{supp}(Q) \times \mathbb{N} \longrightarrow \mathbb{N})$ assigns to each grid position (i,j) a multiset of particles present in this membrane and being processed. Along with insertion of a particle into $Q_{\text{processing}}$, a time stamp is assigned. This is necessary in order to decide when the processing is over. In addition, $Q_{\text{migratable}} : V \longrightarrow Q$ contains for each grid position all particles locally processed and ready to migrate to the adjacent membrane as soon as possible. The fitness function $\texttt{fitness} : (\Pi_\square) \mapsto t$ determines the number of time steps to pass all particles up to the finally empty grid. To this end, the global clock starts with 0, the initial configuration is set up, and afterwards a loop becomes iterated in which the configuration is updated and the number of elapsed time steps increased by 1.

$t := 0$
for $a \in \{1,\ldots,m\}$:
  for $b \in \{1,\ldots,n\}$:
    $Q_{\text{processing}}(a,b) := \emptyset$
    $Q_{\text{migratable}}(a,b) := \emptyset$
    for $(\underline{(i,j)},p,f) \in P$:
      if $\overline{((a=i) \wedge (b=j))}$:
        $Q_{\text{processing}}(a,b) := Q_{\text{processing}}(a,b) \cup \{((\underline{(i,j)},p,f),t)\}$

while $\left(\sum_{b=1}^{n}\sum_{a=1}^{m}(|Q_{\text{processing}}(a,b)| + |Q_{\text{migratable}}(a,b)|) > 0\right)$:
  $t := t+1$
  for $a \in \{1,\ldots,m\}$:
    for $b \in \{1,\ldots,n\}$:
      for $((\underline{(i,j)},p,f),\tau) \in Q_{\text{processing}}(a,b)$:
        if $((a=i) \wedge (b=j))$:
          if $(t \geq \tau + G_{\text{durat}}(a,b))$:
            $Q_{\text{processing}}(a,b) := Q_{\text{processing}}(a,b) \setminus \{((\underline{(i,j)},p,f),\tau)\}$
            $Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \cup \{(\underline{(i,j)},p,f)\}$
          for $(\underline{(i,j)},p,f) \in Q_{\text{migratable}}(a,b)$:
            if $\overline{(G_{\text{mbrns}}(a,b) = \#)}$:
              if $(f = \varepsilon)$:
                if $((i=1) \vee (i=m) \vee (j=1) \vee (j=n))$:

$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)}, p.f)\}$
elseif $(|Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)})| +$
$Q_{\text{migratable}}(\texttt{succ\_to\_exit}\underline{(i,j)})| < G_{\text{capac}}(\texttt{succ\_to\_exit}\underline{(i,j)}))$:
$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)}, p, f)\}$
$Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)}) := Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)})$
$\cup \{((\texttt{succ\_to\_exit}\underline{(i,j)}, p, f), t)\}$
if $(G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge F(G_{\text{mbrns}}(a,b)) = \star)$:
if $(f = \varepsilon)$:
if $(|Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)})| +$
$Q_{\text{migratable}}(\texttt{succ\_to\_exit}\underline{(i,j)})| < G_{\text{capac}}(\texttt{succ\_to\_exit}\underline{(i,j)}))$:
$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)}, p, f)\}$
$Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)}) := Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)})$
$\cup \{((\texttt{succ\_to\_exit}\underline{(i,j)}, p, f), t)\}$
elseif $(|Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))| +$
$Q_{\text{migratable}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))| <$
$G_{\text{capac}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))))$:
$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)}, p, f)\}$
$Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))) :=$
$Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))) \cup$
$\{((\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f)), p, f \ominus \mathit{prefix}(f)), t)\}$
if $(G_{\text{mbrns}}(a,b) \in \Sigma_F \wedge F(G_{\text{mbrns}}(a,b)) \neq \star)$:
if $(p = \mathit{prefix}(F(G_{\text{mbrns}}(a,b))))$:
$q := F(G_{\text{mbrns}}(a,b)) \ominus \mathit{prefix}(F(G_{\text{mbrns}}(a,b))) \ominus$
$\mathit{suffix}(F(G_{\text{mbrns}}(a,b)))$
$\mathit{partner} := \mathit{false}$
if $((\underline{(i,j+1)}, q, G_{\text{mbrns}}(a,b+1)) \in Q_{\text{migratable}}(a,b+1))$:
$\mathit{partner} := \mathit{true}$
$Q_{\text{migratable}}(a,b+1)) := Q_{\text{migratable}}(a,b+1)) \setminus$
$\{(\underline{(i,j+1)}, q, G_{\text{mbrns}}(a,b+1))\}$
elseif $((\underline{(i,j-1)}, q, G_{\text{mbrns}}(a,b-1)) \in Q_{\text{migratable}}(a,b-1))$:
$\mathit{partner} := \mathit{true}$
$Q_{\text{migratable}}(a,b-1)) := Q_{\text{migratable}}(a,b-1)) \setminus$
$\{(\underline{(i,j-1)}, q, G_{\text{mbrns}}(a,b-1))\}$
elseif $((\underline{(i+1,j)}, q, G_{\text{mbrns}}(a+1,b)) \in Q_{\text{migratable}}(a+1,b))$:
$\mathit{partner} := \mathit{true}$
$Q_{\text{migratable}}(a+1,b)) := Q_{\text{migratable}}(a+1,b)) \setminus$
$\{(\underline{(i+1,j)}, q, G_{\text{mbrns}}(a+1,b))\}$
elseif $((\underline{(i-1,j)}, q, G_{\text{mbrns}}(a-1,b)) \in Q_{\text{migratable}}(a-1,b))$:
$\mathit{partner} := \mathit{true}$
$Q_{\text{migratable}}(a-1,b)) := Q_{\text{migratable}}(a-1,b)) \setminus$
$\{(\underline{(i-1,j)}, q, G_{\text{mbrns}}(a-1,b))\}$
if $(\mathit{partner} = \mathit{true})$:
if $(f = \varepsilon)$:
if $(|Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)})| +$
$|Q_{\text{migratable}}(\texttt{succ\_to\_exit}\underline{(i,j)})| < G_{\text{capac}}(\texttt{succ\_to\_exit}\underline{(i,j)}))$:

$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)},p,f)\}$
$Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)}) :=$
$Q_{\text{processing}}(\texttt{succ\_to\_exit}\underline{(i,j)}) \cup$
$\{((\texttt{succ\_to\_exit}\underline{(i,j)}, \mathit{suffix}(F(G_{\text{mbrns}}(a,b))), f \ominus \mathit{prefix}(f)),t)\}$
elseif $(|Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f)))| +$
$|Q_{\text{migratable}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f)))| <$
$G_{\text{capac}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f)))):$
$Q_{\text{migratable}}(a,b) := Q_{\text{migratable}}(a,b) \setminus \{(\underline{(i,j)},p,f)\}$
$Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))) :=$
$Q_{\text{processing}}(\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f))) \cup$
$\{((\texttt{succ\_to\_proc\_unit}(\underline{(i,j)}, \mathit{prefix}(f)), \mathit{suffix}(F(G_{\text{mbrns}}(a,b))), f \ominus \mathit{prefix}(f)),t)\}$
$\texttt{fitness}(\Pi^0_\square) := t$

We are aware of unlikely but potentially possible situations in which the number of time steps will infinitely grow without termination. This can happen due to two reasons: A circular path of membranes whose capacity is exhausted might cause a deadlock. Moreover, in case of utilisation of processing units assembling two particles into a resulting product, some particles could stuck in a membrane but their counterpart is permanently absent. These particles are unable to get completely processed. We can cope with both situations by inspection of $Q_{\text{processing}}$ and $Q_{\text{migratable}}$ over time. If there is no modification of both matrices over a long period of time steps, the fitness evaluation terminates by return of ∞.

3.5 Artificial evolution

For the artificial evolution, we induce a list (population) of grid exploring P systems ($\Pi^i_\square$) in which the index i identifies the individuals. After the initialisation, the evolution loop drives the generations, see pseudocode:

$g := 0;\ i := 1;\ \phi_0 := \texttt{fitness}(\Pi^0_\square)$
while$(i <$ `population_size`$)$:
  $\Pi^i_\square := \texttt{duplicate}(\Pi^0_\square)$
  $\texttt{mutate}(\Pi^i_\square)$
  if $(\texttt{validity\_check}(m,n,G_{\text{mbrns}},H_{\text{dist}},H_{\text{route}})_i)$:
    $i := i + 1$
while$(g <$ `max_number_of_generations`$)$:
  $\phi_i := \texttt{fitness}(\Pi^i_\square) \quad \forall i \in \{1,\ldots,$ `population_size`$\}$
  `sort_by_fitness()`
  $k := \lfloor 0.8 \cdot$ `population_size`$\rfloor$
  $\texttt{remove}(\Pi^j_\square) \quad \forall j \in \{k,\ldots,$ `population_size`$\}$
  $j := 1$
  while $(j <$ `population_size`$)$:
    $\Pi^j_\square := \texttt{duplicate}(\Pi^{\texttt{random}([0..k-1])}_\square)$
    $\texttt{mutate}(\Pi^j_\square)$
    if $(\texttt{validity\_check}(m,n,G_{\text{mbrns}},H_{\text{dist}},H_{\text{route}})_i)$:
      $j := j + 1$
  $g := g + 1$

For duplication of an individual, we prepare the function `duplicate`. `mutate` randomly decides whether a recombination or a shift occurs and carries out the corresponding mutation. `remove` eliminates an individual from the list. `sort_by_fitness` permutates the list of individuals in ascending fitness order.

4 Case Studies

4.1 Cell signalling cascades

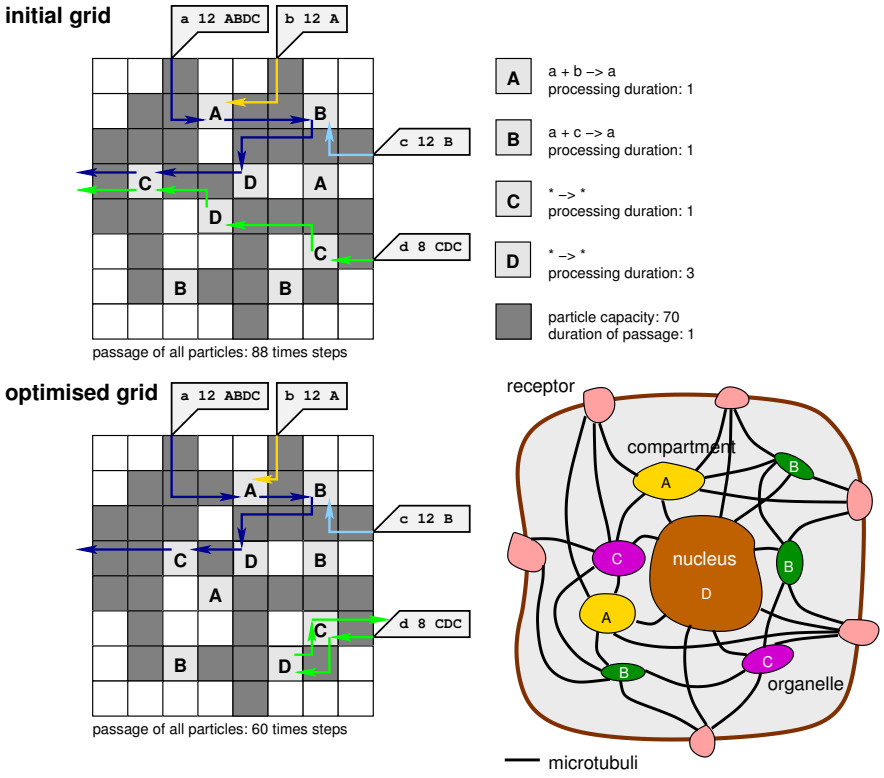


Fig. 4. Signalling in an eucaryotic cell. Signalling molecules `a`, `b`, `c`, and `d` reach receptors at the outer cell membrane. From there, they enter the cell (for instance by endocytosis) passing through a signalling cascade. Transported via microtubuli of the cytoskeleton, they become successively processed within compartments and organelles embedded into the cytosol. The initial grid predefines an abstracted cell structure which exhibits a fitness of 88 time steps to process all signalling molecules in the desired manner. Within `A` and `B`, complex formations are carried out incorporating `b` and `c` into `a`. `D` represents the nucleus for gene expression. The resulting cell response leaves the cell via `C`. There are two signalling pathways. One of them is initiated by `a`, the second one by `d`. Evolutionary structural optimisation improves the fitness to 60 time steps.

4.2 Manufacturing in a cabinet maker's workshop

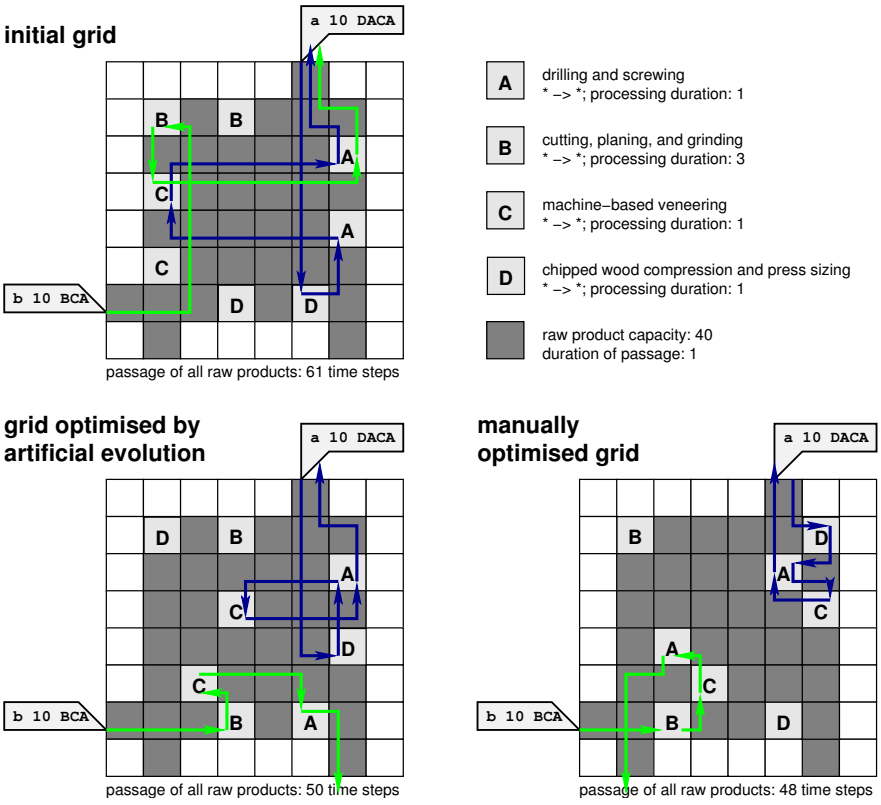


Fig. 5. Cascaded processing of raw products in a cabinet maker's workshop. Raw products called `a` and `b` have been delivered at different entries/exits of a factory work floor depicted as grid. Processing units available in multiple copies allow for completion of a specific work step. We distinguish four types of machines: `A` (drilling, screwing), `B` (cutting, planing, grinding), `C` (veneering), and `D` (press sizing). Raw products `a` require the processing sequence `DACA` while those called `b` need to pass `BCA`. The initial grid arranges all processing units at the outer walls. By evolutionary optimisation, we observe a self-organisation of so-called *production islands*, clusters of processing units corresponding to the individual workflows for both product lines. This diminishs the overall execution time from 61 to 50 time steps. A manual optimisation reveals a best possible result of 48 time steps. From a variety of independently conducted optimisation studies starting from the same initial grid, we obtained numerous topological arrangements. Interestingly, they achieved similar fitness values ranging from 50 to 52 which confirms the observation that artificial evolution typically ends up with results close to but away from the global optimum.

4.3 Alternative supermarket

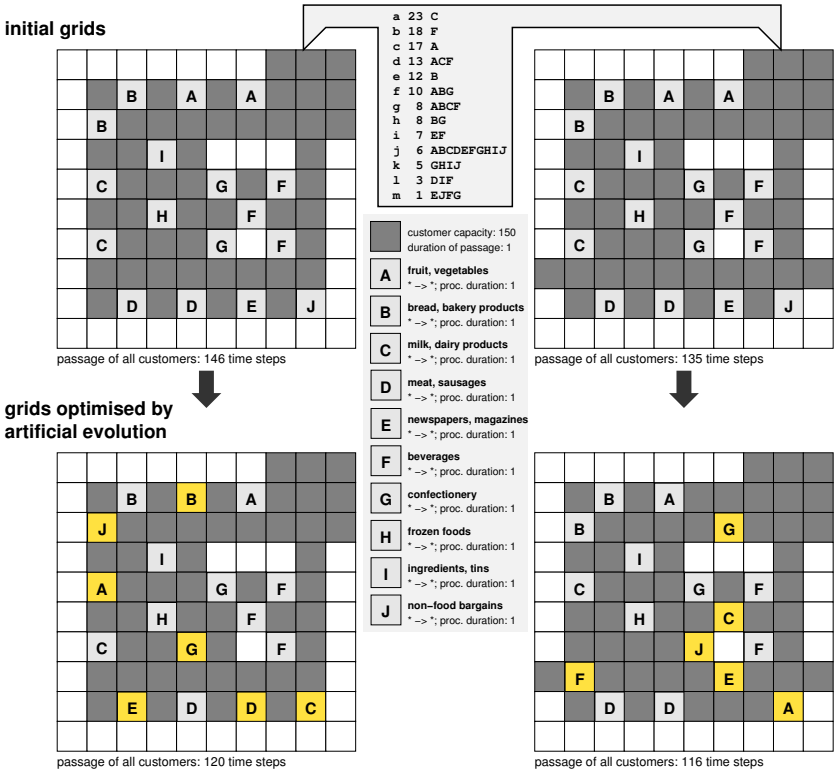


Fig. 6. Upper left grid: Initial placement of product groups in a typical German supermarket with *one central entry/exit*, customer routes in anticlockwise orientation and strictly separated classes of goods. In the example study, 131 customers reflecting frequently observed selections of product combinations visit the supermarket which requires 146 time steps in total. **Lower left grid:** After artificial evolution, a rearrangement of product placement reveals clusters of complementary products like bread/breakfast cereals (**B**) with bargains (**J**) or frozen products (**H**) with confectionery (**G**). Some products like fruit/vegetables (**A**) are offered at different places. For the sample customers, the overall duration has been decreased to 120 time steps. **Upper right grid:** The same initial placement of product groups and the same setting of customers like in the upper left grid is considered but now the supermarket possess *two additional exits*. This action on its own without further optimisation succeeds by a corresponding fitness of 135 time steps. **Lower right grid:** The fitness can further improve by artificial evolution finally resulting in 116 time steps. To this end, the original ample straight-forward pathes have been replaced by more or less local round tours preventing customers from crossing most parts of the supermarket. Especially products in demand near the new exits can effectively accelerate the shopping process for many customers in hurry. By leaving the supermarket earlier with all desired products, they less intensively hamper exhaustive shoppers which in turn can also finalise faster.

5 Discussion and Conclusions

The concept of topological grid optimisation by self-organising dynamical structures using grid-exploring P systems opens one more field of broad applications of membrane-based computing inspired by *bionics*. Interestingly, formalisation of modifiable spatial structures mainly exploits algebraic elements along with their nested composition. Dynamics on that is typically driven by algorithmic denotations in an imperatively or rule-controlled manner rather than prone to explicite formulas or closed analytical terms. We believe that the inherent complexity of structural dynamics suggests an incorporation of algorithmical components into corresponding P systems. In this paper, we exemplified this idea by means of artificial evolution which includes an algorithm for fitness evaluation. Its parameterisation follows common and empirically reliable assumptions. A population size of 50 grids seems to be large enough to create a sufficient variety in which 80% survive for the next generation. No improvement of the best fitness for 100 generations implies termination of the artificial evolution. This turns out to be enough to enable an effective *fine tuning* that can lead to grid structures with improved quality. Dedicated formation of *processing islands* is an example for this. We obtained the best experimental results by utilisation of both evolutionary operators (recombination, shift) in parity and by permanent survival of the initial grid within the population. Unequivocally, the success of artificial evolution also depends on predefinition of a suitable initial grid with a balanced degree of freedom. A free version of the software implemented in JavaScript is available at `http://www-user.tu-cottbus.de/~weberlea/gridtool/` and from the authors upon request. This version supports processing units in passage mode as well as incorporation in assembly mode. Further work is aimed at making the fitness evaluation "smarter". Instead of simply running each particle to the *nearest* destination of the desired type, the underlying algorithm could interpret temporary delays, particle jam, or overcharged processing units which ends up with flexible generation of alternative bypass routes individually for each particle.

References

1. S. Bagchi. Self-adaptive and reconfigurable distributed computing systems. *Applied Soft Computing* **12**:3023-3033, 2012
2. F. Bernardini, M. Gheorghe, N. Krasnogor, J.L. Giavitto. On Self-assembly in Population P Systems. *Lecture Notes in Computer Science* **3699**:46-57, 2005
3. J.D. Bewley, M. Black. *Seeds. Physiology of Development and Germination.* Springer, 1994
4. J. Buhl, J.L. Deneubourg, A. Grimal, G. Theraulaz. Self-organized digging activity in ant colonies. *Behavioral Ecology and Sociobiology* **58**:9-17, 2005
5. S. Camazine, J.L. Deneubourg, N.R. Franks, J. Sneyd, G. Theraulaz, E. Bonabeau. *Self-Organization in Biological Systems.* Princeton University Press, 2003
6. Z. Fangwei, J. Huang, M. Meagher. The Introduction and Design of a New Form of Supermarket: Smart Market. *Information Engineering and Electronic Commerce* (IEEC09), pp. 608-611, IEEE press, 2009

7. Y.C. Fung. *Biomechanics: mechanical properties of living tissues.* Springer, 1993
8. M. Gheorghe, G. Păun. Computing by Self-Assembly: DNA Molecules, Polyominoes, Cells. *Systems Self-Assembly: Multidisciplinary Snapshots* Studies in Multidisciplinarity 5:49-78, Elsevier, 2008
9. D. Guidolin, E. Crivellato, D. Ribatti. The "self-similarity logic" applied to the development of the vascular system. *Developmental Biology* **351**:156-162, 2011
10. J.T. Hancock. *Cell Signalling.* Oxford University Press, 2010
11. T. Hinze, K. Grützmann, B. Höckner, Pe. Sauer, S. Hayat. Categorised Counting Mediated by Blotting Membrane Systems for Particle-based Data Mining and Numerical Algorithms. *Lecture Notes in Computer Science* **8961**:241-257, 2014
12. T. Hinze, J. Behre, C. Bodenstein, G. Escuela, G. Grünert, P. Hofstedt, Pe. Sauer, S. Hayat, P. Dittrich. Membrane Systems and Tools Combining Dynamical Structures with Reaction Kinetics for Applications in Chronobiology. In P. Frisco, M. Gheorghe, M.J. Perez-Jimenez (Eds.), *Applications of Membrane Computing in Systems and Synthetic Biology.*, Series Emergence, Complexity, and Computation, Vol. 7, pp. 133-173, Springer, 2014
13. T. Hinze, K. Kirkici, Pa. Sauer, Pe. Sauer, J. Behre. Membrane Computing Meets Temperature: A Thermoreceptor Model as Molecular Slide Rule with Evolutionary Potential. *Lecture Notes in Computer Science* **9504**:215-235, 2015
14. D. Ivanov, A. Dolgui, B. Sokolov, F. Werner, M. Ivanova. A dynamic model and an algorithm for short-term supply chain scheduling in the smart factory industry 4.0. *International Journal of Production Research* **54(2)**:386-402, 2016
15. B.N. Kholodenko. Cell-signalling dynamics in time and space. *Nature Reviews Molecular Cell Biology* **7**:165-176, 2006
16. H. Kurz, K. Sandau, J. Wilting, B. Christ. Blood Vessel Growth: Mathematical Analysis and Computer Simulation, Fractality, and Optimality. In C.D. Little et al. (eds.) *Vascular Morphogenesis.* Birkhäuser, 1998
17. C. Martin-Vide, G. Păun, J. Pazos, A. Rodriguez-Paton. Tissue P Systems. *Theoretical Computer Science* **296**:295-326, 2003
18. M.A. Meyers, P.Y. Chen, A.Y.M. Lin, Y. Seki. Biological materials: Structure and mechanical properties. *Progress in Materials Science* **53**:1-206, Elsevier, 2007
19. P. Miller. *The Smart Swarm.* Avery Publishing Group, 2010
20. G. Păun. *Membrane Computing: An Introduction.* Springer, 2002
21. P. Prusinkiewicz, A. Lindenmayer. *The Algorithmic Beauty of Plants.* Springer, 1990
22. J. Reichold, M. Stampanoni, A.L. Keller, A. Buck, P. Jenny, B. Weber. Vascular graph model to simulate the cerebral blood flow in realistic vascular networks. *Journal of Cerebral Blood Flow and Metabolism* **29**:1429-1443, 2009
23. N.C. Rivron, J. Rouwkema, R. Truckenmüller, M. Karperien, J. de Boer, A. van Blitterswijk. Tissue assembly and organization: Developmental mechanisms in microfabricated tissues. *Biomaterials* **30**:4851-4858, 2009
24. F. Ruan, D. Chen. Based on RFID and NFC Technology Retail Chain Supermarket Mobile Checkout Mode Research. The International Conference on Artificial Intelligence and Software Engineering (ICAISE 2013). *Atlantis Press*, 2013
25. R. Sedgewick, K. Wayne. *Algorithms.* Fourth Edition, Addison-Wesley, 2011
26. I. Stamatopoulou, P. Kefalas, M. Gheorghe. $OPERAS_{CC}$: An Instance of a Formal Framework for MAS Modeling Based on Population P Systems. *Lecture Notes in Computer Science* **4860**:438-452, 2007
27. A. Szymczak, A. Stillman, A. Tannenbaum, K. Mischaikow. Coronary vessel trees from 3D imagery. *Medical Image Analysis* **10**:548-559, 2006

Agent-Based Simulation of Kernel P Systems with Division Rules Using FLAME

Raluca Lefticaru[1,2], Luis F. Macías-Ramos[3],
Ionuţ Mihai Niculescu[4], Laurenţiu Mierlă[2,4]

[1] CENTRIC, Sheffield Hallam University,
153 Arundel Street, Sheffield, S1 2NU, UK
[2] Department of Computer Science, University of Bucharest,
Str. Academiei nr. 14, 010014, Bucharest, Romania
raluca.lefticaru@fmi.unibuc.ro
[3] Research Group on Natural Computing, Dept. Computer Science and Artificial Intelligence, University of Seville Avda. Reina Mercedes S/N, 41012, Sevilla, Spain
lfmaciasr@us.es
[4] Department of Mathematics and Computer Science, University of Pitesti,
Str. Targu din Vale 1, 110040, Pitesti, Romania
{ionutmihainiculescu,laurentiu.mierla}@gmail.com

Abstract. Kernel P systems (or *kP systems*) bring together relevant features from several P systems flavours into a unified *kernel* model which allows solving complex problems using a straightforward *code programming approach*. `kPWorkbench` is a software suite enabling specification, parsing and simulation of kP systems models defined in the kernel P–Lingua (or `kP-Lingua`) programming language. It has been shown that any computation of a kP system involving only rewriting and communication rules can be simulated by a family of Communicating Stream X-Machines (or *CSXM*), which are the core of `FLAME` *agent based simulation environment*. Following this, kPWorkbench enables translating kP-Lingua specifications into FLAME models, which can be simulated in a sequential or parallel (MPI based) way by using the FLAME framework. Moreover, `FLAME GPU` framework enables efficient simulation of CSXM on CUDA enabled GPGPU devices. In this paper we present an extension of kPWorkbench framework to generate FLAME models from kP–Lingua specifications including structural rules; and consider translation of FLAME specifications into FLAME GPU models. Also, we conduct a performance evaluation regarding simulation of equivalent kP systems and CSXM models in kPWorkbench and FLAME respectively.
Keywords: Membrane computing; kernel P systems; communicating stream X-machines; agent-based simulation.

1 Introduction

Membrane computing is, to date, the youngest Natural Computing discipline. It was introduced in 1998 by *Gheorghe Păun*, as a paradigm which addresses models taking inspiration form the structure and functioning of cells present in

living beings, considering such cells as living entities themselves able to process and generate information.

Computing devices of membrane computing are called membrane systems or P systems [22]. Basic ingredients of a P system are (i) *a membrane structure*, consisting in a set of *regions* delimited by *membranes*; and (ii) *multisets of objects* placed within the regions. Objects may be transformed according to some *evolution rules*, which are applied in a non-deterministic maximally parallel way (emulating how chemical reactions take place among compounds). To emulate *cell membrane permeability*, *evolution rules* can transform existing objects within a region and, additionally, *transfer* them among *adjacent* regions – objects pass through the membrane separating the regions.

Basically, there are three ways to categorize membrane systems: *cell-like* P systems, *tissue–like* P systems and *neural-like* P systems. In cell-like P systems, membranes are arranged in a hierarchical way, inspired by the inner structure of the biological cells. In tissue-like P systems, cells are set in nodes of a directed graph, inspired from the cell inter-communication in tissues. Similarly, in neural-like P systems, cells are arranged in nodes of a directed graph, taking inspiration from the way in which neurons exchange information by the transmission of electrical impulses (spikes) along axons. Neither tissue-like nor neural-like consider the possibility of cells containing inner compartments, that is, in such variants cells are elemental compartments.

Kernel P systems (or *kP systems*) [6] are a novel variant of membrane systems aiming to bring together relevant features from several P systems flavours into a unified *kernel* model which allows solving complex problems using a straightforward *code programming approach*. In particular, a kP system model is defined by placing *compartment type instances* in the nodes of a *dynamic graph*. Each type represents a kind of *elemental compartment* which is associated with a sequence of *rule blocks*. Following this, each rule block is defined by both a *set of guarded evolution rules* and an *execution strategy*. A guarded rule associated to a compartment type is an extension of a classic evolution rule where a new syntactical element, the *guard*, is added. A guard is a logical condition over the multiplicity of objects belonging to the multiset associated to any instance of the corresponding type. Evolution rules may be either *rewriting and communication* rules or *structure changing rules* (cell division, cell dissolution, link creation or link destruction rules). A given compartment instance executes its rule blocks sequentially, with applicable rules to be executed for each block according to the its own execution strategy.

P–Lingua framework [27], possibly the most widely known simulation software for membrane computing, provides support for a reduced version of kP systems, known as *simple kernel P systems*. As a separate effort from that of P–Lingua, a brand new software project, known as `kPWorkbench` [7, 28], was created aiming to provide full support for kP systems as well as advanced model checking features. kPWorkbench allows the specification of kP systems in the kernel P–Lingua (or `kP-Lingua`) programming language, which share some similarities with the original P–Lingua one. kPWorkbench framework features a native sim-

ulator, allowing the simulation of kP system models written in kP–Lingua. On the other hand, kPWorkbench's model checking environment permits the formal verification of kernel P system models.

Regarding parallel simulation of kP systems, in [20] it was shown that any computation of a kP system involving only rewriting and communication evolution rules can be simulated by a family of *Communicating Stream X-Machines* (or *CSXM*), which are extended forms of state machines, having memory and processing functions. CSXM can be efficiently simulated in a parallel way by means of two template-based software frameworks called `FLAME` (*Flexible Large-Scale Agent Modelling Environment*) [29] and `FLAME GPU` [30]. FLAME allows MPI [31] based efficient simulation of CSXM models written in the `FLAME` agent-based specification language, while FLAME GPU allows *CUDA (Compute Unified Device Architecture)* [12, 19, 21, 32] based efficient simulation of CSXM models written in the `FLAME GPU` specification language, an extension/variant of the FLAME one. Both FLAME and FLAME GPU have been used in several experiments, which were performed on *High Performance Computing (HPC)* platforms due to the scale of the associated models and, subsequently, resource-intensive simulation tasks. Some examples include modelling oxygen-responsive transcription factors in *Escherichia coli* [1] or the complex cellular tissue simulation [24].

Following this, in [8] kPWorkbench was extended to provide automated translation from kP systems models written in kP–Lingua into CSXM models written in FLAME specification language. This translation addressed kP systems involving only rewriting and communication evolution rules, with the transformation of systems involving structural rules left as an open issue. Moreover, no support for automated translation to FLAME GPU was provided either.

In this work, we take a step forward regarding the aforementioned results tackling new challenges. Firstly, we address an extension of the kPWorkbench framework to generate FLAME models from kP–Lingua specifications including structural rules such as division and dissolution rules. Secondly, we address the translation of FLAME specifications into FLAME GPU models. Finally, we conduct a performance study regarding the simulation of equivalent kP systems and CSXM models in kPWorkbench and FLAME (serial and parallel mode) respectively. This is conducted following the trail of [2] and [13]. In particular, [13] presents a first approach of implementing the pulse generator model in FLAME GPU [13], conducting a performance comparison with FLAME. Remarkably, the FLAME GPU model used was manually translated, since there was no public available tool to automate the conversion.

This paper is structured as follows. Section 2 outlines previous related work. Section 3 introduces the theoretical background. Section 4 presents our modelling approach in FLAME, while Section 5 presents the possible ways of extending it to FLAME GPU. A case study illustrating the cited approach is discussed in Section 6. Finally, conclusions and further work are drawn in Section 7.

2 Related Work

In this Section, we briefly outline the state-of-the art of parallel simulation of P systems on High Performance Computing (HPC) platforms.

Both P–Lingua and kPWorkbench, as a vast majority of software tools for membrane computing, implement the simulation algorithms in a *sequential* way. This effectively neglects the inherent parallelism of P systems, and leads to non-efficient simulations from the computational complexity point of view. Fortunately, an increasing variety of simulators specially intended to run on *massively-parallel platforms* have been developed along the years. Such HPC platforms include *Field Programmable Gate Array circuits (FPGAs)* [23], *microcontrollers* [9], *computer clusters* [3, 4, 26] and *General–Purpose Graphic Processing Unit (GPGPU)* devices.

In particular, GPGPU hardware comprises a very affordable technology, providing in a *single* device *hundreds* of *massively parallel processors* supporting several *thousand* of concurrent *threads*. To date, many general purpose applications have been successfully migrated to GPGPU platforms, showing good *speed-ups* compared to their corresponding sequential versions. Two are the main programming models enabling software development oriented to GPGPUs. On the one hand, *CUDA (Compute Unified Device Architecture)* programming model, [12, 19, 21, 32] and on the other hand, *Open Computing Language (OpenCL)* framework [18, 25, 33].

An updated exhaustive list of parallel simulators for P systems regarding the aforementioned approaches can be consulted in [14], with the reader also encouraged to check [5] and [15], from where an encyclopaedic knowledge can be obtained. Finally, a couple of surveys summarising the topic can be consulted in [16, 17].

With respect to parallel simulation of kernel P systems, to the best knowledge of the authors, there are only a few related software applications due to the novelty of the model. On the one hand, a parallel implementation on GPGPU architectures using CUDA is reported in [11]. On the other hand, regarding the simulation of kP systems with agents, FLAME and FLAME GPU simulation platforms are detailed in [2, 13, 20], as we discussed above.

3 Background

This Section gives the basic definitions and major results regarding kernel P systems and communicating stream X-machines, following largely from [6, 20]. For this, we will assume that the reader is familiar with usual notations from formal languages, membrane computing and finite automata domains and refer to [6, 10, 20] for further technical details and examples.

We first begin recalling the formal definition of kernel P systems (or kP systems).

Definition 1. *A* kP system *of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where*

- *A is a finite set of elements called objects;*
- *μ defines the membrane structure, which is a graph, (V, E), where V are vertices representing components (compartments), and E edges, i. e., links between components;*
- *$C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, is a compartment of the system consisting of a* compartment type*, t_i, from a set T and an initial multiset, $w_{i,0}$ over A; the type $t_i = (R_i, \rho_i)$ consists of a set of evolution rules, R_i, and an execution strategy, ρ_i;*
- *i_0 is the output compartment where the result is obtained.*

A kernel P system can have several compartment instances of the same type: while they share the same set of rules and execution strategies, they may have different multiset of objects at different computation steps and different neighbours according to the graph relation specified by (V, E). Within the kernel P systems framework, the following kinds of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \longrightarrow y\{g\}$, where $x \in A^+$ and y represents a multiset of objects over A^* with potential different compartment type targets (each symbol from the right side can be sent to a different compartment, specified by its type; if more compartments of the same type are linked to the current compartment, then one is randomly chosen).
  Compared to cell-like P systems, the targets in kP systems are the type of compartments to which the objects will be sent, not particular instances. Also, for kP systems, complex guards can be represented, using multisets over A with relational and Boolean operators.
  For example, rule $r : ab \longrightarrow bc\{\geq a^3 \wedge < b^2\}$ can be applied if and only if the current multiset includes the left hand side of r, i. e., ab and the guard holds: the current multiset has at least 3 a's and less then 2 b's.
- *structure changing* rules: membrane division, membrane dissolution, link creation and link destruction rules, which all may also incorporate complex guards and that are covered in detail in [6].

As we stated above, besides of a set of evolution rules, each compartment type in a kP system has an associated execution strategy. Execution strategies offer a lot of flexibility to the kP system designer, as the rules corresponding to a compartment can be grouped in several blocks, every block having one of the following strategies:

- *sequential*: if the current rule is applicable, then it is executed, advancing towards the next rule/block of rules; otherwise, the execution terminates;
- *choice*: 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 whole block is simply skipped;
- *arbitrary*: the rules from the block can be executed zero or more times by nondeterministically choosing any of the applicable rules;
- *maximal parallel*: the classic execution mode used in membrane computing.

On the other hand, a stream X-machine is an extended form of finite state machine in which the transitions are labelled by (partial) functions (called *processing functions*) instead of simple symbols. Remarkably, the machine has a memory M, that can be imagined as the domain of the variables of the system to be modelled. The input received by the machine is processed in order: depending on the current state of the machine and the input symbol to be processed, one of the processing functions will read the current input symbol, discard it from the input sequence and produce an output symbol while (possibly) changing the value of the memory and taking the machine to a different state. Finally, if several processing functions can compute the same input, then one of them is randomly chosen (non-determinism). Formal definition of stream X-machines follows:

Definition 2. *A* Stream X-Machine *(SXM for short) is a tuple* $Z = (\Sigma, \Gamma, Q, M, \Phi, F, I, T, m_0)$*, where:*

- Σ *and* Γ *are finite sets called the input alphabet and output alphabet respectively;*
- Q *is the finite set of states;*
- M *is a (possibly) infinite set called memory;*
- Φ *is the type of* Z*, a finite set of function symbols. A basic processing function* $\phi : M \times \Sigma \longrightarrow \Gamma \times M$ *is associated with each function symbol* ϕ*.*
- F *is the (partial) next state function,* $F : Q \longrightarrow 2^Q$*. As for finite automata,* F *is usually described by a state-transition diagram.*
- I *and* T *are the sets of initial and terminal states respectively,* $I \subseteq Q$*,* $T \subseteq Q$*;*
- m_0 *is the initial memory value, where* $m_0 \in M$*;*
- *all the above sets, i. e.,* $\Sigma, \Gamma, Q, M, \Phi, F, I, T$ *, are non-empty.*

Several theoretical frameworks have been developed addressing communicating stream X-machines, that is, concurrent systems where different stream X-machines work in parallel exchanging data via communication channels. In what follows, we recall the one from [10], which is the closest to the the implementation of FLAME, according to [20].

Definition 3. *A Communicating* Stream X-Machine System *(CSXMS for short) with* n *components is a tuple* $S_n = ((Z_i)_{1 \leq i \leq n}, E)$*, where:*

- $Z_i = (\Sigma_i, \Gamma_i, Q_i, M_i, \Phi_i, F_i, I_i, T_i, m_{i,0})$ *is the SXM with number* i*,* $1 \leq i \leq n$*.*
- $E = (e_{ij})_{1 \leq i,j \leq n}$ *is a matrix of order* $n \times n$ *with* $e_{ij} \in \{0, 1\}$ *for* $1 \leq i, j \leq n$*,* $i \neq j$ *and* $e_{ii} = 0$ *for* $1 \leq i \leq n$*.*

The CSXMS works as follows:

- Each individual Communicating SXM (CSXM for short) is a SXM plus an infinite input queue (FIFO structure); the CSXM consumes inputs from its queue.
- An input symbol received from the external environment (also a FIFO structure) will move to the input queue of one CSXM, if it is contained in its input alphabet. If more than one CSXM satisfies such condition, then the symbol will enter the input queue of one of these in a non-deterministic fashion.

– Each pair of CSXMs, say Z_i and Z_j, have two unidirectional communication channels. The communication channel from Z_i to Z_j is enabled if $e_{ij} = 1$ and disabled otherwise
– There exists the possibility for an output symbol produced by a CSXM, say Z_i, to pass to the input queue of another CSXM, say Z_j, providing that the communication channel between them is enabled, and if the symbol is included in the input alphabet of Z_j. If several CSXMs satisfy these conditions one of them is non-deterministic chosen, whereas if none exists the symbol goes to the output environment (also a FIFO structure).

One important result proving the possibility of simulating kP systems with CSXMs is given in the following theorem from [20].

Theorem 1. *For any kP system, $k\Pi$, of degree n and using only rewriting and communication rules, there is a communicating stream X-machine system, S_{n+1}, with $n+1$ components such that for any multiset w computed by $k\Pi$ there is a complete sequence of transitions in S_{n+1} leading to $s(w)$.*

In this Theorem, w is the final configuration of the kP system, $w = (w_1, \ldots, w_n)$, where each w_i represents the final multiset occurring in compartment i. On the other hand, $s(w)$ corresponds to any of the strings obtained by concatenating the symbols occurring in w. Remarkably, the Proof of this Theorem, as shown in [20], suggests the manner in which a FLAME model for a given kP system can be constructed. We will briefly examine this in the next Section.

4 Modelling Kernel P Systems with Structure Changing Rules in FLAME

In this Section, we describe how kernel P systems incorporating structure changing rules can be mapped into FLAME specification language. We start recalling the way in which Communicating Stream X-Machines Systems are defined in that language.

FLAME framework provides an environment for defining communicating agents, specified in an XML format, which contains information regarding their memory variables, name of processing functions, message structures that can be exchanged for communication, etc. FLAME uses an implementation of CSXMs in which: (a) the associated automaton of each CSXM has no loops (this ensures that the execution will end after a finite number of processing functions calls); (b) the CSXMs receive no inputs from the environment – inputs are usually those produced in the previous computation step or the ones defined in the initial configuration file; and (c) the processing function scripts are written in C files.

The input Communicating Stream X-Machines System is defined in XMML format (X-Machine Mark-up Language), which is translated by FLAME into simulation program source code, either in its serial or parallel (MPI) version.

Next, this program can be compiled together with the agent processing functions script files (written in C) by any C/C++ compiler, giving place to simulation executable code can be run, in the case of the parallel version, on High Performance Computing (HPC) platforms.

To take advantage of the aforementioned efficient simulation capability of FLAME, in [8] kPWorkbench was extended to provide automated translation from kP systems models written in kP–Lingua into CSXM models written in FLAME specification language. This translation addressed kP systems involving only rewriting and communication evolution rules, with the transformation of systems involving structural rules left as an open issue. In what follows, we outline the main ideas in which such transformation process relies on (additional details can be found in [8, 13, 20]):

- Each compartment type in the kP system is associated an agent type in FLAME, while each compartment type instance is associated an agent of the corresponding agent type.
- Multisets of objects from each compartment type instance are stored in the corresponding agent memory using, in general, dynamic arrays of complex data types.
- Execution strategies (sequential, choice, arbitrary, maximal parallel) of the compartment types are encapsulated in C functions.
- Communication between compartments is materialized by using FLAME's agents message passing mechanism. In particular, communication among linked compartments is ensured by means of message filtering, while the non-deterministic choice among one of the possible target compartments is ensured by means of non-deterministic message processing.
- Finally, the graph structure of the kP system, which maps the links among compartments, can be stored in a distributed way among each agent memory as a dynamic array containing the identifiers of the agents representing the compartments sharing an active link with the compartment represented by the current agent.

Next, we address how kP systems incorporating structural rules (membrane division, membrane dissolution, link creation, link destruction) can be translated into FLAME. Let us notice that previous experiments regarding kP systems to FLAME translation, such as the ones references above, did not considered kP systems having structural rules. We start describing the process for membrane dissolution, link creation and link destruction, which is quite straightforward compared to that of membrane division:

- Membrane dissolution can be implemented by either (1) extending the agent memory with a flag-type data value storing whether the corresponding compartment is active or has been dissolved; or (2) removing the corresponding FLAME agent, when the membrane dissolution takes place. The first approach should be used when keeping trace of kP system evolution is required and, subsequently, agent deletion is not aimed. In both approaches,

each time that a dissolution takes place, all the agents having connections to the "dissolved" agent have to be notified via messages, in order to update their connection arrays.
– Link creation and link destruction rules can be implemented by adding or removing elements in the connection arrays accordingly.

In what follows we address transformation of kP systems involving division rules. Translation of such rules is the most challenging to implement and, consequently, we devote a specific part of this paper to its study.

4.1 Implementing kP Systems Division Rules in FLAME

Let us recall that, in general, P systems operate by applying rewriting rules defined over multisets of objects associated to the different membranes, in a synchronized non-deterministic maximally parallel way. P systems show a double level of parallelism: a first level comprises parallel application of rules within individual membranes, while a second level comprises all the membranes working simultaneously, that is, in parallel. These features make P systems powerful computing devices. In particular, the double level of parallelism allows a space-time trade-off enabling *the generation of an exponential workspace in polynomial time*. This is usually accomplished by applying iteratively membrane creation rules, such as division rules.

As such, P systems are suitable to tackle relevant real-life problems, usually involving **NP**-complete problems. Moreover, P systems are excellent tools to investigate on the computational complexity boundaries, in particular tackling the **P versus NP** problem. In this way, by studying how the ingredients relative to their syntax and semantics affect to their ability to solve **NP**–complete problems in a feasible way, computational properties, sharper frontiers between efficiency and non-efficiency can be discovered.

In order to take advantage of the full power of P systems, it is required to simulate them on HPC platforms, which can suitably manage the demanding resource requirements of their inherent double parallelism. In the particular case of kernel P systems, this involves efficiently simulating division rules on massively parallel devices. This can be accomplished by transforming the corresponding kP systems models into FLAME specification language.

Regarding this, FLAME inherently favours the possibility of simulating membrane division, since it supports adding new agents during the simulation execution in such a way that all the newly created agents are introduced at the beginning of the next iteration. Nevertheless, several tasks have to be performed to properly simulate membrane division of a given compartment:

– The newly created agent memory has to be initialized with the data corresponding to the multiset of the underlying newly created compartment.
– The connection arrays of both the newly created agent and each agent representing a compartment linked to the original dividing one have to be updated. As such, it is required to implement a mechanism for the newly created agents to be associated unique identifiers.

– This is accomplished by creating in the system a single instance of a new agent type, called the *instance manager*. This agent will receive and process requests of new identifiers from agents representing dividing compartments via message passing. The instance manager will then provide – again via message passing – a new identifier that will be used by the "dividing" agent to initialize the newly created agents and to send a connection array update signal to all its linked agents.

Fig. 1 shows the visual representation of a FLAME model incorporating different kind of translated evolution rules. In particular, this state machine visualisation is automatically drawn by the FLAME editor, based on the kPWorkbench generated model of a kP system solving `SubSetSum`. States are represented with ellipses, processing functions with rectangles, and messages exchanged between agents with green parallelograms.

Comparing the Main and Output compartments in Fig.1, one can check that the Main compartment has additional states and transitions corresponding to the application of division rules, preliminaries for the creation of new membranes, request identifiers or receive identifiers messages.

It is worth pointing out that a FLAME agent structure will vary depending on the execution strategies and blocks from its corresponding kP system compartment. For example, comparing the kP–Lingua specification from Fig. 2 and its corresponding FLAME model in Fig. 1, one can easily identify that each *choice* block has a corresponding processing function and next state in the CSXM. Similarly, the execution order respect of several strategies is reflected in the agent structure. In addition, ramifications may appear for the cases when a structural rule is chosen to be applied.

Following the process outlined above, the existing kP–Lingua to FLAME kPWorkbench translator module has been extended to support translation of division and dissolution rules with the corresponding algorithm included as an Appendix at the end of this paper. The new module has been successfully used to generate FLAME models for instances of `SubSetSum`. Such instances have then been used in the experiments detailed in Section 6.

5 Adapting the Modelling Approach to FLAME GPU

In this Section we briefly discuss some design constraints when translating FLAME models to FLAME GPU and recommended workarounds.

Although FLAME GPU framework is an extension of FLAME, models designed for FLAME are not supported by FLAME GPU. Apart from small differences that could be easily tackled, e.g. using a slightly different XML schema, with different namespace or tag names, there are also important differences in the data types which can be used by each environment.

In order to address these issues, some design guidelines have to be taken into account, as noted in [13], where authors manually translated a model of a pulse generator from FLAME to FLAME GPU, in order to have it implemented in both frameworks and compare their performance with kPWorkbench.

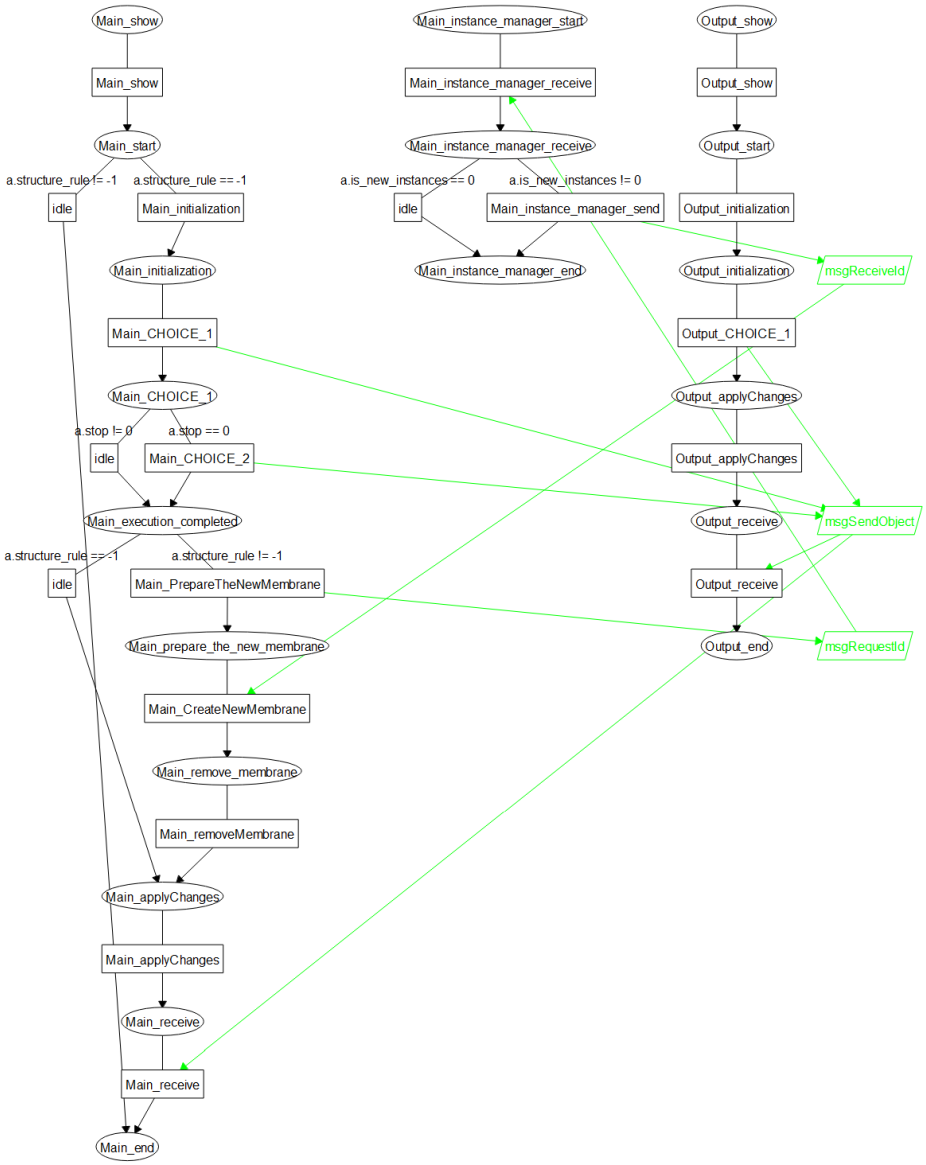


Fig. 1: Graph of the states, functions and messages between agents, corresponding to the FLAME model for Subset Sum problem, having two agent types for the compartments types Main and Output, plus and an additional Main Instance Manager agent in charge of the Main compartment division process (for allocation of new identifiers)

Firstly, memory in FLAME GPU is pre-allocated due to CUDA programming model constraints. As such, agents memory neither support dynamic arrays nor

fixed arrays with complex types. The recommender workaround is the serialization of dynamic arrays (with/without composed objects) which appear in the FLAME models into static arrays of basic types, recommended to have fixed length equal to a power of 2.

Secondly, although in FLAME it is possible to add several new agents in one step, which is useful for example when a membrane is divided into 3 new compartments, in FLAME GPU only one agent is possible to be added per function. Consequently, the recommended workaround implies creating additional functions in the X-machine structure, to add the remaining membranes to be created.

Finally, contrary to FLAME, in FLAME GPU each agent can only create a single message, which clash with communication rules semantics, where multiple compartments may receive different objects. The recommended workaround is to expand the memory space for each message, allowing it to contain data for multiple targets.

6 Case Study: Subset Sum Problem

As previous work on modelling kP systems with FLAME [2, 20] and FLAME GPU [13] addressed models with communication and rewriting rules, in this Section we will illustrate the case of a kP–Lingua model consisting in a kP system family solving `SubSetSum`, which involves division rules as well as other kernel P systems specific features, such as presence of guards plus sequential and choice execution strategies.

The `SubSetSum` problem can be roughly described as: "Given the set $S_n = \{a_1, a_2, \ldots, a_n\}$, is there a subset of S_n, having the sum of elements equal to x?" Regarding to our model, in order to ensure that the computation will continue until all the possible membrane divisions have been applied, we have considered $S_n = \{1, 2, \ldots, n\}$ and the expected sum $x = n(n+1)/2$, which will return the *yes* answer in this case. The kP-Lingua specification considered for $n = 4$ is given in Fig. 2, however corresponding files have been generated for each $n \in \{2, 3, \ldots, 20\}$, a complete folder containing all the files and scripts needed to run the experiments is provided for downloading purposes [5].

The model shown in Fig. 2 is a slightly adapted version from that of [8], and was chosen because of its programming-like structure (sequential blocks) and its simplicity regarding beforehand computation of the number of expected execution steps $(n+1)$ and number of membranes in the last configuration $(2^n + 1)$. Two compartment types are used, named *Main* and *Output*, respectively. The *Main* compartment type contains two choice blocks to (1) generate the *positive answer* when required; and (2) generate the subsets to be checked by applying division rules. The *Output* compartment type take care of (1) controlling the execution by means of a *counter* object; and (2) generating the *negative answer* when required.

[5] `https://www.dropbox.com/sh/dfrxceu4d3d9hhl/AACVTnHyexphjyrJLELNd9-ua?dl=0`

Simplicity of the model eased the experimentation process, since the goal was to conduct non-deterministic experiments but assuring that the computation would end after the same number of steps, with a maximum number of membranes created by division, but different execution traces each time.

```
type Main {
  choice {
    = 10x: a -> {yes} (Output) .
    > 10x: a -> halt .
  }
  choice {
    !r1: a -> [a, r1][x, a, r1] .
    !r2: a -> [a, r2][2x, a, r2] .
    !r3: a -> [a, r3][3x, a, r3] .
    !r4: a -> [a, r4][4x, a, r4] .
  }
}
type Output {
  choice {
    start -> step .
    <5step : step -> 2step .
    <yes & =5step: step -> 2step, no, halt .
  }
}
main {a} (Main) - output {start} (Output) .
```

Fig. 2: kP–Lingua specification for `SubSetSum` (n=4)

In order to assess the performance of the different modelling approaches and implementations for kPWorkbench and FLAME, we conducted several experiments involving different instances of the presented kP system solution to `SubSetSum`, and their translated FLAME counterparts, respectively. With respect to FLAME, besides the serial simulation of the models, which was addressed in previous works like [2, 13, 20], also the parallel MPI based simulation provided by FLAME was considered. Since FLAME does not support MPI simulation in Windows environments, the Sevilla HPC Server [34] `mulhacen` was configured with FLAME and Open MPI [35] to conduct the experiments.

Comparisons were realised between average execution times for: kPWorkbench, FLAME in serial mode and FLAME in parallel version, run with different number of processors, $NP \in \{2, 3, 4\}$, running all on the same machine, a Xeon Server, having 4 cores, Intel i5 Xeon E3-1230V3 @ 3.30GHz, main memory 32 GBytes DDR3 @ 2400Mhz.

For each instance of `SubSetSum`, $n \in \{2, 3, \ldots, 20\}$ and each tool configuration, 3 runs were considered. In each case the user time + system time was recorded and the average total time for these three runs was considered.

n	Compart.	Flame	kPWorkbench	Flame NP 2	Flame NP 3	Flame NP 4
2	4	0.00	0.09	0.00	0.00	0.18
3	8	0.00	0.09	0.00	0.00	0.22
4	16	0.00	0.09	0.00	0.00	0.18
5	32	0.00	0.10	0.00	0.00	0.23
6	64	0.01	0.11	0.19	0.19	0.21
7	128	0.02	0.14	0.21	0.25	0.28
8	256	0.05	0.15	0.18	0.27	0.43
9	512	0.10	0.24	0.27	0.43	0.51
10	1024	0.15	0.36	0.47	0.67	0.85
11	2048	0.26	0.65	0.82	1.10	1.74
12	4096	0.53	1.29	1.52	2.19	3.42
13	8192	1.04	2.78	3.03	4.74	6.33
14	16384	2.33	6.73	6.47	9.59	14.91
15	32768	5.30	17.69	15.03	22.73	30.10
16	65536	13.41	52.56	35.56	53.58	70.36
17	131072	36.17	202.01	89.81	138.66	187.16
18	262144	106.16	832.31	250.45	387.24	507.62
19	524288	352.28	3388.19	793.75	1211.27	1609.34
20	1048576	1088.26	13605.23	3575.90	5787.97	7885.79

Table 1: Average times for solving Subset sum problem for different n values

Table 1 shows the statistics after 3 runs. Columns have the following meaning: n value; the number of compartments resulted from division at the end of computation (2^n of type Main); average time needed by Flame serial version, by kPWorkbench, by Flame parallel version using 2, 3 or 4 processors. All the times are given in seconds and were measured by `/usr/bin/time -v`, in order to have the same metric for all the tools.

Tools configuration. Because all the execution times would have been increased by FLAME saving to disk all the intermediary configurations, we have chosen to output the same amount of information with each tool (chosen rules and new configurations) and save only the last configuration file in XML format for the FLAME simulation.

Comparing our case study with previous experiments assessing FLAME and kPWorkbench performance [2, 13], this is the first time when FLAME is not saving large XML files after each iteration. This modification, realised in order to have equal conditions, is explaining why the FLAME simulator is obtaining better times in this case study, compared to kPWorkbench, although in previous experiments it has been different. Also, another important difference is the type of the model, previous experiments did not use division and this could result in different execution times.

As in the previous experiments only the serial version of FLAME was employed, we lack of other results to compare with, in order to address the question: why are the times obtained in the parallel version much higher than the serial

version? One explanation could be that model chosen does not have very complex processing functions, so the time needed for communication between processors is increasing the total time needed, without benefiting from the parallelism. Another explanation could be some tweaking settings of FLAME or Open MPI, which could be better configured, in order to get the best performance for the parallel implementation.

Also, in Table 1 we have provided the number of compartments for each n as a *rough* estimator of the space used: with a larger n the number of rules and also of object types for the compartment increases. So, a larger number of compartments comes also with more rules and more object types to be stored.

In order to have a better visualisation of the experimental results, the data from Table 1 has been represented in Fig. 3. The left plot shows the average time versus the number of compartments, the right plot displays for the horizontal axis the logarithmic scale of its left counterpart.

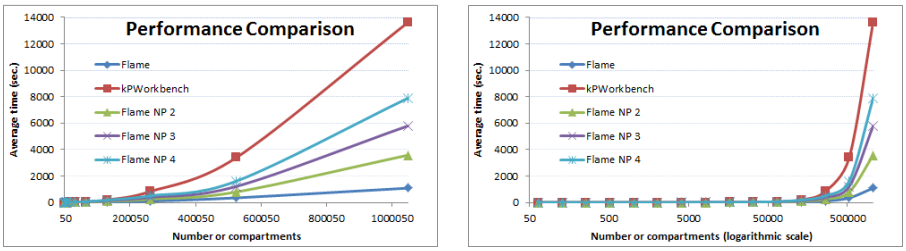


Fig. 3: Comparative simulation results for kPWorkbench, FLAME serial and parallel versions, with NP 2, 3 and 4

Unfortunately, at the time of writing this paper, there are neither public available tools for conversion from kP-Lingua specifications to FLAME GPU, nor from FLAME models to FLAME GPU. This is the reason why FLAME GPU models were not considered in the evaluation, although we have studied this possibility. However, as reported in a previous article [13], where a pulse generator model was translated manually to FLAME GPU, better execution times are expected for GPU version, but the construction of the model is much more tedious compared to the FLAME version.

7 Conclusions and Further Work

This paper presents recent efforts towards modelling of kernel P systems with structural rules in two agent based simulation environments, known as FLAME and FLAME GPU. In this context, we have extended the module of kPWorkbench for automated generation of FLAME models from kP–Lingua specifications to consider kP systems with division and dissolution rules. We also provided an overview of the differences between FLAME and its GPU version, and

outlined the main issues that should be taken into account for a model transformation.

Finally, we have conducted experiments to compare the performances of FLAME, serial and parallel versions, with respect to kPWorkbench, for a kernel P system with division rules.

As future work, we will tackle the automated translation from either kP-Lingua specifications or their FLAME counterparts models to FLAME GPU specifications, based on the main ideas presented here. Also we will address alternative MPI implementations and realise performance comparisons about the application of division and dissolution rules on more complex examples.

Acknowledgements

The work of RL, IN and LM was supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI (project number: PNII-ID-PCE-2011-3-0688).

References

[1] H. Bai, M. D. Rolfe, W. Jia, S. Coakley, R. K. Poole, J. Green, and M. Holcombe. Agent-based modeling of oxygen-responsive transcription factors in *Escherichia coli*. *Plos computational biology*, 10(4), 2014.

[2] M. E. Bakir, S. Konur, M. Gheorghe, I. Niculescu, and F. Ipate. High Performance Simulations of Kernel P Systems. In *2014 IEEE International Conference on High Performance Computing and Communications, 6th IEEE International Symposium on Cyberspace Safety and Security, 11th IEEE International Conference on Embedded Software and Systems, HPCC/CSS/ICESS 2014, Paris, France, August 20-22, 2014*. IEEE, 2014, pp. 409–412.

[3] G. Ciobanu and G. Wenyuan. P Systems Running on a Cluster of Computers. In C. Martín–Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*. Vol. 2933, in Lecture Notes in Computer Science, pp. 123–139. Springer Berlin Heidelberg, 2004.

[4] L. Diez Dolinski, R. Núñez Hervás, M. Cruz Echeandía, and A. Ortega. Distributed Simulation of P Systems by Means of Map-Reduce: First Steps with Hadoop and P-Lingua. In J. Cabestany, I. Rojas, and G. Joya, editors, *Advances in Computational Intelligence*. Vol. 6691, in Lecture Notes in Computer Science, pp. 457–464. Springer Berlin Heidelberg, 2011.

[5] M. García-Quismondo. Modelling and simulation of real-life phenomena in Membrane Computing. PhD thesis. University of Seville, Nov. 2013.

[6] M. Gheorghe, F. Ipate, C. Dragomir, L. Mierla, L. Valencia-Cabrera, M. García-Quismondo, and M. J. Pérez-Jiménez. Kernel P Systems - Version I. *Eleventh Brainstorming Week on Membrane Computing (11BWMC)*:97–124, Aug. 2013.

[7] M. Gheorghe, F. Ipate, L. Mierla, and S. Konur. kPWorkbench: A Software Framework for Kernel P Systems. In *Thirteenth Brainstorming Week on Membrane Computing, WBMC 2015, Sevilla, Spain, February 2-6, 2015*. L. F. Macías-Ramos, G. Păun, A. Riscos-Núñez, and L. Valencia–Cabrera, editors. Fenix Editora, 2015, pp. 179–194.

[8] M. Gheorghe, S. Konur, F. Ipate, L. Mierla, M. E. Bakir, and M. Stannett. An Integrated Model Checking Toolset for Kernel P Systems. In *Membrane Computing - 16th International Conference, CMC 2015, Valencia, Spain, August 17-21, 2015, Revised Selected Papers*. G. Rozenberg, A. Salomaa, J. M. Sempere, and C. Zandron, editors. Vol. 9504. In Lecture Notes in Computer Science. Springer, 2015, pp. 153–170.

[9] A. Gutiérrez, L. Fernández, F. Arroyo, and V. Martínez. Design of a Hardware Architecture Based on Microcontrollers for the Implementation of Membrane Systems. In *Symbolic and Numeric Algorithms for Scientific Computing, 2006. SYNASC '06. Eighth International Symposium on*, 2006, pp. 350–353.

[10] F. Ipate, T. Bălănescu, P. Kefalas, M. Holcombe, and G. Eleftherakis. A new model of communicating stream X-machine systems. *Romanian Journal of Information Science and Technology*, 6(1):165–184, 2003.

[11] F. Ipate, R. Lefticaru, L. Mierlă, L. V. Cabrera, H. Han, G. Zhang, C. Dragomir, M. J. P. Jiménez, and M. Gheorghe. *Kernel P Systems: Applications and Implementations*. In. *Proceedings of The Eighth International Conference on Bio–Inspired Computing: Theories and Applications (BIC-TA), 2013*. Z. Yin, L. Pan, and X. Fang, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1081–1089.

[12] D. B. Kirk and W. W. Hwu. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st ed., 2010.

[13] S. Konur, M. Kiran, M. Gheorghe, M. Burkitt, and F. Ipate. Agent-Based High-Performance Simulation of Biological Systems on the GPU. In *17th IEEE International Conference on High Performance Computing and Communications, HPCC 2015, 7th IEEE International Symposium on Cyberspace Safety and Security, CSS 2015, and 12th IEEE International Conference on Embedded Software and Systems, ICESS 2015, New York, NY, USA, August 24-26, 2015*. IEEE, 2015, pp. 84–89.

[14] L. F. Macías-Ramos. Developing efficient simulators for cell machines. PhD thesis. University of Seville, Feb. 2016.

[15] M. Á. Martínez-del-Amor. Accelerating Membrane Systems Simulators using High Performance Computing with GPU. PhD thesis. University of Seville, May 2013.

[16] M. A. Martínez-del-Amor, M. García-Quismondo, L. F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núòez, and M. J. Pérez-Jiménez. Simulating P Systems on GPU Devices: A Survey. *Fundam. Inf.*, 136(3):269–284, July 2015.

[17] M. A. Martínez-del-Amor, L. F. Macías-Ramos, L. Valencia-Cabrera, and A. R. a. Mario J. Pérez-Jiménez. Accelerated simulation of P systems on the GPU: A survey. In *Bio-inspired computing - theories and applications - 9th international conference, BIC-TA 2014, wuhan, china, october 16-19, 2014. proceedings*. L. Pan, G. Paun, M. J. Pérez-Jiménez, and T. Song, editors. Vol. 472. In Communications in Computer and Information Science. Springer, 2014, pp. 308–312.

[18] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg. *OpenCL programming guide*. Addison-Wesley, 1st ed., 2011.

[19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[20] I. Niculescu, M. Gheorghe, F. Ipate, and A. Stefanescu. From Kernel P Systems to X-Machines and FLAME. *Journal of Automata, Languages and Combinatorics*, 19(1-4):239–250, 2014.

[21] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU Computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[22] G. Păun. Computing with Membranes. *J. Comput. Syst. Sci.*, 61(1):108–143, 2000.

[23] B. Petreska and C. Teuscher. A Reconfigurable Hardware Membrane System. In C. Martín-Vide, G. Mauri, G. Păun, G. Rozenberg, and A. Salomaa, editors, *Membrane Computing*. Vol. 2933, in Lecture Notes in Computer Science, pp. 269–285. Springer Berlin Heidelberg, 2004.

[24] P. Richmond, D. C. Walker, S. Coakley, and D. M. Romano. High performance cellular level agent-based simulation with FLAME for the GPU. *Briefings in bioinformatics*, 11(3):334–347, 2010.

[25] H. Takizawa, K. Koyama, K. Sato, K. Komatsu, and H. Kobayashi. CheCL: Transparent Checkpointing and Process Migration of OpenCL Applications. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 864–876.

[26] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-Hadoop: MapReduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739–750, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[27] The P-Lingua Website.
URL: `http://www.p-lingua.org/`.

[28] kPWorkbench Home Page.
URL: `http://kpworkbench.org/`.

[29] FLAME website.
URL: `http://www.flame.ac.uk/`.

[30] FLAME GPU website.
URL: `http://www.flamegpu.com/`.

[31] The Message Passing Interface (MPI) standard.
URL: `http://www.mcs.anl.gov/research/projects/mpi/`.

[32] The NVIDIA Website.
URL: http://www.nvidia.com/content/global/global.php.
[33] OpenCL standard webpage.
URL: http://www.khronos.org/opencl.
[34] The Sevilla HPC Server.
URL: http://www.gcn.us.es/gpucomputing/.
[35] The Open MPI project.
URL: https://www.open-mpi.org/.

Algorithm 1 Transforming a kP Systems into Flame algorithm

```
1: procedure AddTransition(startState, stopState, strategy, guard)
   ▷ procedure adding the appropriate transition strategy to the current agent stack given as parameter and
     FLAME function applying rules conforming to execution strategy
   ▷ guard is an optional parameter that represents the transition guard
2:     if strategy is Sequence then
3:         agentTransitions.Push(startState, stopState, SequenceFunction, guard)
           ▷ FLAME function SequenceFunction applies rules in sequentially mode
4:     else if strategy is Choice then
5:         agentTransitions.Push(startState, stopState, ChoiceFunction, guard)
           ▷ FLAME function ChoiceFunction applies rules in choice mode
6:     else if strategy is ArbitraryParallel then
7:         agentTransitions.Push(startState, stopState, ArbitraryParallelFunction, guard)
           ▷ FLAME function ArbitraryParallelFunction applies rules in arbitrary parallel mode
8:     else if strategy is MaximalParallel then
9:         agentTransitions.Push(startState, stopState, MaximalParallelFunction, guard)
           ▷ FLAME function MaximalParallelFunction applies rules in maximal parallel mode
10:    end if
11: end procedure
12:
   ▷ main algorithm for traforming a kP system into Flame
13:
14: agentsStates.Clear()
15: agentsTransitions.Clear()
   ▷ empty state and transition stacks of agents
16: foreach membrane in kPSystem do
   ▷ for each membrane of kP system build corresponding agent, consisting of states and transitions
17:    agentStates.Clear()
18:    agentTransitions.Clear()
       ▷ empty state and transition stacks of agent that is built for the current membrane
19:    agentStates.Push(startState)
       ▷ adding the initial state of the X machine
20:    agentStates.Push(initializationState)
       ▷ adding initialization state
21:    agentTransitions.Push(startState, initializationState, IsNotPreviousApplyStructureRule)
       ▷ adding transition between the initial and initialization states; this transition performs objects allocation
         on rules and other initializations; if the agent is active, no rule of structure has been applied in the
         previous iteration
22:    agentTransitions.Push(startState, endState, IsPreviousApplyStructureRule)
       ▷ adding transition between the initial and end state; if the agent is inactive, a rule of structure has
         been applied in the previous iteration
23:    foreach strategy in membrane do
       ▷ for each strategy of the current membrane the corresponding states and transitions are built
24:        previousState = agentStates.Top()
           ▷ the last state is stored in a temporary variable
25:        if is first strategy and strategy.hasNext() then
           ▷ when the strategy is the first of several, state and transition corresponding to the execution strategy
             are added
26:            agentStates.Push(strategy.Name)
27:            AddTransition(previousState, strategy.Name, strategy)
28:        else
29:            if not strategy.hasNext() then
               ▷ if it is the last strategy, the transition corresponding to the execution strategy is added
30:                AddTransition(previousState, completedExecutionState, strategy)
31:            else
```

Algorithm 1 Transforming a kP Systems into Flame algorithm (continued)

```
32:                 agentStates.Push(strategy.Name)
                  ▷ add corresponding state of the current strategy
33:                 if strategy.Previous() is Sequence then
                  ▷ verify that previous strategy is of sequence type
34:                     AddTransition(previousState, strategy.Name, strategy, IsApplyAllRules)
                      ▷ add transition from preceding strategy state to the current strategy state. The guard is
                        active if all the rules have been applied in the previous strategy transition.
35:                     agentTransitions.Push(previousState, completedExecutionState, IsNotApplyAllRules)
                      ▷ add transition from preceding strategy state to state in which all strategies were finalized.
                        The guard is active if not all rules have been applied in the previous strategy transition
36:                 else
37:                     AddTransition(previousState, strategy.Name, strategy)
                      ▷ add transition from preceding strategy state to the current strategy state
38:                     agentTransitions.Push(previousState, completedExecutionState, IsApplyStructureRule)
                      ▷ add transition from preceding state strategy to state in which all strategies were finalized.
                        The guard is active when the structural rule has been applied on the previous strategy
                        transition
39:                 end if
40:             end if
41:         end if
42:     end for
43:     agentStates.Push(completedExecutionState)
      ▷ adding state in which all strategies were finalized
44:     agentStates.Push(PrepareTheNewMembranes)
      ▷ adding state in which id(s) is required for newly created membrane(s)
45:     agentTransitions.Push(completedExecutionState, PrepareTheNewMembranes, IsApplyStructureRule)
      ▷ add the transition to the prepare the new membranes state on which id(s) is required for newly created
        membrane(s), if the structure rule has been applied. The request is made through messages to the
        instance manager, agent that allocate new IDs for new agents of current type.
46:     agentStates.Push(CreateNewMembrane)
      ▷ adding state in which IDs are received through messages from instance manager agent and the new
        agents are created
47:     agentTransitions.Push(PrepareTheNewMembranes, CreateNewMembrane)
      ▷ on this transition the new agents are created with the new received ids
48:     agentStates.Push(applyChangesState)
      ▷ adding state in which changes produced by the applied rules are committed
49:     agentTransitions.Push(completedExecutionState, applyChangesState, IsNotApplyStructureRule)
      ▷ add transition to the apply changes state where changes produced by rules are applied, if has not been
        applied any structure rule
50:     agentTransitions.Push(applyChangesState, receiveState)
      ▷ adding transition on which changes produced by the applied rules are committed
51:     agentStates.Push(receiveState)
      ▷ add state that receives objects sent by applying the communication rules in other membranes
52:     agentTransitions.Push(receiveState, endState)
      ▷ add transition to the end state that receives objects sent by applying the communication rules in other
        membranes
53:     agentStates.Push(endState)
      ▷ add the final state
54:     agentsStates.PushAll(agentStates.Content())
      ▷ add the contents of the stack that holds the current agent states to the stack that holds the states of all
        agents
55:     agentsTransitions.PushAll(agentStates.Content())
      ▷ add the contents of the stack that holds the current agent transitions to the stack that holds the transitions
        of all agents
56: end for
```

Shallow Non-Confluent P Systems*

Alberto Leporati, Luca Manzoni, 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, luca.manzoni, mauri, porreca, zandron}@disco.unimib.it

Abstract. We prove that non-confluent (i.e., strongly nondeterministic) P systems with active membranes working in polynomial time are able to simulate polynomial-space nondeterministic Turing machines, and thus to solve all **PSPACE** problems. Unlike the confluent case, this result holds for shallow P systems. In particular, depth 1 (i.e., only one membrane nesting level and using elementary membrane division only) already suffices, and neither dissolution nor send-in communication rules are needed.

1 Introduction

Families of *confluent* recogniser P systems with active membranes [9] are known to characterise **PSPACE** in polynomial time [13,14]. In this kind of P systems the computations can be locally nondeterministic, but the final result (acceptance or rejection) must be consistent across all computations. This result seems to require that the membrane nesting depth of each P system in the family depends on the length of the input (the published results show that a *linear* depth suffices [13]); furthermore, all known algorithms employ non-elementary membrane division (i.e., division of membranes containing further membranes, resulting in the replication of whole subtrees of the membrane structure).

More recently, it has been proved [2] that when only elementary division (i.e., division for membranes not containing further membranes) is available, the power of P systems decreases to $\mathbf{P}^{\#\mathbf{P}}$, the class of problems solved in polynomial time by deterministic Turing machines with an oracle for a counting problem [8]; this class is conjectured to be strictly smaller than **PSPACE**. More specifically, P systems of depth 1 (consisting of an outermost membrane containing only elementary membranes) already characterise $\mathbf{P}^{\#\mathbf{P}}$ [3], and thus increasing the depth without also allowing non-elementary division does not increase the computing power. P systems of depth 0, where there exists only one membrane and division is unavailable, are known to characterise **P** [15,3].

* This work was partially supported by Fondo d'Ateneo (FA) 2015 of Università degli Studi di Milano-Bicocca: "Complessità computazionale e applicazioni crittografiche di modelli di calcolo bioispirati".

Fewer results are known for non-confluent P systems, where the computations need not agree on the result, and the overall behaviour is accepting if and only if there exists an accepting computation (i.e., a strongly nondeterministic behaviour analogous to Turing machines). Clearly, all lower bounds of confluent P systems hold for non-confluent ones. The only other published result concerning non-confluent recogniser P systems with active membranes, to the authors' best knowledge, is a characterisation of **NP** by means of polynomial-time non-confluent P systems with active membranes without any kind of membrane division [11].

Membrane division in confluent P systems is commonly used to simulate the effect of nondeterminism, by exploring in parallel all possible nondeterministic choices and combining the results by disjunction [15], threshold or majority [3], or alternation of conjunctions and disjunctions [13], depending on which rules are available and the depth of the membrane structure. It is then natural to ask whether these results can be somehow improved by employing actual nondeterminism, i.e., by exploiting non-confluence in addition to membrane division.

In this paper we prove that this is indeed the case, since the lower bound **PSPACE** can actually be reached by "shallow" (small-depth) polynomial-time non-confluent P systems: specifically, depth 1, and thus division only for elementary membranes, already suffice for reaching **PSPACE**. Furthermore, the P systems employed can be *monodirectional* [1,4], i.e., without using send-in communication rules. Monodirectionality is known to decrease the power of confluent P systems; for instance, polynomial-time monodirectional confluent P systems of depth 1 characterise $\mathbf{P}_{\parallel}^{\mathbf{NP}}$ (the class of problems solved in polynomial time with parallel queries to an **NP** oracle, conjectured to be smaller than $\mathbf{P}^{\#\mathbf{P}}$), and $\mathbf{P}^{\mathbf{NP}}$ (the class where the oracle queries are not restricted to be parallel, which is probably smaller than **PSPACE**) if the depth is unbounded [4].

2 Basic Notions

In this paper we use P systems with active membranes [9] using only object evolution rules $[a \to w]_h^\alpha$, send-out communication rules $[a]_h^\alpha \to [\ ]_h^\beta\ b$ and elementary membrane division rules $[a]_h^\alpha \to [b]_h^\beta\ [c]_h^\gamma$.

The *depth* of a P system is defined as the depth of its membrane structure when considered as a rooted tree, i.e., as the length of the longest path from the outermost membrane to an elementary membrane.

In particular, we are dealing with *recogniser P systems* Π [10], whose alphabet includes the distinguished result objects yes and no; exactly one result object must be sent out from the outermost membrane to signal acceptance or rejection, and only at the last computation step. If all possible computations of Π agree on the result, the P system is said to be *confluent*. In this paper, however, we only deal with the more general *non-confluent* recogniser P systems, where different computations need not agree on the result, and the final result is acceptance if and only if *at least one* computation is accepting.

A decision problem, or language $L \subseteq \Sigma^\star$, is solved by a *family* of P systems $\boldsymbol{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$, where Π_x accepts if and only if $x \in L$. In that case, we

say that $L(\boldsymbol{\Pi}) = L$. As usual, we require a uniformity condition [6] on families of P systems:

Definition 1. *A family of P systems $\boldsymbol{\Pi} = \{\Pi_x : x \in \Sigma^\star\}$ is* (polynomial-time) uniform *if the mapping $x \mapsto \Pi_x$ can be computed by two polynomial-time deterministic Turing machines E and F as follows:*

- *$F(1^n) = \Pi_n$, where n is the length of the input x and Π_n is a common P system for all inputs of length n, with a distinguished input membrane.*
- *$E(x) = w_x$, where w_x is a multiset encoding the specific input x.*
- *Finally, Π_x is simply Π_n with w_x added to its input membrane.*

The family $\boldsymbol{\Pi}$ is said to be (polynomial-time) semi-uniform if there exists a single deterministic polynomial-time Turing machine H such that $H(x) = \Pi_x$ for each $x \in \Sigma^\star$.

The class of decision problems solved by uniform families of non-confluent P systems with active membranes working in polynomial time is denoted by the symbol $\mathbf{NPMC}_{\mathcal{AM}}$. The corresponding class for families of P systems with depth-1 membrane structures using only object evolution, send-out communication and elementary membrane division rules is denoted by $\mathbf{NPMC}_{\mathcal{AM}(\text{depth-1},-\text{i},-\text{d},-\text{ne})}$, where −i, −d, and −ne denote the lack of send-in, dissolution, and non-elementary division rules, respectively. For the complexity classes defined in terms of Turing machines, such as **NP**, $\mathbf{P}_{\|}^{\mathbf{NP}}$, $\mathbf{P}^{\mathbf{NP}}$, $\mathbf{P}^{\#\mathbf{P}}$, and **PSPACE** we refer the reader to Papadimitriou's book [8].

3 Simulating Nondeterministic Turing Machines

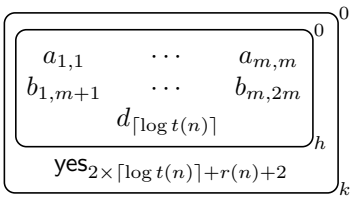
Let N be a nondeterministic Turing machine working in polynomial space $p(n)$. Let Σ be the tape alphabet of N, and let Q be its set of states. Without loss of generality, we assume that N has a unique accepting configuration, consisting of a unique accepting state, an entirely blank tape, and the tape head located on the leftmost position. We can assume that all computations of N halt within exponential time $t(n) = |\Sigma|^{p(n)} \times |Q| \times p(n)$.

Suppose that string x is an input for N, and let $m = p(|x|) + 3$. A configuration $\mathcal{C}$ of N can be encoded as a delimited string $\$a_1 \cdots a_{k-1} q a_k \cdots a_{p(n)}\$$ of length m over the alphabet $\Sigma \cup Q \cup \{\$\}$. This denotes that the tape of N contains the string $a_1 \cdots a_{k-1} a_k \cdots a_{p(n)}$, that the machine is in state q, and that the tape head is located on the k-th tape cell.

Deciding whether N accepts an input x is equivalent to deciding whether the unique final accepting configuration $\mathcal{C}'$ is reachable from the initial configuration $\mathcal{C}$ on input x within $t(|x|)$ steps. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ be configurations of N, let $\mathcal{C}_1 \to \mathcal{C}_2$ denote that $\mathcal{C}_2$ is reachable from $\mathcal{C}_1$ via a single computation step, and let $\mathcal{C}_1 \to^t \mathcal{C}_2$ denote reachability within t steps. Then, we have

$$\begin{aligned}
&\mathcal{C}_1 \to^1 \mathcal{C}_2 && \text{iff} && \mathcal{C}_1 = \mathcal{C}_2 \text{ or } \mathcal{C}_1 \to \mathcal{C}_2 \\
&\mathcal{C}_1 \to^t \mathcal{C}_2 \text{ with } t > 1 && \text{iff} && \text{there exists } \mathcal{C} \text{ such that } \mathcal{C}_1 \to^{\lceil t/2 \rceil} \mathcal{C} \to^{\lfloor t/2 \rfloor} \mathcal{C}_2
\end{aligned}$$

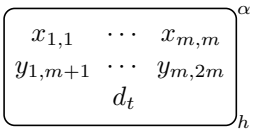
The Turing machine N on input x of length n is simulated by a P system Π_x, whose initial configuration is



(1)

Here the string $a_1 \cdots a_m$ encodes the initial configuration of N on input x, that is, $a_1 \cdots a_m = \$q_0 x \sqcup^{p(n)-n} \$$ with $\sqcup$ denoting a blank cell and q_0 the initial state of N; these symbols need a further subscript in Π_x in order to keep track of their position within the string. Analogously, the string $b_1 \cdots b_m$ encodes the unique accepting configuration of N, that is, $b_1 \cdots b_m = \$q_{\mathsf{yes}} \sqcup^{p(n)} \$$, where q_{yes} is the accepting state. In this case, the symbols are subscripted with $m+1, \ldots, 2m$ as if the P system stored a single string $a_1 \cdots a_m b_1 \cdots b_m$; this will prove useful in a later phase of the simulation. The other objects do not encode information about N, but play an auxiliary role in the simulation; in particular, the function $r(n)$, appearing in the subscript of the object yes, will be defined later.

For the whole first phase of the computation of Π_x (including the initial configuration) the membranes with label h maintain, as an invariant, a configuration of the form



(2)

where $1 \leq t \leq \lceil \log t(n) \rceil$, the charge α is either 0 or $+$, and $x_{1,1} \cdots x_{m,m}$ and $y_{1,m+1} \cdots y_{m,2m}$ are multisets respectively encoding the strings $x_1 \cdots x_m$ and $y_1 \cdots y_m$, which in turn encode two configurations $\mathcal{C}_1$ and $\mathcal{C}_2$ of N as described above. This invariant is restored every two steps of the first phase of the computation of Π_x.

The purpose of this membrane, for $t > 1$, is to guess a computation path of length at most 2^t from $\mathcal{C}_1$ to $\mathcal{C}_2$; computation paths from $\mathcal{C}_1 \to \cdots \to \mathcal{C}_2$ shorter than 2^t are padded to length 2^t by repeating some intermediate configurations (recall that $\mathcal{C} \to^1 \mathcal{C}$ for all $\mathcal{C}$). If $t = 0$, the membrane checks whether the configuration $\mathcal{C}_2$ is reachable by N in one step from $\mathcal{C}_1$; if it is the case, then it outputs an object yes, and otherwise an object no after exactly $2t + r(n) + 1$ computation steps.

Let us describe recursively the behaviour of the membrane. If $t > 0$, then the problem of guessing a computation $\mathcal{C}_1 \to^{2^t} \mathcal{C}_2$ is divided into the two subproblems of guessing a computation $\mathcal{C}_1 \to^{2^{t-1}} \mathcal{C}$ and a computation $\mathcal{C} \to^{2^{t-1}} \mathcal{C}_2$, where $\mathcal{C}$ is a nondeterministically guessed mid-point. This mid-point is guessed by rewriting each object σ_i of the multiset $x_{1,1} \cdots x_{m,m}$ into a primed version σ_i' of itself,

together with an object τ_i'', where τ is a nondeterministically chosen symbol of the alphabet of the configurations of N:

$$[\sigma_i \to \sigma_i' \, \tau_i'']_h^\alpha \qquad \text{for } \alpha \in \{0,+\},\ \sigma,\tau \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m \tag{3}$$

Notice that the P system guesses an arbitrary string as a configuration $\mathcal{C}$; the string might even be an invalid encoding (e.g., with multiple symbols denoting the state of N); the validity of the configuration will be checked later. Simultaneously, the objects of the target configuration $y_{1,m+1} \cdots y_{m,2m}$ are primed:

$$[\sigma_i \to \sigma_i']_h^\alpha \qquad \text{for } \alpha \in \{0,+\},\ \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } m+1 \le i \le 2m \tag{4}$$

While the objects encoding the configurations of N are thus rewritten, the membrane is divided by d_t into two membranes differing only in their charge:

$$[d_t]_h^\alpha \to [d_t']_h^+ \, [d_t']_h^0 \qquad \text{for } \alpha \in \{0,+\} \text{ and } 1 \le t \le \lceil \log t(n) \rceil$$

Hence, the original membrane h leads to the following configuration:

$$\underset{h}{\boxed{\begin{array}{ccc} x'_{1,1} & \cdots & x'_{m,m} \\ z''_{1,1} & \cdots & z''_{m,m} \\ y'_{1,m+1} & \cdots & y'_{m,2m} \\ & d'_t & \end{array}}^{+}} \qquad \underset{h}{\boxed{\begin{array}{ccc} x'_{1,1} & \cdots & x'_{m,m} \\ z''_{1,1} & \cdots & z''_{m,m} \\ y'_{1,m+1} & \cdots & y'_{m,2m} \\ & d'_t & \end{array}}^{0}}$$

where the objects $z''_{i,i}$ represent the mid-point configuration $\mathcal{C}$ guessed by the membrane. Now configuration $\mathcal{C}$ becomes the target configuration in the left membrane, having charge $+$. This requires eliminating all primes, deleting the objects $y'_{1,m+1} \cdots y'_{m,2m}$ and adjusting the subscripts of $z''_{1,1} \cdots z''_{m,m}$; this is performed by the following rules:

$$\begin{array}{ll} [\sigma_i' \to \sigma_i]_h^+ & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m \\ [\sigma_i' \to \epsilon]_h^+ & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } m+1 \le i \le 2m \\ [\sigma_i'' \to \sigma_{i+m}]_h^+ & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m \end{array}$$

On the other hand, the configuration $\mathcal{C}$ becomes the source configuration in the right membrane (with charge 0), where the objects $x'_{1,1} \cdots x'_{m,m}$ must be deleted:

$$\begin{array}{ll} [\sigma_i' \to \epsilon]_h^0 & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m \\ [\sigma_i' \to \sigma_i]_h^0 & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } m+1 \le i \le 2m \\ [\sigma_i'' \to \sigma_i]_h^0 & \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m \end{array}$$

Finally, the object d_t' is rewritten into d_{t-1} inside both membranes:

$$[d_t' \to d_{t-1}]_h^\alpha \qquad \text{for } \alpha \in \{0,+\} \text{ and } 1 \le t \le \lceil \log t(n) \rceil$$

Hence, the P system reaches the configuration

$$\begin{array}{|ccc|}_{h}^{+}\\ x_{1,1} & \cdots & x_{m,m}\\ z_{1,m+1} & \cdots & z_{m,2m}\\ & d_{t-1} & \end{array} \qquad \begin{array}{|ccc|}_{h}^{0}\\ z_{1,1} & \cdots & z_{m,m}\\ y_{1,m+1} & \cdots & y_{m,2m}\\ & d_{t-1} & \end{array}$$

with two membranes having a configuration of the form (2). This restores the associated invariant.

After $2 \times \lceil \log t(n) \rceil$ steps (twice the initial subscript of the object d_t), all membranes with label h simultaneously reach a configuration of the form

$$\begin{array}{|ccc|}_{h}^{\alpha}\\ x_{1,1} & \cdots & x_{m,m}\\ y_{1,m+1} & \cdots & y_{m,2m}\\ & d_0 & \end{array}$$

for some $\alpha \in \{0, +\}$ and $x_1, \ldots, x_m, y_1, \ldots, y_m \in \Sigma \cup Q \cup \{\$\}$. The last phase of the simulation is triggered by objects d_0 being sent out and changing the charge of the membranes to $-$:

$$[d_0]_h^\alpha \to [\,]_h^- \ \# \qquad \text{for } \alpha \in \{0, +\} \tag{5}$$

While rule (5) is applied, the charge of the membrane is still 0 or $+$, and the rules of type (3) and (4) are still enabled; thus, each membrane h reaches a configuration of the form

$$\begin{array}{|ccc|}_{h}^{-}\\ x'_{1,1} & \cdots & x'_{m,m}\\ z''_{1,1} & \cdots & z''_{m,m}\\ y'_{1,m+1} & \cdots & y'_{m,2m}\\ \end{array}$$

When the charge of a membrane with label h is $-$, the objects of the form τ_i'' (which have been just produced, but are actually not needed in this phase) are deleted:

$$[\tau_i'' \to \epsilon]_h^- \qquad \text{for } \tau \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le m$$

The remaining objects σ_i' are rewritten into a "tilded" version. This allows us to re-use the charges 0 and $+$ in the next phase of the computation (which will make use of a simulation from [3]) without creating conflicts with previous rules.

$$[\sigma_i' \to \tilde{\sigma}_i]_h^- \qquad \text{for } \sigma \in \Sigma \cup Q \cup \{\$\} \text{ and } 1 \le i \le 2m$$

This leads to the configuration

$$\begin{array}{|ccc|}_{h}^{-}\\ \tilde{x}_{1,1} & \cdots & \tilde{x}_{m,m}\\ \tilde{y}_{1,m+1} & \cdots & \tilde{y}_{m,2m}\\ \end{array} \tag{6}$$

Each membrane h now contains what can be considered as a string of length $2m$, consisting of the concatenation of two (possibly malformed) encodings of configurations of N.

From [3] we know that a single membrane is able to efficiently simulate a polynomial-time *deterministic* Turing machine (as long as there is no communication with adjacent membranes) if the tape is encoded as in configuration (6). The idea is to simulate a larger number of charges (referred to as *extended* charges) by encoding them in the subscripts of each object; the subscripts are kept synchronised by multiple sequential updates of the actual charges $\{+, 0, -\}$. The extended charges are employed in order to store pairs (q, i) of state and tape head position of the simulated Turing machine. A transition such as $\delta(q, a) = (q', a', +1)$ is then implemented as a rule of the form $[a_i]_h^{(q,i)} \to [a'_i]_h^{(q',i+1)}$, which is actually carried out in multiple steps using standard charges, object evolution and send-out communication rules.

In our particular case, each membrane h can simulate a Turing machine that checks whether the content of such membrane consists of two valid consecutive configurations of N, or two identical valid configurations of N. We can assume, without loss of generality, that such Turing machine halts exactly in polynomial time $r(n)$ for all strings of length n; the result is given by outputting yes or no from the membrane.

After $2 \times \lceil \log t(n) \rceil + r(n)$ steps, a total of $2^{\lceil \log t(n) \rceil}$ instances of yes and no reach the outermost membrane k. If there is at least one instance of no, then there existed an instance of membrane h containing either an invalid configuration, or two non-consecutive configurations; this means that the P system Π_x guessed a malformed computation of N. In that case, it sends out any of the objects no as the final result of the computation:

$$[\mathsf{no}]_k^0 \to [\ ]_k^- \ \mathsf{no}$$

If there is no instance of no inside k, then all membranes h contained valid consecutive configurations (or pairs of identical valid configurations). Since the initial membrane h contained the initial and final configurations of N on input x, this means that Π_x guessed a legitimate accepting computation of N. In that case, all objects yes sent out from the elementary membranes h are ignored, and instead the timed object yes_t, which already appears in the initial configuration (1), produces the output of the P system. This object counts down for the entire duration of the simulation:

$$[\mathsf{yes}_t \to \mathsf{yes}_{t-1}]_k^0 \qquad \text{for } 1 \leq t \leq 2 \times \lceil \log t(n) \rceil + r(n) + 2$$

If at time $2 \times \lceil \log t(n) \rceil + r(n) + 2$ membrane k still has charge 0, then the P system has not rejected, and it can send out yes_0 as yes, as the result:

$$[\mathsf{yes}_0]_k^0 \to [\ ]_k^- \ \mathsf{yes}$$

In both cases Π_x halts by sending out a result at the last computation step.

The P system Π_x has thus an accepting computation if and only if there exists a computation path from the initial configuration of N on input x to its accepting computation, that is, if and only if N accepts.

Notice that the mapping $x \mapsto \Pi_x$ is uniform, since all rules of Π_x only depend on the *length* of the input, and not on the input itself. Furthermore, the initial membrane structure is the same for all Π_x. The only portion of the initial configuration that depends on the actual input x is the content of membrane h, which is chosen as the input of the P system. The mapping $x \mapsto \Pi_x$ can also be computed in polynomial time: the encoding of the input simply consists in adding subscripts to the input symbols of x, and the rules are easy to compute, since they all range over sets independent of the input, or over sets of natural numbers depending on the input length, and never require sophisticated computation.

Theorem 1. *Let N be a nondeterministic Turing machine working in polynomial space. Then, there exists a uniform family $\boldsymbol{\Pi}$ of non-confluent P systems of depth* 1*, using only object evolution, send-out and elementary division rules, and working in polynomial time such that $L(N) = L(\boldsymbol{\Pi})$.* □

Since arbitrary polynomial-space Turing machines can be simulated, the whole class they characterise is solved by shallow non-confluent P systems with a limited range of rules:

Corollary 1. $\mathbf{PSPACE} \subseteq \mathbf{NPMC}_{\mathcal{AM}(\text{depth-1},-\text{i},-\text{d},-\text{ne})}$. □

4 Conclusions

The results obtained in this paper show that, even with depth-1 membrane structures and monodirectional communication, non-confluent P systems with active membranes are already able to solve **PSPACE**-complete problems in polynomial time, and are thus conjecturally stronger than confluent ones with the same restrictions (and even those with only one of such restrictions).

This result is a first step towards a characterisation of the power of polynomial-time non-confluent P systems with active membranes. If **PSPACE** turned out to also be an upper bound, although it has been conjectured that it might not be so [14], this would show that non-confluence subsumes both nesting depth beyond 1 and bidirectionality. In that case, it would be interesting to find other parameters (such as unusual combinations of admissible rules) which can be "tuned" in order to obtain complexity classes between **NP** and **PSPACE**.

We also conjecture that an algorithm similar to the one provided here for P systems with active membranes can also be implemented for tissue-like P systems using either cell division [12] or cell separation rules [7], since they seem to share several features and limitations of cell-like P systems of depth 1 [5].

References

1. Alhazov, A., Freund, R.: On the efficiency of P systems with active membranes and two polarizations. In: Mauri, G., Păun, Gh., Pérez-Jiménez, M.J., Rozenberg,

G., Salomaa, A. (eds.) Membrane Computing, 5th International Workshop, WMC 2004. Lecture Notes in Computer Science, vol. 3365, pp. 146–160. Springer (2005)
2. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Simulating elementary active membranes, with an application to the P conjecture. In: Gheorghe, M., Rozenberg, G., Sosík, P., Zandron, C. (eds.) Membrane Computing, 15th International Conference, CMC 2014, Lecture Notes in Computer Science, vol. 8961, pp. 284–299. Springer (2014)
3. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Membrane division, oracles, and the counting hierarchy. Fundamenta Informaticae 138(1–2), 97–111 (2015)
4. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Monodirectional P systems. In: Macìas-Ramos, L.F., Păun, Gh., Riscos-Núñez, A., Valencia-Cabrera, L. (eds.) Proceedings of the 13th Brainstorming Week on Membrane Computing. pp. 207–226. Fénix Editora (2015)
5. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Tissue P systems can be simulated efficiently with counting oracles. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing, 16th International Conference, CMC 2015. Lecture Notes in Computer Science, vol. 9504, pp. 251–261 (2015)
6. Murphy, N., Woods, D.: The computational power of membrane systems under tight uniformity conditions. Natural Computing 10(1), 613–632 (2011)
7. Pan, L., Pérez-Jiménez, M.J.: Computational complexity of tissue-like P systems. Journal of Complexity 26(3), 296–315 (2010)
8. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
9. Păun, Gh.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics 6(1), 75–90 (2001)
10. Pérez-Jiménez, M.J., Romero-Jiménez, A., Sancho-Caparrini, F.: Complexity classes in models of cellular computing with membranes. Natural Computing 2(3), 265–284 (2003)
11. Porreca, A.E., Mauri, G., Zandron, C.: Non-confluence in divisionless P systems with active membranes. Theoretical Computer Science 411(6), 878–887 (2010)
12. Păun, Gh., Pérez-Jiménez, M.J., Riscos Núñez, A.: Tissue P systems with cell division. International Journal of Computers, Communications & Control 3(3), 295–303 (2008)
13. Sosík, P.: The computational power of cell division in P systems: Beating down parallel computers? Natural Computing 2(3), 287–298 (2003)
14. 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)
15. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-complete problems using P systems with active membranes. In: Antoniou, I., Calude, C.S., Dinneen, M.J. (eds.) Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, pp. 289–301. Springer (2001)

Data Structures with cP Systems or Byzantine Succintly

Radu Nicolescu

Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand
r.nicolescu@auckland.ac.nz

Abstract. We refine our earlier version of P systems with complex symbols. The new version, called cP systems, enables the creation and manipulation of high-level data structures which are typical in high-level languages, such as: relations (graphs), associative arrays, lists, trees. We assess these capabilities by attempting a revised version of our previously best solution for the Byzantine agreement problem – a famous problem in distributed algorithms, with non-trivial data structures and algorithms. In contrast to our previous solutions, which use a greater than exponential number of symbols and rules, the new solution uses a *fixed sized* alphabet and ruleset, independent of the problem size. The new ruleset follows closely the conceptual description of the algorithm. This revised framework opens the way to further extensions, which may bring P systems closer to the conceptual Actor model.

Keywords: Distributed algorithms, Byzantine agreement, EIG trees, membrane computing, P systems, cP systems, inter-cell parallelism, intra-cell parallelism, Prolog terms and unification, complex symbols, cells with subcells, generic rules, synchronous and asynchronous models, Actor model.

1 Introduction

We refine our earlier version of P systems with complex symbols. The new version, called cP systems, enables the creation and manipulation of high-level data structures which are typical in high-level languages, such as: relations (graphs), associative arrays, lists, trees.

We assess these capabilities by attempting a revised version of our previously best solution for the Byzantine agreement problem, a famous problem in distributed algorithms, with non-trivial data structures and algorithms. In contrast to our previous solution, which uses a greater than exponential number of symbols and rules, the new solution uses a *fixed sized* alphabet and ruleset, independent of the problem size. The new ruleset follows closely the conceptual description of the algorithm.

Section 2 introduces high-level data structures in cP systems. Sections 3, 4, and 5 discuss the Byzantine algorithm and its classical implementation based on

EIG trees; this material is reproduced or adapted from our earlier papers [7, 8]. The remaining sections discuss our newly revised solution.

2 Data structures in cP systems

We assume that the reader is familiar with the membrane extensions collectively known as *complex symbols*, proposed by Nicolescu et al. [20, 19, 21, 17]. However, to ensure some degree of self-containment, our revised extensions, called cP systems, are reviewed in Appendix A. The reader is encouraged to check the main changes from our earlier versions, including: a simplified definition for complex symbols (subcells) and a standard set of complexity measures.

In this section we sketch the design of high-level data structures, similar to the data structures used in high-level pseudocode or high-level languages: numbers, relations, functions, associative arrays, lists, trees, together with alternative more readable notations. These data structures are critical in our model of the Byzantine algorithm.

Natural numbers. Natural numbers can be represented via *multisets* containing repeated occurrences of the *same* atom. For example, considering that *1* represents an ad-hoc unary digit, the following complex symbols can be used to describe the contents of a virtual integer *variable* a: $a() = a(\lambda)$ — the value of a is 0; $a(1^3)$ — the value of a is 3. For concise expressions, we may alias these number representations by their corresponding numbers, e.g. $a() \equiv a(0), b(1^3) \equiv b(3)$. Nicolescu et al. [20, 19, 21] show how the basic arithmetic operations can be efficiently modelled by P systems with complex symbols.

Relations and functions. Consider the *binary relation* r, defined by: $r = \{(a, b),\ (b, c),\ (a, d),\ (d, c)\}$ (which has a diamond-shaped graph). Using complex symbols, relation r can be represented as a *multiset* with four r items, $\{r(\kappa(a)\ \upsilon(b)),\ r(\kappa(b)\ \upsilon(c)),\ r(\kappa(a)\ \upsilon(d)),\ r(\kappa(d)\ \upsilon(c))\}$, where ad-hoc atoms κ and υ introduce *domain* and *codomain* values (respectively). Hiding the less relevant representation choices, we may alias the items of this multiset by a more expressive notation such as: $\{(a \overset{r}{\rightleftarrows} b),\ (b \overset{r}{\rightleftarrows} c),\ (a \overset{r}{\rightleftarrows} d),\ (d \overset{r}{\rightleftarrows} c)\}$.

If the relation is a *functional relation*, then we can emphasise this by using another operator, such as "mapsto". For example, the functional relation $f = \{(a, b),\ (b, c),\ (d, c)\}$ can be represented by multiset $\{f(\kappa(a)\ \upsilon(b)),\ f(\kappa(b)\ \upsilon(c)),\ f(\kappa(d)\ \upsilon(c))\}$ or by the more suggestive notation: $\{(a \overset{f}{\mapsto} b),\ (b \overset{f}{\mapsto} c),\ (d \overset{f}{\mapsto} c)\}$. To highlight the actual mapping value, instead of $a \overset{f}{\mapsto} b$, we may also use the succinct abbreviation $f[a] = b$.

In this context, the $\rightleftarrows$ and $\mapsto$ operators are considered to have a high associative priority, so the enclosing parentheses are mostly required for increasing the readability (e.g. in text).

Associative arrays. Consider the *associative array* x, with the following key-value mappings (i.e. functional relation): $\{1 \mapsto a; 1^3 \mapsto c; 1^7 \mapsto g\}$. Using complex symbols, array x can be represented as a multiset with three items,

$\{x(\kappa(1)\,\upsilon(a)),\ x(\kappa(1^3)\,\upsilon(c)),\ x(\kappa(1^7)\,\upsilon(g))\}$, where ad-hoc atoms κ and υ introduce keys and values (respectively). Hiding the less relevant representation choices, we may alias the items of this multiset by the more expressive notation $\{1 \overset{x}{\mapsto} a,\ 1^3 \overset{x}{\mapsto} c,\ 1^7 \overset{x}{\mapsto} g\}$.

Lists. Consider the *list* y, containing the following sequence of values: $[u; v; w]$. Using complex symbols, list y can be represented as $y(\,\gamma(u\ \gamma(v\ \gamma(w\ \gamma()))))$, where the ad-hoc atom γ represents the list constructor *cons* and $\gamma()$ the empty list. Hiding the less relevant representation choices, we may alias this list by the more expressive equivalent notation $y(u \,|\, v \,|\, w)$ – or by $y(u \,|\, y')$, $y'(v \,|\, w)$ – where operator | separates the head and the tail of the list. The notation $z(|)$ is shorthand for $z(\gamma())$ and indicates an empty list, z.

Trees. Consider the *binary tree* z, described by the structured expression $(a, (b), (c, (d), (e)))$, i.e. z points to a root node which has: (i) the value a; (ii) a left node with value b; and (iii) a right node with value c, left leaf d, and right leaf e. Using complex symbols, tree z can be represented as $z(a\ \phi(b)\ \psi(c\ \phi(d)\ \psi(e)))$, where ad-hoc atoms ϕ, ψ introduce left subtrees, right subtrees (respectively).

3 Byzantine agreement

The Byzantine agreement problem was first proposed by Pease *et al.* in 1980 [22] and further elaborated in Lamport *et al.*'s seminal paper [12]. This problem addresses a fundamental issue in complex systems: correctly functioning processes must be able to overcome their possible differences and achieve a consensus, despite arbitrarily faulty processes that can give conflicting information to different parts of the system.

The Byzantine agreement has become one of the most studied problems in distributed algorithms—some even consider it the "crown jewel" of distributed algorithms. Lynch covers several versions of this problem and their solutions, including a complete description of the classical algorithm, based on Exponential Information Gathering (EIG) trees as a data structure [13].

Recent years have seen revived interest in this problem and its solutions, to achieve higher performance or stronger resilience, in a wide variety of contexts [4, 1, 3, 14], including, for example, solutions for quantum computers [2].

To the best of our knowledge, except our previous work on Byzantine agreement problem [9–11, 7, 8], no other complete solutions for P systems has been published. In the context of P systems, this problem was briefly mentioned, without solutions [6, 5]. Our solution was based on the classical EIG-based algorithm, where each EIG node was implemented by a distinct cell.

In this paper, we provide a newly revised P solution for the Byzantine agreement problem, based on EIG trees, for N processes connected in a complete graph. Each process is modelled by the combination of $N + 1$ cells: one "main" cell, which evaluates the EIG tree, plus one "firewall" cell for each duplex link (including one for itself). The new main cell uses a *fixed* number of states and high-level rules: 9 states and 16 rules for the main cell, and 5 states and 7 rules

for the firewall cell; these high-level rules closely map the EIG algorithm description. In contrast, our previous best solution [8] used $\mathcal{O}(L)$ states and $\mathcal{O}(N!)$ symbols and rules (i.e. factorial complexity), where L is the number of messaging rounds (where $L = \lceil (N-1)/3 \rceil$).

4 EIG trees

We assume that the reader is familiar with the basic terminology and notations: functions, relations, graphs, nodes (vertices), arcs, directed graphs, dags, trees, alphabets, strings and multisets [18]. Given two sets, A, B, a subset f of their cartesian product, $f \subseteq A \times B$, is a *functional relation* if $\forall (x, y_1), (x, y_2) \in f \Rightarrow y_1 = y_2$. Obviously, any function $f : A \to B$ can be viewed a functional relation, $\{(x, f(x)) \mid x \in A\}$, and, vice-versa, any functional relation can be viewed as a function.

We now recall a few basic concepts from combinatorial enumerations. The *integer range* from m to n is denoted by $[m, n]$, i.e. $[m, n] = \{m, m+1, \ldots, n\}$, if $m \leq n$, and $[m, n] = \emptyset$, if $m > n$. The set of *permutations* of n of length m is denoted by $P(n, m)$, i.e. $P(n, m) = \{\pi : [1, m] \to [1, n] \mid \pi \text{ is injective}\}$. A permutation π is represented by the sequence of its values, i.e. $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$, and we will often abbreviate this further as the sequence $\pi = \pi_1.\pi_2 \ldots \pi_m$. The sole element of $P(n, 0)$ is denoted by $()$, or by λ, if the context removes any possible ambiguity. Given a subrange $[p, q]$ of $[1, m]$, we define a *subpermutation* $\pi(p : q) \in P(n, q-p+1)$ by $\pi(p : q) = (\pi_p, \pi_{p+1}, \ldots, \pi_q)$. The *image* of a permutation π, denoted by $\mathrm{Im}(\pi)$, is the set of its values, i.e. $\mathrm{Im}(\pi) = \{\pi_1, \pi_2, \ldots, \pi_m\}$. The *concatenation* of two permutations is denoted by $\odot$, i.e. given $\pi \in P(n, m)$ and $\tau \in P(n, k)$, such that $\mathrm{Im}(\pi) \cap \mathrm{Im}(\tau) = \emptyset$, $\pi \odot \tau = (\pi_1, \pi_2, \ldots, \pi_m, \tau_1, \tau_2, \ldots, \tau_k) \in P(n, m+k)$.

An *Exponential Information Gathering* (EIG) tree $T_{N,L}$, $N \geq L \geq 0$, is a labelled rooted tree of height L that is defined recursively as follows. The tree $T_{N,0}$ is a rooted tree with just one node, its root, labelled λ. For $L \geq 1$, $T_{N,L}$ is a rooted tree with $1 + N|T_{N-1,L-1}|$ nodes (where $|T|$ is the size of tree T), root λ, having N subtrees, where each subtree is isomorphic with $T_{N-1,L-1}$ and each node, except the root, is labelled by an element of $[1, N]$ that is *different* from any ancestor node (and also different from any left sibling node, if we want to display it like an ordered tree). Note that, $T_{N,L-1}$ is isomorphic and identically labelled with the tree obtained from $T_{N,L}$ by deleting all its leaves.

It is straightforward to see that there is a bijective correspondence between the permutations of $P(N, L)$ and the sequences (concatenations) of labels on all paths from root to the leaves of $T_{N,L}$. Thus, each node σ in an EIG tree $T_{N,L}$ is uniquely identified by a permutation $\pi_\sigma \in P(N, l)$, where $l \in [0, L]$ is also σ's depth, and, vice-versa, each such permutation π has a corresponding node σ_π. We will further use this node-permutation identification, while referring to nodes.

Given EIG tree $T_{N,L}$, an attribute is a function $\aleph : T_{N,L} \to V$, for some value set V; alternatively, $\aleph$ can be given as a functional subset of $\{\pi \in P(N, t) \mid t \in$

$[0, L]\} \times V$. The classical EIG-based Byzantine algorithm uses two attributes: (i) a top-down attribute *val*, here called α; and (ii) a bottom-up attribute *newval*, here called β.

Figure 1 illustrates three isomorphic EIG trees, (a) $T^2_{4,2}$, (b) $T^3_{4,2}$, (c) $T^4_{4,2}$. As we will see next, theses are the EIG trees built by non-faulty processes 2, 3, 4 (respectively) in our sample scenario 5.1, where process 1 is Byzantine-faulty (so its own internal structure is irrelevant).

Consider EIG tree 1.b, for process 3, $T^3_{4,2}$. Level 0 corresponds to permutation set $\{\lambda\}$. Level 1 corresponds to permutation set $\{(1), (2), (3), (4)\}$. Level 2 corresponds to permutation set $\{(1,2), (1,3), (1,4), (2,1), (2,3), (2,4), (3,1), (3,2), (3,4), (4,1), (4,2), (4,3)\}$. This tree is decorated with two attributes, α and β. Using the alternate notation for permutations (to avoid embedded parentheses), attribute α corresponds to the functional relation $\{(\lambda, 1), (1, 0), (2, 0), (3, 1), (4, 1), (1.2, 0), (1.3, 0), (1.4, 1), (2.1, 0), (2.3, 0), (2.4, 0), (3.1, 0), (3.2, 1), (3.4, 1), (4.1, 1), (4.2, 1), (4.3, 1)\}$.

5 The EIG-based Byzantine agreement algorithm

Each process starts with its *own* initial decision choice. At the end, all non-faulty processes must take the same final decision, even if the faulty processes attempt to disrupt the agreement, accidentally or intentionally.

The classical EIG-based algorithm solves the Byzantine agreement problem in the *binary decision* case (true $= 1$, false $= 0$), for N *processes*, connected in a *complete graph* (where edges indicate *reliable duplex communication lines*), provided that $N \geq 3F+1$, where F is the maximum number of faulty processes. This is a *synchronous* algorithm; celebrated results (see for example [13]) show that the Byzantine agreement is *not* possible if $N \leq 3F$, in the *asynchronous* case or when the communication links are *not* reliable.

Without providing a complete description, we provide a sketch of the classical algorithm, *reformulated* on the basis of the theoretical framework introduced in Section 4. For a more complete and verbose description of this algorithm, including correctness and complexity proofs, we refer the reader to Lynch [13].

Each non-faulty process, h, has its own copy of an EIG tree, $T^h_{N,L}$, where $L = F + 1$. This tree is decorated with two attributes, $\alpha^h, \beta^h : \{\pi \in P(N, t) \mid t \in [0, L]\} \rightarrow \{0, 1, \texttt{null}\}$, where `null` designates undefined items (not yet evaluated). Attributes α^h and β^h are also known as val_h and $newval_h$ [13], or *top-down* and *bottom-up* [9]. As their alternative names suggest, α^h is first evaluated, in a top-down tree traversal, in increasing level order; next, β^h is evaluated, in a bottom-up traversal, in decreasing level order.

The algorithm works in two phases. Its *first phase* is a *messaging* phase which completes the evaluation of the top-down attribute α^h. Initially, $\alpha^h(\lambda) = v^h$, the initial choice of process h; all the other α^h and β^h values are still undefined. Next, there are L messaging rounds. At round $t \in [1, L]$, h broadcasts to all processes (including self), a reversibly encoded message which identifies its α^h values at level $t - 1$ and their EIG destinations. Here, we encode all these via the set

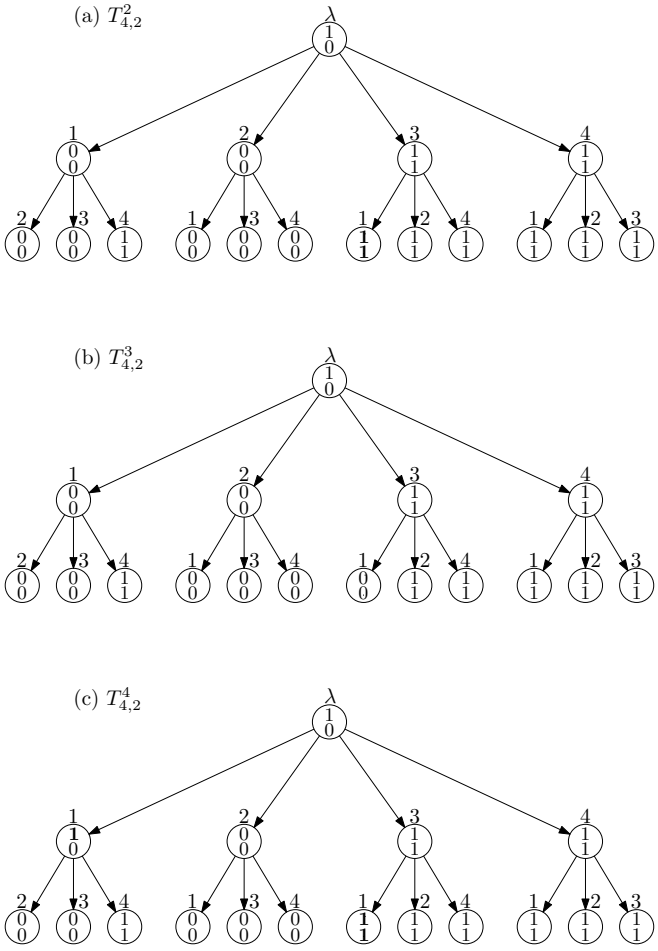


Fig. 1: Three sample EIG trees, $T^h_{4,2}$, $h \in \{2, 3, 4\}$, completed with two attributes, α and β. The node labels appear besides the node blob. Each node blob contains its two attribute values: the top-down α value at the top, and the bottom-up β value at the bottom.

$\{(\pi \odot h, \alpha^h(\pi)) \mid \pi \in P(N, t-1), h \notin \mathrm{Im}(\pi)\}$. All other non-faulty processes broadcast messages, in a similar way. More compact encodings are possible, but we don't follow this issue here.

Process h decodes and processes the messages that it receives. From process f, $f \in [1, N]$, process h receives the set $\{(\pi \odot f, \alpha^f(\pi)) \mid \pi \in P(N, t-1)\}$. Each item $(\pi \odot f, \alpha^f(\pi))$ is used to assign further α^h values, to the next level down the EIG tree, by setting $\alpha^h(\pi \odot f) = \alpha^f(\pi)$.

As this formula suggests, it is indeed *critical* that h "knows" the origin f of each received message and that this origin mark cannot be faked by faulty

processes. Wrong or missing values are replaced by the value of a predefined default parameter, $v_0 \in \{0, 1\}$. Thus, there are L messaging rounds and, after the last round, all nodes are decorated with values of attribute α. In fact, only the last level α values are actually needed, to start the next phase, a practical implementation can choose to discard the other α values.

Then, the algorithm switches to its *second phase*, the evaluation of the bottom-up attribute β^h. First, for leaves, $\beta^h(\pi) = \alpha^h(\pi), \pi \in P(N, L)$.

Next, given β^h values for level $t \in [1, L]$, each β^h value for the next level up, $\beta^h(\pi), \pi \in P(N, t-1)$, is evaluated on the basis of the β^h values of node π's children, i.e. on the multiset $\{\beta^h(\pi \odot f) \mid f \in [1, N] \setminus \mathrm{Im}(\pi)\}$, using a local majority voting scheme: $\beta^h(\pi) = 0$, if a strict majority of the above multiset values are 0; or, $\beta^h(\pi) = 1$, if a strict majority of the above multiset values are 1; or, $\beta^h(\pi) = v_0$ (the same default parameter mentioned above), if there is a tie.

At the end, the β^h value for the EIG root, $\beta^h(\lambda)$, is process h's final decision. All non-faulty processes will simultaneously reach the same final decision; any decision taken by faulty nodes is not relevant.

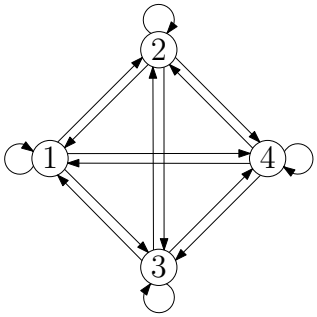
Example 5.1 (Sample Byzantine scenario). Consider a Byzantine scenario with $N = 4$ and $F = 1$, thus $L = 2$. Assume that processes 1, 2, 3 and 4 start with initial choices 0, 0, 1, and 1, respectively. Further, assume that process 1 is faulty. Figure 2 shows sample messages which could be exchanged in this scenario and Figure 1 shows the corresponding EIG trees, for non-faulty processes 2,3,4.

Each of the non-faulty processes, 1, 2 and 3, broadcasts identical messages to each of the four processes. The faulty process 1 sends conflicting messages. In our scenario, $x = 0$, in the message sent to 1, 2 and 3, but $x = 1$, in the message sent to 4. Also, $y = 1$, in the message sent to 1, 2 and 4, but $y = 0$, in the message sent to 3. White spaces are placeholders indicating potential messages which are not created, because they would have contained duplicated process numbers (1.1, 2.2, 3.3, 4.4). The second phase is not detailed here, except the common final decisions (the question mark indicates an irrelevant value).

The second phase is illustrated in Figure 1, for all non-faulty processes $2, 3, 4$. All three EIG tress are shown completed all attribute values. Consider the EIG tree (b) owned by process 3, $T^3_{4,2}$. The α^3 values are filled from messages received in the two messaging rounds, as indicated in Figure 2.

The β^3 values are evaluated as required by the algorithm, by a local majority voting scheme. The evaluation of $\beta^3(\lambda)$ reaches a tie, on multiset $\{0, 0, 1, 1\}$, which has two 0's and two 1's; this tie is broken using the default value, here we assume $v_0 = 0$. Thus, $\beta^3(\lambda) = 0$ is the final decision of process 3, which is different from its initial choice, $\alpha^3(\lambda) = 1$.

A similar argument shows that all other non-faulty processes, 2 and 4, end with the same final decision, 0, thereby achieving the required agreement, despite starting with different initial choices and the conflicting messages sent by faulty process 1. Briefly, the Byzantine-faulty process may sometimes affect the outcome (between 0 and 1), but cannot affect the consensus: all non-faulty processes will take the same final decision.



Process	1	2	3	4
Initial choice	0	0	1	1
Faulty	Yes	No	No	No
Round 1 messages	$(1, x)$	$(2, 0)$	$(3, 1)$	$(4, 1)$
Round 2 messages	$(2.1, 0)$ $(3.1, y)$ $(4.1, 1)$	$(1.2, 0)$ $(3.2, 1)$ $(4.2, 1)$	$(1.3, 0)$ $(2.3, 0)$ $(4.3, 1)$	$(1.4, 1)$ $(2.4, 0)$ $(3.4, 1)$
Final decision	?	0	0	0

Fig. 2: A sample Byzantine scenario, $N = 4$, $F = 1$, where process 1 is Byzantine faulty. Process 1 sends out syntactically correct but different messages to the non-faulty processes: $x = 0, y = 1$ to process 2; $x = 0, y = 0$ to process 3; $x = 1, y = 1$ to process 4. As shown in Figure 1, non-faulty processes 2, 3, 4 build different EIG trees, but they still reach the same final decision.

6 Revised Byzantine agreement solution

Each non-faulty node (process) is modelled by a subsystem which combines $N + 1$ cells: one "main" cell, plus one "firewall" cell for each process (including one for itself). The EIG tree evaluation functionality is localized into the main cell. The main cells communicate only via their associated firewall cells. Figure 3 illustrates the communication digraph for the particular case $N = 4$. The whole subsystem corresponding to a faulty process (1 in our example), may be replaced by any arbitrary entity.

The system evolves along 9 states, for the main cells, and 5 states, for the firewall cells. Figure 4 gives a bird's eye view of this process, for the particular case $N = 4$. Both kind of cells start in state S_0. In state S_0, main cells, μ_i, build the root of their EIG trees.

The *first* (*messaging*) phase is covered by L repetitions of the state cycle S_1, S_2, S_3, S_4. In state S_1, main cells, μ_i, broadcast their outgoing θ' messages to their local firewalls, ν_{ij}. In state S_2, local firewalls, ν_{ij}, forward isomorphic messages θ'' to their partners firewalls, ν_{ji}. In state S_3, local firewalls, ν_{ij}, forward isomorphic messages θ' to their main cells, μ_i. In state S_4, main cells, μ_i, use incoming θ' messages to build the next level of their EIG trees. The messaging phase ends after L messaging rounds, when all cells, μ_i and ν_{ij}, enter state S_5. State S_5 is the end state for the firewall cells, ν_{ij}.

In state S_5, all main cells, μ_i, have computed all top-down attributes, α, and start the *second* phase, i.e. the bottom-up evaluation of attributes β. This second phase ends after L repetitions of the state cycle S_6, S_7, S_8, when all main cells, μ_i, enter state S_9. Each bottom-up cycle computes a new level of the β

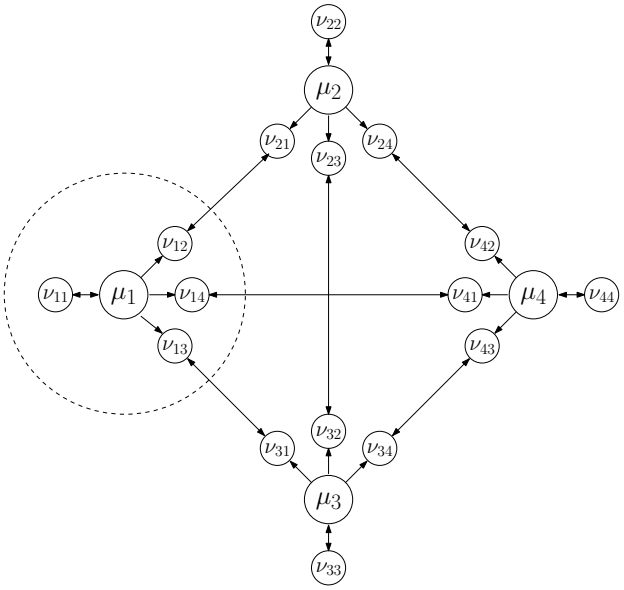


Fig. 3: A sample Byzantine problem, $N = 4$, modelled as a cP system digraph with N subsystems, N main cells (μ_i) and N^2 firewall cells (ν_{ij}), $i, j \in [1, N]$. The dashed blob delimits the subsystem of a possibly faulty process (here 1).

attributes. At the end, the final decision value is given by the β attribute of the EIG root.

7 Initial configurations

Figure 5 lists the initial multiset for the main cell μ_i, i.e. for process $i \in \{1, 2, \ldots, n\}$. Process IDs are encapsulated in ι (*iota*) sub-subcells. Subcell $\bar{\mu}$ contains the ID of the current process, i, i.e. $\iota(1^i)$. Subcell $\bar{\pi}$ contains the set of all process IDs, i.e. $\{1, 2, \ldots, n\}$, given as the multiset $\{\iota(1^1), \iota(1^2), \ldots \iota(1^n)\}$. Subcell $\bar{\ell}$ represents l, the maximum number of levels for the EIG tree; this is computed as $l = \lceil (n-1)/3 \rceil$. Subcells $\bar{\delta}$ contain the two admissible decision values, here t (true=1) and f (false=0). Subcell $\bar{\alpha}$ contains $v_i \in \{t, f\}$, process i's initial choice. Subcell $\bar{v}_0$ contains $v_0 \in \{t, f\}$, i.e. the default value, known by all processes. All these initial symbols will be immutable.

Figure 6 lists the initial multiset for the firewall cell ν_{ij} (serving μ_i) to its partner firewall cell ν_{ji} (serving μ_j), for $i, j \in \{1, 2, \ldots, n\}$. Subcell $\bar{\mu}$ contains the ID of the current process, here $\iota(1^i)$. Subcell $\bar{\nu}$ contains the ID of the other linked process, here $\iota(1^j)$. As in main cell μ_i, subcell $\bar{\ell}$ represents l, the maximum number of levels for the EIG tree. All these initial symbols will be immutable.

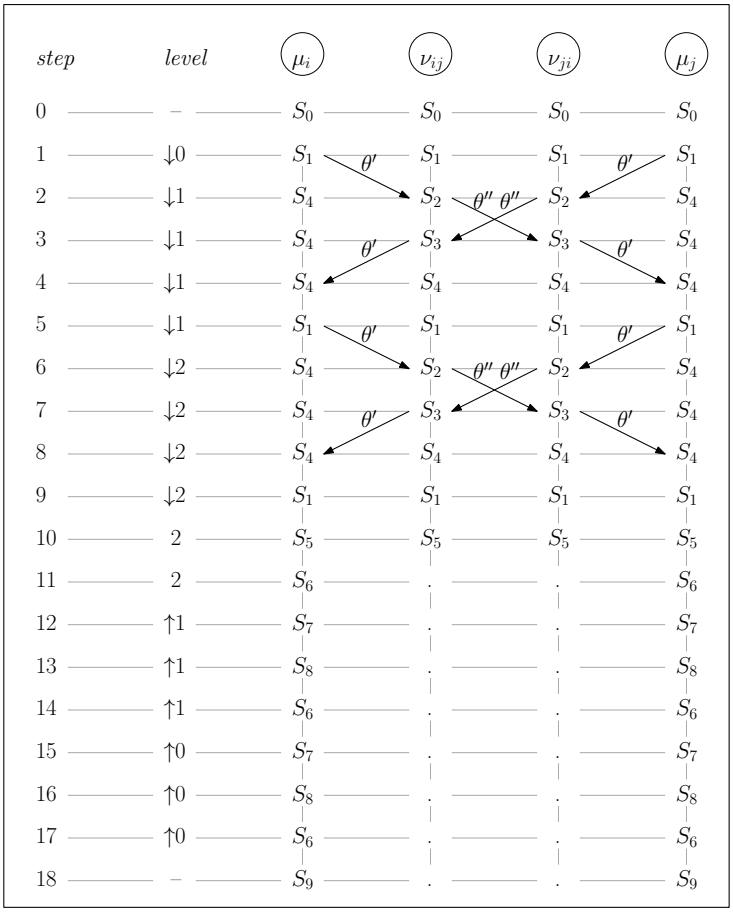


Fig. 4: State and interaction chart. Here: (i) μ_i, μ_j are the main cells of two non-faulty processes, i, j; (ii) ν_{ij}, ν_{ji} are firewall cells at the end of the communication link between i and j.

$$\{\ \bar{\pi}(\iota(1^1)\ \iota(1^2)\ \ldots\ \iota(1^n)),\ \bar{\mu}(\iota(1^i)),\ \bar{\ell}(1^l),\ \bar{\delta}(t),\ \bar{\delta}(f),\ \bar{\alpha}(v_i),\ \bar{v}_0(v_0)\ \}$$

Fig. 5: Initial multiset for the main cell μ_i, i.e. for process $i \in \{1, 2, \ldots, n\}$.

$$\{\ \bar{\mu}(\iota(1^i)),\ \bar{\nu}(\iota(1^j)),\ \bar{\ell}(1^l)\ \}$$

Fig. 6: Initial multiset for the firewall cell ν_{ij} (serving μ_i) to its partner firewall cell ν_{ji} (serving μ_j), for $i, j \in \{1, 2, \ldots, n\}$.

8 Rules for messaging phase

Sending messages. Figure 7 lists the ruleset which completes the initialisation of main cell μ_i and then sends out expected round messages, exiting to state S_5 at the end of the messaging phase. Subcell ℓ gives the current level in the EIG tree. Subcell θ represents a node in the EIG tree during the top-down messaging phase. Sub-subcells θ/ℓ, θ/α, θ/π, and θ/ρ respectively indicate the node level, its top-down α value, its associated permutation (i.e. EIG branch), and the set of all process numbers appearing in π (i.e. $\mathrm{Im}(\pi)$). Subcell θ' represents an outgoing message and has sub-subcells isomorphic to θ (except that it does not have a ρ sub-subcell).

Technically, subcell θ/π is a list of process IDs and subcell θ/ρ is the corresponding set of process IDs; both are initially empty. For example, the EIG branch 3.1 with top-down attribute $\alpha = 0$ is represented by subcell $\theta(\ell(2)\ \pi(\iota(1)\,|\,\iota(3))\ \rho(\iota(1)\,\iota(3))\ \alpha(0))$. Conceptually, the information hold by subcells θ/ℓ and θ/ρ is redundant, as it can be computed from π; however, their presence simplifies and speeds up the overall evolution.

Rule (0) constructs a zero current level and the root of the EIG tree. Rule (1) exists to state S_5 when all required messaging rounds (tree levels) have been completed. Rule (2) sends out messages θ', based on values from node θ. The outgoing messages already indicate their destination: the included level is already increased by one and the current process id is prepended to the branch permutation. Rule (3) increments the current level.

Firewall. Figure 8 lists the ruleset of the firewall cell ν_{ij}. During the messaging phase, its states are exactly synchronised to the states of its main cell, μ_i. Messages θ' are received or assumed to be received from its main cell, μ_i, and are forwarded as θ'' to its partner firewall cell, ν_{ji} (associated to main cell μ_j). Messages θ'' are assumed to be received from the partner firewall cell, ν_{ji}; θ'' are forwarded to the main cel, μ_i, as θ' – but *only if* the permutation head properly identifies the other process (so the sender's identity cannot be faked). No

S_0	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_1\ \ell(0)\ \theta(\ell(0)\ \pi(\vert)\ \rho()\ \alpha(V))$ $\parallel \bar{\alpha}(V)$	(0)
S_1	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	S_5 $\parallel \bar{\ell}(L)$ $\parallel \ell(L)$	(1)
S_1	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	$S_2\ \theta'(\ell(L1)\ \pi(X\vert P)\ \alpha(V))\downarrow_\forall$ $\parallel \bar{\mu}(X)$ $\parallel \ell(L)$ $\parallel \theta(\ell(L)\ \pi(P)\ \alpha(V)\ \rho(\_))$	(2)
$S_1\ \ell(L)$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_4\ \ell(L1)$	(3)

Fig. 7: Ruleset which completes the initialisation of the main cells and then (repeatedly) sends out expected round messages, exiting to state S_5 at the end of the messaging phase.

other messages can reach the main cell (this explains why these cells are called "firewall").

Subcell ℓ gives the current "reverse" level in the EIG tree – which is decremented from maximum to zero (instead of being incremented the other way). Rule (0) constructs this "reverse' current level. At the end of the messaging phase, rule(1) exits to end state S_5. Rule (2) decrements the current level. Rule (3) transforms incoming message θ' into an isomorphic θ'' and forwards it to its partner firewall, ν_{ji}. Rule (4) transforms incoming message θ'' into an isomorphic θ' and forwards it to its main cell, μ_i, provided that the sender is properly recorded as the permutation head. Rules (5) and (6) keep the synchronisation with main cell, μ_i.

S_0	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_1\ \ell(L) \parallel \bar{\ell}(L)$	(0)
$S_1\ \ell()$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	S_5	(1)
$S_1\ \ell(L1)$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_2\ \ell(L)$	(2)
$S_2\ \theta'(T)$	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	$S_3\ \theta''(T)\downarrow_\forall$	(3)
$S_3\ \theta''(\pi(Y\vert P)\ T')$	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	$S_4\ \theta'(\pi(Y\vert P)\ T')\uparrow_\forall \parallel \nu(Y)$	(4)
S_3	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	S_4	(5)
S_4	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	S_1	(6)

Fig. 8: Ruleset of the firewall cell.

Receiving messages. Figure 9 lists the ruleset which receives incoming messages and records these in a new level down in the EIG tree. Note that, after passing the firewall, all these incoming messages have a *correct sender*. Rule (1) records the α attributes of the incoming message θ', whose level and branch permutation match existing nodes. For missing or unmatched (malformed) messages, rule (2) assumes the default value v_0. Rules (3) and (4) delete all previous level θ nodes and incoming θ' messages (not strictly necessary, but keeps the cells clean).

For example, with reference to Figures 2 and 1, consider that process 3 receives from process 1 the second level message $\theta'(\ell(2)\ \pi(\iota(1)\,|\,\iota(3))\ \alpha(0))$. At this stage, process 2 already is in state S_4 and has the following subcells: $\bar{\pi}(\iota(1)\dots)$, $\ell(2)$, $\bar{\delta}(0)$, $\bar{\bar{\delta}}(1)$, $\ell(2)$, $\theta(\ell(1)\ \pi(\iota(3)\,|)\ \rho(\iota(3))\ \alpha(1))$. Then, rule (4) selects the target state S_1 and creates new subcell $\theta(\ell(2)\ \pi(\iota(1)\,|\,\iota(3))\ \rho(\iota(1)\,\iota(3))\ \alpha(0))$ – for EIG branch 3.1 and top-down value 0.

$$
\begin{array}{lll}
S_4 & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_1\ \theta(\ell(L1)\ \pi(Y|P)\ \rho(YQ)\ \alpha(V)) \qquad (1)\\
 & & \|\ \bar{\pi}(Y\_)\\
 & & \|\ \bar{\delta}(V)\\
 & & \|\ \ell(L1)\\
 & & \|\ \theta(\ell(L)\ \pi(P)\ \rho(Q)\ \alpha(\_))\\
 & & \neg\ \theta(\ell(L)\ \pi(P)\ \rho(YQ')\ \alpha(\_))\\
 & & \|\ \theta'(\ell(L1)\ \pi(Y|P)\ \alpha(V))\\
\\
S_4 & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_1\ \theta(\ell(L1)\ \pi(Y|P)\ \rho(YQ)\ \alpha(V)) \qquad (2)\\
 & & \|\ \bar{\pi}(Y\_)\\
 & & \|\ \bar{v}_0(V)\\
 & & \|\ \ell(L1)\\
 & & \|\ \theta(\ell(L)\ \pi(P)\ \rho(Q)\ \alpha(\_))\\
 & & \neg\ \theta(\ell(L)\ \pi(P)\ \rho(YQ')\ \alpha(\_))\\
 & & \neg\ \theta'(\ell(L1)\ \pi(Y|P)\ \alpha(\_))\\
\\
S_4\ \theta(r(L)\ \_) & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_1 \qquad (3)\\
 & & \|\ \ell(L1)\\
\\
S_4\ \theta'(\_) & \rightarrow_{\texttt{max}\otimes\texttt{min}} & S_1 \qquad (4)
\end{array}
$$

Fig. 9: Ruleset to receive a set of messages and record these in a new level down in the EIG tree.

9 Rules for second phase

Figure 10 lists the ruleset which iteratively evaluates the bottom-up β attributes for main cell μ_i. Subcell τ represents a node in the EIG tree during the bottom-up evaluation phase (we could have reused the already existing θ's, but it seems cleaner this way). Sub-subcells τ/ℓ and τ/π have the same meaning as for θ's. Sub-subcell τ/β contains the value of the bottom-up β attribute. Subcell ω stores the final decision (in agreement with all non-faulty processes).

Rule (1) determines β for the leaves of the EIG tree. Rule (2) fires after the β evaluation has reached the EIG root and records the final decision. Rule (3) decrements the EIG level. Rule (4) cancels all pairs of opposite τ/β's, i.e. containing t vs. f. If any t remains, rule (5) decides for t. If any f remains, rule (6) decides for f. Otherwise, rule (7) decides for the default value v_0. Finally, rule (7) deletes all previously existing τ's (again, not necessary, but keeps cells clean).

$S_5\ \theta(\ell(L)\ \pi(P)\ \rho(\_)\ \alpha(V))$	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	$S_6\ \tau(\ell(L)\ \pi(P)\ \beta(V))$ $\parallel\ \ell(L)$	(1)
$S_6\ \ell()$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_9\ \omega(V)$ $\parallel\ \tau(r()\ \pi()\ \beta(V))$	(2)
$S_6\ \ell(L1)$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_7\ \ell(L)$	(3)
$S_7\ \tau(\ell(L1)\ \pi(Y\|P)\ \beta(t))$ $\tau(\ell(L1)\ \pi(Y\|P)\ \beta(f))$	$\rightarrow_{\texttt{max}\otimes\texttt{max}}$	S_8 $\parallel\ \ell(L)$	(4)
$S_7\ \tau(\ell(L1)\ \pi(Y\|P)\ \beta(t))$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_8\ \tau(\ell(L)\ \pi(P)\ \beta(t))$ $\parallel\ \ell(L)$	(5)
$S_7\ \tau(\ell(L1)\ \pi(Y\|P)\ \beta(f))$	$\rightarrow_{\texttt{min}\otimes\texttt{min}}$	$S_8\ \tau(\ell(L)\ \pi(P)\ \beta(f))$ $\parallel\ \ell(L)$	(6)
S_7	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	$S_8\ \tau(\ell(L)\ \pi(P)\ \beta(V))$ $\parallel\ \bar{v}_0(V)$ $\parallel\ \ell(L)$ $\neg\ \tau(\ell(L)\ \pi(P)\ \beta(\_))$	(7)
$S_8\ \tau(\_)$	$\rightarrow_{\texttt{max}\otimes\texttt{min}}$	S_6	(8)

Fig. 10: Ruleset which evaluates the bottom-up attribute β.

10 Static complexity

Table 11 summarizes the main differences between the previous best solution [7, 8] and the current solution. In contrast to the previous solution, this new solution, based on complex symbols and generic rules, uses a small and *fixed* size set of objects, states and rules. The high-level rules map naturally to the main steps of the algorithm described in Sections 4 and 5. In fact, this high-level ruleset compares favourably even with the semi-formal description of these sections, because: (i) it is fully formal; (ii) it is directly executable; and (iii) it seems succinter.

Complexity measure	**Previous version**	**Current version**
Cells per process	$3N + 1$	$N + 1$
Atomic symbols	$O(N!)$	16
States	$O(L)$	$9 + 5$
Rules	$O(N!)$	$16 + 7$

Fig. 11: Summary of complexity measures (where $L = \lceil (N-1)/3 \rceil$).

11 Conclusions and open problems

We have refined our earlier version of P systems with complex symbols. The new version, called cP systems, enables the creation and manipulation of high-level data structures which are typical in high-level languages, such as: relations (graphs), associative arrays, lists. We leveraged these capabilities to present a revised succint version of our previously best solution for the Byzantine agreement problem. In contrast to the previous solution, which uses a super-exponential number of symbols and rules, the new solution uses a *fixed sized* alphabet and ruleset, independent of the problem size, and the ruleset follows closely the conceptual description of the algorithm.

Like other versions of P systems, our cP systems are formal models which can become directly executable, if properly supported by tools. Further research should address this issue, best by formalising the cP semantics.

Dinneen et al. [7, 8] also mention an open problem which is still unsolved. Even if their numbers have been substantially reduced, our solution still requires N^2 additional firewalls cells, one of each side of each communication channel. These firewalls are not conceptually required, as each node could decide which symbols it should accept and when. Further research is needed on this. Besides increasing the speed, a proper solution could more generally bring P systems closer to the Actor model.

Acknowledgments. We are deeply indebted to the co-authors of our former studies on the Byzantine agreement and to the anonymous reviewers, for their most valuable comments and suggestions.

References

1. Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. In: Herbert, A., Birman, K.P. (eds.) SOSP. pp. 59–74. ACM (2005)
2. Ben-Or, M., Hassidim, A.: Fast quantum Byzantine agreement. In: Gabow, H.N., Fagin, R. (eds.) STOC. pp. 481–485. ACM (2005)
3. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. J. Cryptology 18(3), 219–246 (2005)
4. Castro, M., Liskov, B.: Practical Byzantine fault tolerance and proactive recovery. ACM Trans. Comput. Syst. 20(4), 398–461 (2002)
5. Ciobanu, G.: Distributed algorithms over communicating membrane systems. Biosystems 70(2), 123–133 (2003)
6. Ciobanu, G., Desai, R., Kumar, A.: Membrane systems and distributed computing. In: Păun, G., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) WMC-CdeA. Lecture Notes in Computer Science, vol. 2597, pp. 187–202. Springer-Verlag (2002)
7. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: A faster P solution for the Byzantine agreement problem. In: Gheorghe, M., Păun, G., Hinze, T., Rozenberg, G., Salomaa, A. (eds.) 11th International Conference on Membrane Computing (CMC11), Revised Selected Papers. Lecture Notes in Computer Science, vol. 6501, pp. 175–197. Springer (2010)
8. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: A faster P solution for the Byzantine agreement problem. Report CDMTCS-388, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand (July 2010), `http://www.cs.auckland.ac.nz/CDMTCS/researchreports/388-DKN.pdf`
9. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: P systems and the Byzantine agreement. The Journal of Logic and Algebraic Programming 79, 334–349 (2010)
10. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: P systems and the Byzantine agreement. Report CDMTCS-375, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand (January 2010), `http://www.cs.auckland.ac.nz/CDMTCS//researchreports/375Byzantine.pdf`
11. Dinneen, M.J., Kim, Y.B., Nicolescu, R.: A faster P solution for the Byzantine agreement problem. In: Gheorghe, M., Păun, G., Hinze, T. (eds.) Eleventh International Conference on Membrane Computing (CMC11), 24–27 August, 2010, Friedrich Schiller University, Jena (Germany). pp. 167–192 (2015)
12. Lamport, L., Shostak, R.E., Pease, M.C.: The Byzantine generals problem. ACM Trans. Program. Lang. Syst. 4(3), 382–401 (1982)
13. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1996)
14. Martin, J.P., Alvisi, L.: Fast Byzantine consensus. IEEE Trans. Dependable Sec. Comput. 3(3), 202–215 (2006)
15. 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)
16. Nicolescu, R.: Parallel thinning with complex objects and actors. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) 15th International

Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8961, pp. 330–354. Springer (2015)
17. Nicolescu, R.: Structured grid algorithms modelled with complex objects. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Sixteenth Conference on Membrane Computing (CMC16), Revised Selected Papers, Lecture Notes in Computer Science, vol. 9504, pp. 321–337. Springer (2015)
18. Nicolescu, R., Dinneen, M.J., Kim, Y.B.: Towards structured modelling with hyperdag P systems. International Journal of Computers, Communications and Control 2, 209–222 (2010)
19. Nicolescu, R., Ipate, F., Wu, H.: Programming P systems with complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) 14th Conference on Membrane Computing, Revised Selected Papers, Lecture Notes in Computer Science, vol. 8340, pp. 280–300. Springer (2013)
20. Nicolescu, R., Ipate, F., Wu, H.: Towards high-level P systems programming using complex objects. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y. (eds.) 14th International Conference on Membrane Computing, CMC14, Chişinău, Moldova, August 20-23, 2013, Proceedings. pp. 255–276. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova, Chişinău (2013)
21. Nicolescu, R., Wu, H.: Complex objects for complex applications. Romanian Journal of Information Science and Technology 17(1), 46–62 (2014)
22. Pease, M.C., Shostak, R.E., Lamport, L.: Reaching agreement in the presence of faults. J. ACM 27(2), 228–234 (1980)
23. Păun, G., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)

A Appendix
cP Systems : P Systems with Complex Symbols

We present the details of our complex-symbols framework, slightly revised from our earlier papers [16, 17].

A.1 Complex symbols as subcells

Complex symbols play the roles of cellular micro-compartments or substructures, such as organelles, vesicles or cytoophidium assemblies ("snakes"), which are embedded in cells or travel between cells, but without having the full processing power of a complete cell. In our proposal, *complex symbols* represent nested data compartments which have no own processing power: they are acted upon by the rules of their enclosing cells.

Technically, our *complex symbols*, also called *subcells*, are similar to Prolog-like *first-order terms*, recursively built from *multisets* of atoms and variables. *Atoms* are typically denoted by lower case letters (or, occasionally, digits), such as a, b, c, *1*. *Variables* are typically denoted by uppercase letters, such as X, Y, Z. For improved readability, we also consider *anonymous variables*, which are denoted by underscores ("_"). Each underscore occurrence represents a *new*

unnamed variable and indicates that something, in which we are not interested, must fill that slot.

Terms are either (i) simple atoms, or (ii) atoms (called *functors*), followed by one or more parenthesized *multisets* (called *arguments*) of other symbols (terms or variables), e.g. $a(b^2X), a(X^2c(Y)), a(b^2)(c(Z))$. Functors that are followed by more than one parenthesized argument are called *curried* (by analogy to functional programming) and, as we see later, are useful to precisely described deep 'micro-surgical" changes which only affect inner nested symbols, without directly touching their enclosing outer symbols. Terms that do *not* contain variables are called *ground*, e.g.:

- Ground terms: a, $a(\lambda)$, $a(b)$, $a(bc)$, $a(b^2c)$, $a(b(c))$, $a(bc(\lambda))$, $a(b(c)d(e))$, $a(b(c)d(e))$, $a(b(c)d(e(\lambda)))$, $a(bc^2d)$; or, a curried form: $a(b^2)(c(d)e^3)$.
- Terms which are not ground: $a(X)$, $a(bX)$, $a(b(X))$, $a(XY)$, $a(X^2)$, $a(XdY)$, $a(Xc())$, $a(b(X)d(e))$, $a(b(c)d(Y))$, $a(b(X)d(e(Y)))$, $a(b(X^2)d(e(Xf^2)))$; or, a curried form: $a(b(X))(d(Y)e^3)$; also, using anonymous variables: $a(b\_)$, $a(X\_)$, $a(b(X)d(e(\_)))$.

Note that we may abbreviate the expression of complex symbols by removing inner λ's as explicit references to the empty multiset, e.g. $a(\lambda) = a()$.

Complex symbols (subcells, terms) can be formally defined by the following grammar:

```
<term> ::= <atom> | <functor> ( '(' <argument> ')' )+
<functor> ::= <atom>
<argument> ::= λ | ( <term-or-var> )+
<term-or-var> ::= <term> | <variable>
```

Unification. All terms (ground or not) can be (asymmetrically) *matched* against *ground* terms, using an ad-hoc version of *pattern matching*, more precisely, a *one-way first-order syntactic unification*, where an atom can only match another copy of itself, and a variable can match any bag of ground terms (including the empty bag, λ). This may create a combinatorial *non-determinism*, when a combination of two or more variables are matched against the same bag, in which case an arbitrary matching is chosen. For example:

- Matching $a(b(X)fY) = a(b(cd(e))f^2g)$ deterministically creates a single set of unifiers: $X, Y = cd(e), fg$.
- Matching $a(XY^2) = a(de^2f)$ deterministically creates a single set of unifiers: $X, Y = df, e$.
- Matching $a(XY) = a(df)$ non-deterministically creates one of the following four sets of unifiers: $X, Y = \lambda, df$; $X, Y = df, \lambda$; $X, Y = d, f$; $X, Y = f, d$.

Performance note. If the rules avoid any matching non-determinism, then this proposal should not affect the performance of P simulators running on existing machines. Assuming that bags are already taken care of, e.g. via hash-tables,

our proposed unification probably adds an almost linear factor. Let us recall that, in similar contexts (no occurs check needed), Prolog unification algorithms can run in $O(ng(n))$ steps, where g is the inverse Ackermann function. Our conjecture must be proven though, as the novel presence of multisets may affect the performance.

A.2 Generic rules

Rules use *states* and are applied top-down, in the so-called *weak priority* order. Rules may contain *any* kind of terms, ground and not-ground. In *concrete* models, *cells* can only contain *ground* terms. *Cells* which contain *unground* terms can only be used to define *abstract* models, i.e. high-level patterns which characterise families of similar concrete models.

Pattern matching. Rules are matched against cell contents using the above discussed *pattern matching*, which involves the rule's left-hand side, promoters and inhibitors. Moreover, the matching is *valid* only if, after substituting variables by their values, the rule's right-hand side contains ground terms only (so *no* free variables are injected in the cell or sent to its neighbours), as illustrated by the following sample scenario:

- The cell's *current content* includes the *ground term*:
  $n(a\,\phi(b\,\phi(c)\,\psi(d))\,\psi(e))$
- The following *rewriting rule* is considered:
  $n(X\,\phi(Y\,\phi(Y_1)\,\psi(Y_2))\,\psi(Z)) \;\rightarrow\; v(X)\,n(Y\,\phi(Y_2)\,\psi(Y_1))\,v(Z)$
- Our pattern matching determines the following *unifiers*:
  $X = a$, $Y = b$, $Y_1 = c$, $Y_2 = d$, $Z = e$.
- This is a *valid* matching and, after *substitutions*, the rule's *right-hand* side gives the *new content*:
  $v(a)\;\; n(b\,\phi(d)\,\psi(c))\;\; v(e)$

Generic rules format. We consider rules of the following *generic* format (we call this format generic, because it actually defines templates involving variables):

$$\begin{array}{lll} \textit{current-state}\;\; \textit{symbols}\ldots & \rightarrow_\alpha & \textit{target-state}\;\; (\textit{in-symbols})\ldots \\ & & (\textit{out-symbols})_\delta\ldots \\ & & \mid \textit{promoters}\ldots\;\; \neg\, \textit{inhibitors}\ldots \end{array}$$

Where:

- All *symbols*, including *states*, *promoters* and *inhibitors*, are *multisets of terms*, possibly containing *variables* (which can be *matched* as previously described).
- Parentheses can be used to clarify the association of symbols, but otherwise have no own meaning.

– Subscript $\alpha \in \{\texttt{min}, \texttt{max}\} \times \{\texttt{min}, \texttt{max}\}$, indicates a combined *instantiation* and *rewriting* mode, as further discussed in the example below.
– *In-symbols* become available after the end of the current step only, as in traditional P systems (we can imagine that these are sent via an ad-hoc fast *loopback* channel);
– *Out-symbols* are sent, at the end of the step, to the cell's structural neighbours. These symbols are enclosed in round parentheses which further indicate their destinations, above abbreviated as δ. The most usual scenarios include:
  • $(a)\downarrow_i$ indicates that a is sent to child i (unicast);
  • $(a)\uparrow_i$ indicates that a is sent to parent i (unicast);
  • $(a)\downarrow_\forall$ indicates that a is sent to all children (broadcast);
  • $(a)\uparrow_\forall$ indicates that a is sent to all parents (broadcast);
  • $(a)\updownarrow_\forall$ indicates that a is sent to all neighbours (broadcast).

  All symbols sent via one *generic rule* to the same destination form one single *message* and they travel together as one single block (even if the generic rule has multiple instantiations).

Example. To explain our combined instantiation and rewriting mode, let us consider a cell, σ, containing three counter-like complex symbols, $c(1^2)$, $c(1^2)$, $c(1^3)$, and the four possible instantiation⊗rewriting modes of the following "decrementing" rule:

$$(\rho_\alpha)\ S_1\ c(1\,X) \to_\alpha S_2\ c(X), \text{where } \alpha \in \{\texttt{min},\texttt{max}\} \times \{\texttt{min},\texttt{max}\}.$$

1. If $\alpha = \texttt{min}\otimes\texttt{min}$, rule $\rho_{\texttt{min}\otimes\texttt{min}}$ nondeterministically generates and applies (in the $\texttt{min}$ mode) *one* of the following two rule instances:

$$(\rho'_1)\ \ S_1\ c(1^2) \to_{\texttt{min}} S_2\ c(1) \quad \text{or}$$
$$(\rho''_1)\ \ S_1\ c(1^3) \to_{\texttt{min}} S_2\ c(1^2).$$

   Using (ρ'_1), cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^3)$. Using (ρ''_1), cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.
2. If $\alpha = \texttt{max}\otimes\texttt{min}$, rule $\rho_{\texttt{max}\otimes\texttt{min}}$ first generates and then applies (in the $\texttt{min}$ mode) the following *two* rule instances:

$$(\rho'_2)\ \ S_1\ c(1^2) \to_{\texttt{min}} S_2\ c(1) \quad \text{and}$$
$$(\rho''_2)\ \ S_1\ c(1^3) \to_{\texttt{min}} S_2\ c(1^2).$$

   Using (ρ'_2) and (ρ''_2), cell σ ends with counters $c(1)$, $c(1^2)$, $c(1^2)$.
3. If $\alpha = \texttt{min}\otimes\texttt{max}$, rule $\rho_{\texttt{min}\otimes\texttt{max}}$ nondeterministically generates and applies (in the $\texttt{max}$ mode) *one* of the following rule instances:

$$(\rho'_3)\ \ S_1\ c(1^2) \to_{\texttt{max}} S_2\ c(1) \quad \text{or}$$
$$(\rho''_3)\ \ S_1\ c(1^3) \to_{\texttt{max}} S_2\ c(1^2).$$

   Using (ρ'_3), cell σ ends with counters $c(1)$, $c(1)$, $c(1^3)$. Using (ρ''_3), cell σ ends with counters $c(1^2)$, $c(1^2)$, $c(1^2)$.

4. If $\alpha = \texttt{max}\otimes\texttt{max}$, rule $\rho_{\texttt{min}\otimes\texttt{max}}$ first generates and then applies (in the `max` mode) the following *two* rule instances:

$$(\rho_4') \quad S_1\ c(1^2) \to_{\texttt{max}} S_2\ c(1) \quad \text{and}$$
$$(\rho_4'') \quad S_1\ c(1^3) \to_{\texttt{max}} S_2\ c(1^2).$$

Using (ρ_4') and (ρ_4''), cell σ ends with counters $c(1)$, $c(1)$, $c(1^2)$.

The interpretation of `min`⊗`min`, `min`⊗`max` and `max`⊗`max` modes is straightforward. While other interpretations could be considered, the mode `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.

If a rule does not contain any non-ground term, then it has only one possible instantiation: itself. Thus, in this case, the instantiation is an *idempotent* transformation, and the modes `min`⊗`min`, `min`⊗`max`, `max`⊗`min`, `max`⊗`max` fall back onto traditional modes `min`, `max`, `min`, `max`, respectively.

Special cases. Simple scenarios involving generic rules are sometimes semantically equivalent to loop-based sets of non-generic rules. For example, consider the rule

$$S_1\ a(x(I)\ y(J)) \ \to_{\texttt{max}\otimes\texttt{min}}\ S_2\ b(I)\ c(J),$$

where the cell's contents guarantee that I and J only match integers in ranges $[1, n]$ and $[1, m]$, respectively. Under these assumptions, this rule is equivalent to the following set of non-generic rules:

$$S_1\ a_{i,j} \ \to_{\texttt{min}} S_2\ b_i\ c_j,\ \forall i \in [1, n], j \in [1, m].$$

However, unification is a much more powerful concept, which cannot be generally reduced to simple loops.

Note. 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.

Benefits. 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 (often *impossible* with only atomic symbols).

A.3 Synchronous vs asynchronous

In our models, we do not make any *syntactic* difference between the synchronous and asynchronous scenarios; this is strictly a *runtime* assumption [15]. Any model is able to run on both the synchronous and asynchronous runtime "engines", albeit the results may differ.

In the *synchronous* scenario of traditional P systems, all rules in a step take together exactly *one* time unit and then all message exchanges (including loopback messages for in-symbols) are performed at the end of the step, in *zero* time (i.e. instantaneously). Alternatively, but logically equivalent, we here consider that rules in a step are performed in *zero* time (i.e. instantaneously) and then all message exchanges are performed in exactly *one* time unit. We prefer the second interpretation, because it allows us to interpret synchronous runs as special cases of asynchronous runs.

In the *asynchronous* scenario, we still consider that rules in a step are performed in *zero* time (i.e. instantaneously), but then, to arrive at its destination, each message may take *any* finite real time in the $(0, 1]$ interval (i.e. travelling times are typically scaled to the travel time of the slowest message). Additionally, unless otherwise specified, we also assume that messages traveling on the same directed arc follow a *FIFO* rule, i.e. no fast message can overtake a slow progressing one. This definition closely emulates the standard definition used for asynchronous distributed algorithms [13]. Clearly, the asynchronous model is highly non-deterministic, but most useful algorithms manage to remain confluent.

In both scenarios, we need to cater for a particularity of P systems, where a cell may remain active after completing its current step and then will automatically start a new step, without necessarily receiving any new message. In contrast, in classical distributed models, nodes may only become active after receiving a new message, so there is no self-activation without messaging. We can solve this issue by (i) assuminging a hidden self-activation message that cells can post themselves at the end of the step (together with the *in-symbols*) and (ii) postulating that such self-addressed messages will arrive not later than any other messages coming from other cells.

Obviously, any algorithm that works correctly in the asynchronous mode will also work correctly in the synchronous mode, but the converse is *not* generally true: extra care may be needed to transform a correct synchronous algorithm into a correct asynchronous one; there are also general control layers, such as *synchronisers*, that can attempt to run a synchronous algorithm on an existing asynchronous runtime, but this does not always work [13].

Complexity measures. We consider a set of basic complexity measures similar to the ones used in the traditional *distributed algorithms* field.

- *Time complexity*: the supremum over all possible running times (which, although not perfect, is the most usual definition for the asynchronous time complexity).
- *Message complexity*: the number of exchanged messages.
- *Atomic complexity*: the number of atoms summed over all exchanged messages (analogous to the traditional bit complexity).

Rewriting P Systems with Flat-splicing Rules

Linqiang Pan[1], Bosheng Song[1,⋆], and K.G. Subramanian[2]

[1] Key Laboratory of Image Information Processing and Intelligent Control, School of Automation, Huazhong University of Science and Technology, Wuhan 430074, Hubei, China
lqpan@mail.hust.edu.cn, boshengsong@hust.edu.cn
[2] Department of Mathematics, Madras Christian College, Tambaram, Chennai 600059 India
kgsmani1948@gmail.com

Abstract. Rewriting P systems, as language generating devices, are one of the earliest classes of P systems with structured strings as objects and the rewriting rules as evolution rules. Flat splicing is an operation on strings, inspired by a splicing operation on circular strings. In this work, we consider a variant of rewriting P systems with only regular or linear rewriting rules and alphabetic flat splicing rules, and the language generative power of rewriting P systems with flat splicing rules in comparison with flat splicing systems and Chomsky hierarchy is investigated.
Keywords: Bio-inspired computing, Membrane computing, P system, Flat splicing, Formal language

1 Introduction

Membrane computing, which was motivated by the organization and functioning of living cells, has grown to a great extent in breadth and depth, at least at the theoretical level, since its introduction by Păun [12, 13] around the year 2000. The novel computing device in this area, known broadly as *P system*, has become a versatile framework in many application problems as well [3]. Among different varieties of P systems, the *rewriting P system* considered as a language generating device, is one of the earliest P system models introduced by Păun [12] with structured strings as objects and rewriting rules as in formal language theory, in order to deal with symbolic computations. Subsequently, several variants with different additional features have been introduced and investigated [2, 4, 8–11].

On the other hand, in the area of DNA computing, Head [5] introduced a novel operation on strings, called *splicing*, while modelling the recombinant behaviour of DNA strings. Inspired by the splicing operation on circular strings [6], Berstel et al. [1] consider an operation on strings called *flat splicing* which "cuts" a string $u = x\alpha\beta y$ between α and β and inserts in u, a string $v = \gamma z\delta$ between α and β as dictated by a flat splicing rule of the form $(\alpha|\gamma - \delta|\beta)$. In particular, when $\alpha, \beta, \gamma, \delta$ are letters of an alphabet or the empty word, the rule is called an *alphabetic flat splicing rule*.

⋆ Corresponding author.

In this work, we consider a variant of rewriting P systems with only regular or linear rewriting rules and alphabetic flat splicing rules. The language generative power of rewriting P systems with flat splicing rules in comparison with flat splicing systems is investigated. Moreover, the languages generated by rewriting P systems with flat splicing rules are also compared with the languages in Chomsky hierarchy. Specifically, we prove that the context-free languages are included in the languages generated by rewriting P systems with two membranes, with a priority relation on the regular rules and initial strings in the regions having length at least one.

2 Preliminaries

We refer to [16] for concepts and results related to formal grammars and languages and to [12, 13, 15] for P systems.

An *alphabet* V is a finite and nonempty set of symbols and a *word* (also called a linear word) w is a finite sequence of symbols belonging to V. The set of all words over V is denoted by V^* which includes the empty word λ (with no symbols) and $V^+ = V^* - \{\lambda\}$. The *length* $|w|$ of a word w is the number of symbols in w counting repetitions.

We denote the families of languages generated by *context-sensitive*, *context-free*, *linear* or *regular grammars* of the Chomsky hierarchy by CSL, CFL, LIN, respectively [16]. The family of finite languages is denoted by FIN.

We recall the notion of *flat splicing* on words, which was considered by Berstel et al. [1]. A flat splicing rule r is of the form $(\alpha|\gamma - \delta|\beta)$, where $\alpha, \beta, \gamma, \delta$ are words over the alphabet V. The words $\alpha, \beta, \gamma, \delta$ are called the *handles* of the rule. When all the four handles of the rule r are letters in V or the empty word, the flat splicing rule r is called *alphabetic*.

For two words $x = u\alpha\beta v$, $y = \gamma w \delta$, $u, v, w \in V^*$, an application of the flat splicing rule $r = (\alpha|\gamma - \delta|\beta)$ to the pair (x, y) yields the word $z = u\alpha\gamma w\delta\beta v$ and we write $(x, y) \vdash_r z$. In other words, an application of the rule r inserts the second word y between α and β in the first word x yielding the word z. When $\alpha = \beta = \gamma = \delta = \lambda$, the flat splicing rule is simply $(\lambda|\lambda - \lambda|\lambda)$ and an application of this kind of rule allows insertion of any word y into any other word x and the insertion can be done anywhere in x.

A *flat splicing system* (FSS) [1] is a triple $\mathcal{S} = (\Sigma, I, R)$, where Σ is an alphabet, I, called initial set, is a set of words over Σ and R is a finite set of flat splicing rules. The FSS $\mathcal{S}$ is respectively called finite, regular or context-free according to the I is a finite set, a regular set or a context-free language. The language L generated by $\mathcal{S}$ is the smallest language containing I and such that for any two words $u, v \in L$ and any rule $r \in R$, if the rule r is applicable to the pair (u, v) and if the word w is obtained on applying the rule r to the pair (u, v), that is, if $(u, v) \vdash_r w$, then w is also in L. When all the flat splicing rules are alphabetic, the FSS is called an *alphabetic flat splicing system* ($AFSS$). The families of languages generated by FSS and $AFSS$ are respectively denoted by

$\mathcal{L}(FSS, X)$ and $\mathcal{L}(AFSS, X)$ for $X = FIN, REG$ or CF according as the initial set is finite, regular or CF.

We illustrate an alphabetic flat splicing system and its work with an example.

Example 1. Consider the alphabetic flat splicing system $\mathcal{S}_1 = (\{a, b, c\}, \{ac, b\}, R_1)$, where

$$R_1 = \{r_1 = (a|a - c|c), r_2 = (b|b - \lambda|\lambda), r_3 = (a|b - \lambda|c).\}$$

Application of the rule r_2 to the pair of words (b, b) inserts (the second) b to the right of (the first b) yielding b^2. Likewise applying the rule r_2 to the pair (b, b^2) or (b^2, b) yields b^3. Note that an application of r_2 to the pair (b^2, b) can insert b to the right of b^2 or between the two $b's$. Thus proceeding like this, the words generated will be of the form b^n, $n \geq 1$. In a similar manner, applying the rule r_1 to the pair (ac, ac) will yield a^2c^2 and continuing this we obtain words of the form a^nc^n, $n \geq 1$. On the other hand, using the rule r_3 to the pairs of the form (a^nc^n, b^m), $n, m \geq 1$, yields the word $a^nb^mc^n$. The language generated by $\mathcal{S}_1$ is

$$L(\mathcal{S}_1) = \{b^n | n \geq 1\} \cup \{a^nc^n | n \geq 1\} \cup \{a^nb^mc^n \mid n, m \geq 1\}.$$

It has been shown in [1] that alphabetic flat splicing rules and context-free initial sets can produce only context-free languages.

Theorem 1. *[1] The language generated by an alphabetic flat splicing system with context-free initial set is context-free.*

Insertion/deletion systems have been investigated in the context of study on models in DNA computing (see, for example, [14], chapter 6) and a number of language theoretical results have been established. In particular, insertion systems have close similarity with flat splicing systems as pointed out in [1]. But it has been shown in [1] that these two systems generate incomparable families of languages. We now recall insertion systems as described in [1].

An insertion system $\gamma = (\Sigma, I, R)$, where Σ is a nonempty alphabet, I, called the initial set, is a finite nonempty set of words over Σ and R is a finite nonempty set of rules, called *insertion rules*, of the form $(u|\beta|v)$, where u, β, v are words over Σ. Given a word w of the form $w = xuvy$, an application of the rule $(u|\beta|v)$ generates the word $w' = xu\beta vy$. The language generated by the system γ is the set of words over Σ obtained by repeated application of the insertion rules, starting with the words in the initial set I. The family of languages generated by insertion systems is denoted by $L(ins)$. An insertion system is alphabetic if the contexts u, v of the rules $(u|\beta|v)$ have length at most 1. The family of languages generated by insertion systems as defined above, is denoted by $\mathcal{L}(ins)$ and by $\mathcal{L}(ains)$ when the insertion systems are alphabetic.

The following results on (alphabetic) insertion systems and flat splicing systems are known [1].

Theorem 2. (i) *The families of languages generated by flat splicing systems and insertion systems are incomparable.*

(ii) *Alphabetic insertion systems always generate context-free languages.*

3 A Rewriting P System with Linear Rewriting Rules and Alphabetic Flat Splicing Rules

We now introduce a cell-like rewriting P system with internal output computing languages of structured strings. The regions of the P system can have regular or linear rewriting rules and/or alphabetic flat splicing rules and initial objects in the regions are only symbols from an alphabet.

Definition 1. *A rewriting P system with linear rules and/or alphabetic flat splicing rules of degree $m \geq 1$ ($RP_m(LIN/AFSR)$) is*

$$\Pi = (V, T, \mu, F_1, \cdots, F_m, R_1, \cdots, R_m, i_o),$$

where

(i) V *is the total alphabet of the system;*
(ii) $T \subset V$ *is the terminal alphabet;*
(iii) μ *is the membrane structure consisting of* m *membranes labelled in a one-to-one way with* $1, \ldots, m$*;*
(iv) $F_1, \ldots, F_m$ *are finite subsets of* V *associated with the* m *regions of* μ *(the elements of* $F_i, 1 \leq i \leq m$*, are called initial symbols);*
(v) $R_1, \ldots, R_m$ *are finite sets of rules associated with the* m *regions of* μ*;*
(vi) i_o *is the label of an elementary membrane of* μ*, called the output membrane.*

A rule in a region can be a linear rewriting rule of the form $A \to \alpha B \beta$ or $A \to \gamma$, $A, B \in V - T$, $\alpha, \beta, \gamma \in V^*$ or an alphabetic flat splicing rule as described earlier. The rules have attached targets *here, out, in* (in general, *here* is omitted). A linear rule in a region rewrites a string in the region as in a Chomsky grammar while an alphabetic flat splicing rule in a region is applied to a pair of strings in the region. If the rewriting rules in the regions are only regular rules of the form $A \to \alpha B$ or $A \to \gamma$, $A, B \in V - T$, $\alpha, \gamma \in V^*$, then we denote the system as $RP_m(REG/AFSR)$.

A computation in a $RP_m(LIN/AFSR)$ is defined in a way similar to a string rewriting P system [12, 13]. A computation starts from an initial configuration defined by the membrane structure with the initial symbols, if any, in the m regions. The rules in a region are used in a nondeterministic maximally parallel manner which means that the strings to evolve and the rules to be applied to them are chosen in a nondeterministic manner, but all strings in all the regions which can evolve at a given step should do it. On the other hand, the application of a linear rule to a string in a region or an alphabetic flat splicing rule to a pair of strings in a region, is sequential in the sense that only one rule is applied to a string or a pair of strings, resulting in an evolved string which is placed in the region indicated by the target associated with the rule used. The target *here* means that the evolved string remains in the same region, *out* means that the evolved string exits the current membrane (if the rule was applied in the skin membrane, then it can exit the system such that strings leaving the system are "lost" in the environment), and *in* means that the string is sent to one of the

directly lower membranes, nondeterministically chosen if there exist several of them (if no internal membrane exists, then a rule with the target indication *in* cannot be used).

A computation is successful only if it stops reaching a configuration where no rule can be applied to the existing strings. The result of a halting computation consists of strings is composed only of symbols from T placed in the output membrane in the halting configuration. The set of all such strings computed (also called generated) by the system Π is denoted by $L(\Pi)$.

The family of all languages $L(\Pi)$ generated by systems Π as above, with at most m membranes, with linear rules and/or flat splicing rules is denoted by $\mathcal{L}(RP_m(LIN/AFSR))$. If the rewriting rules are regular the corresponding family is denoted by $\mathcal{L}(RP_m(REG/AFSR))$.

In order to illustrate the definition of $RP_m(REG/AFSR)$, we give the following example.

Example 2. Consider the $RP_2(REG/AFSR)$

$$\Pi_1 = (\{S_1, S_2, A, B_1, B_2, B_3, a, b, c, d\}, \{a, b, c, d\}, [_1\, [_2\,]_2\,]_1, \{S_1, S_2\}, \emptyset, R_1, R_2, 2),$$

where R_1 consists of the regular rewriting rules

$$S_1 \to B_1, S_1 \to cB_2, S_1 \to acB_3, S_2 \to xyA,$$

$$B_1 \to acB_1, B_2 \to acB_2, B_3 \to acB_3, A \to dbA$$

and the alphabetic flat splicing rules

$$(\lambda|x - A|B_1), (\lambda|x - A|B_2), (\lambda|x - A|B_3),$$

with all the three alphabetic flat splicing rules having target indication "*in*" while R_2 consists of the following regular rewriting rules:

$$A \to \lambda, B_1 \to \lambda, B_2 \to \lambda, B_2 \to d, B_3 \to d.$$

Computation in Π_1 takes place as follows: The region 1 has axiom strings S_1, S_2 while region 2 has none initially. One of the three rules $S_1 \to B_1, S_1 \to cB_2, S_1 \to acB_3$ in region 1 could be applied to the axiom string S_1 initially. If we start with applying the rewriting rule $S_1 \to B_1$ to the axiom string S_1 and follow it by the application of the rule $B_1 \to acB_1$, for certain times, say n times ($n \geq 1$), then this yields the string $(ac)^n B_1$. Simultaneously, in region 1, the axiom string S_2 yields the string $xy(db)^n A$ (for the same n) by the application of the rule $S_2 \to xyA$ followed by the application of the rule $A \to dbA$ for n times. If at this stage the alphabetic flat splicing rule $(\lambda|x - A|B_1)$ in region 1, with target indication "*in*" is applied to the pair $((ac)^n B_1, xy(db)^n A)$, then the string $(ac)^n xy(db)^n AB_1$ is generated and is sent to region 2, where the application of the rules $A \to \lambda, B_1 \to \lambda$ erase the symbols A, B_1 thereby yielding the string $(ac)^n xy(db)^n$.

Likewise, if we start with applying the rewriting rule $S_1 \to cB_2$ to the axiom string S_1 in region 1 and follow it by the application of the rule $B_2 \to acB_2$,

for certain times, say n times ($n \geq 1$), then this yields the string $c(ac)^nB_2$. Again the alphabetic flat splicing rule $(\lambda|x - A|B_2)$ in region 1, with target indication "*in*" can be applied to the pair $(c(ac)^nB_2, xy(db)^nA)$, then the string $c(ac)^nxy(db)^nAB_2$ is generated and is sent to region 2, where the application of the rule $A \to \lambda$, and either $B_2 \to \lambda$ or $B_2 \to d$ erase the symbol A and either erase B_2 or replace it by d thereby yielding the string $c(ac)^nxy(db)^n$ or $c(ac)^nxy(db)^nd$. Generation of strings of the form $(ac)^nxy(db)^nd$ is similar with the computation in region 1 starting with applying the rule $S_1 \to acB_3$ to the axiom string S_1 and following it by the application of the rule $B_3 \to acB_3$ certain number of times, so that the alphabetic flat splicing rule $(\lambda|x-A|B_3)$ applied to the pair $(ac(ac)^nB_3, xy(db)^nA)$ will yield and send the string $ac(ac)^nxy(db)^nAB_3$ to region 2, where A is erased and B_3 is replaced by d. Thus the language L generated by Π_1 is $L = L(\Pi_1) = M \cup cM \cup cMd \cup acMd$, where $M = \{(ac)^nxy(db)^n|n \geq 0\}$.

Remark 1. The language L in Example 2 is in fact considered in [1] and an alphabetic flat splicing system with six rules is given generating L. It has also been shown that L cannot be generated by an insertion system.

We now examine the generative power of the rewriting P systems with regular and/or flat splicing rules.

Theorem 3. *(i)* $\mathcal{L}(RP_1(REG/AFSR)) - \mathcal{L}(FSS, FIN) \neq \emptyset$*;*
(ii) $\mathcal{L}(RP_1(REG/AFSR)) - \mathcal{L}(ins) \neq \emptyset$*.*

Proof. The language $L_1 = \{xa^ny|n \geq 0\}$ over the alphabet $\{x, a, y\}$ is not in the family $\mathcal{L}(FSS, FIN)$ as shown in [1]. But clearly, it is in $RP_1(REG/AFSR)$ since we can have a $RP_1(REG/AFSR)$ with only one membrane containing regular rules $S \to xA, A \to aA, A \to y$ with the initial symbol S generating L_1. We do not need any alphabetic splicing rules in the membrane. This proves statement (i).

In order to prove statement (ii), we note that the language $L = M \cup cM \cup cMd \cup acMd$ is shown to be not $\mathcal{L}(ins)$ in [1], where $M = \{(ac)^nxy(db)^n|n \geq 0\}$ in Example 2. But it is in $RP_1(REG/AFSR)$ since we can have a $RP_1(REG/AFSR)$ with only membrane containing initial symbols S, a, b, c, d, a regular rule $S \to xy$ and the following six alphabetic flat splicing rules (as in [1]) $(c|x - y|\lambda), (\lambda|c - y|d), (a|c - d|\lambda), (\lambda|a - d|b), (c|a - b|\lambda), (\lambda|c - b|d)$. In fact, these alphabetic splicing rules alone are shown to generate L in [1], but their initial set has xy (a string of length more than 1) and so we have included the rule $S \to xy$ in the membrane to generate xy. Note that we have defined the initial objects in the regions of the P system to be only symbols (of length 1). This proves statement (ii). □

Theorem 4. *(i)* $REG \subset \mathcal{L}(RP_1(REG/AFSR)) \subset \mathcal{L}(RP_2(REG/AFSR))$*;*
(ii) $\mathcal{L}(RP_2(REG/AFSR)) - \mathcal{L}(FSS, REG) \neq \emptyset$*;*
(iii) $\mathcal{L}(RP_2(REG/AFSR))$ *contains a context-sensitive language which is not context-free. As a consequence,* $\mathcal{L}(RP_2(REG/AFSR)) - CSL \neq \emptyset$*.*

Proof. The inclusion $REG \subseteq \mathcal{L}(RP_1(REG/AFSR))$ is straightforward as the regular rules generating a regular language can be taken as the rules in the only one membrane of a corresponding $RP_1(REG/AFSR)$ (with no alphabetic flat splicing rule in the region) and the start symbol of the grammar is the initial object in the membrane. The proper inclusion in statement (i) follows by noting that the language L in the proof of statement (ii) of Theorem 3 is a non-regular language but is in $\mathcal{L}(RP_1(REG/AFSR))$.

In order to prove statement (ii), we note that the inclusion $\mathcal{L}(RP_1(REG/AFSR)) \subseteq \mathcal{L}(RP_2(REG/AFSR))$ holds by definition while the proper inclusion is seen by considering the non-regular context-free language $L_2 = \{xa^nb^ny | n \geq 1\}$ over the alphabet $\{x, y, a, b\}$. In fact, the following $RP_2(REG/AFSR)$, Π_2, generates L:

$$\Pi_2 = (\{S_1, S_2, A, B, x, y, a, b\}, \{x, y, a, b\}, [_1\, [_2\,]_2\,]_1, \{S_1, S_2\}, \emptyset, R_1, R_2, 2),$$

where R_1 consists of the regular rules $S_1 \to xA, A \to aA, S_2 \to B, B \to bB$ and the alphabetic flat splicing rule $(a|\lambda - B|A)$ with target in. R_2 consists of the regular rules $A \to a, B \to \lambda$.

In fact, in region 1, the initial symbols S_1, S_2 respectively generate xa^nA and b^nB (for the same $n \geq 1$) and if at this stage, the alphabetic flat splicing rule is applied to the pair (xa^nS_1, b^nB), then the string xa^nAb^nB is generated and is sent to region 2. Here in region 2, the application of the rules $A \to a, B \to \lambda$ yields the string xa^nb^ny.

But the language L_2 cannot be generated by any $RP_1(REG/AFSR)$ with only one membrane. In fact, regular rules alone are not enough as the language is non-regular while alphabetic flat splicing rules alone are not enough which can be shown by an argument similar to the one given in [1] in proving that the language $\{xa^ny | n \geq 0\}$ is not a flat splicing language. On the other hand, if we assume that the only membrane has some regular rules and some flat splicing rules that can generate the words xa^nb^ny, then strings with some powers of a or powers of b cannot be inserted independently between x and y. The only possibility is to insert between x and y for some string having the form a^nb^n with suitable nonterminals, if any, in between. Since the nonterminals are to be erased ultimately, the only membrane should have rules which will lead to terminal strings of the required form. But this would mean that strings (without the symbols x, y) not in the language will be generated. This shows only one membrane is not enough.

In order to prove statement (iii), consider the language

$$L_3 = \{c^p \mid p \geq 1\} \cup \{a^nb^n \mid n \geq 1\} \cup \{a^nb^nc^{n+m} \mid n, m \geq 1\},$$

which is context-sensitive and not context-free. The following $RP_2(REG/AFSR)$ generates the language L_3:

$$(\{S_1, S_2, S_3, a, b, c\}, \{a, b, c\}, [_1\, [_2\, [_3\,]_3\,]_2\,]_1, \{S_1, S_2\}, \{S_3\}, R_1, R_2, 2),$$

where R_1 consists of the regular rules $S_1 \to aS_1, S_2 \to bS_2$ and an alphabetic flat splicing rule $(a|b - S_2|S_1), in$ while R_2 consists of the regular rules $S_1 \to \lambda, S_2 \to \lambda, S_3 \to \lambda, S_3 \to cS_3$ and an alphabetic flat splicing rule $(S_2|c - S_3|S_1), in$.

The computation takes place as follows: In region 1, S_1, S_2 generate the strings a^nS_1 and b^nS_2, while at the same time in region 2, S_3 generates c^nS_3. At this point if the alphabetic flat splicing in region 1 takes place on the pair (a^nS_1, b^nS_2), then the string generated is $a^nb^nS_2S_1$ which is sent to region 2. Here application of the regular rule $S_3 \to cS_3$ could take place, say m times, so that the string generated is $c^{n+m}S_3$ $(m \geq 1)$. If at this point, the alphabetic flat splicing rule is applied on the pair $(a^nb^nS_2S_1, c^{n+m}S_3)$, the string generated is $a^nb^nS_2c^{n+m}S_3S_1$. At the same time, prior to applying the flat splicing rules, if the erasing rules are applied, strings of the form c^p, a^nb^n will be generated. Thus the language generated is L_3.

Theorem 5. $\mathcal{L}(RP_1(LIN/AFSR)) \subset \mathcal{L}(RP_2(LIN/AFSR))$.

Proof. The inclusion holds by definition. In order to prove proper inclusion, consider the language $L_4 = \{a^nb^nc^n \mid n \geq 1\}$, which is context-sensitive and not context-free. The following $RP_2(LIN/AFSR)$ generates the language L_4:

$$(\{S_1, S_2, a, b, c\}, \{a, b, c\}, [_1\, [_2\,]_2\,]_1, \{S_1, S_2\}, \emptyset, R_1, R_2, 2),$$

where R_1 consists of the linear rules $S_1 \to aS_1c, S_2 \to bS_2$ and an alphabetic flat splicing rule $(a|b - S_2|S_1), in$, while R_2 consists of the rules $S_1 \to \lambda, S_2 \to \lambda$. The computation takes place as follows: In region 1, S_1, S_2 generate the strings $a^nS_1c^n$ and b^nS_2 for the same n. At this point if the alphabetic flat splicing in region 1 is applied on the pair $(a^nS_1c^n, b^nS_2)$, then the string generated is $a^nb^nS_2S_1c^n$ which is sent to region 2. Here application of the rules $S_1 \to \lambda, S_2 \to \lambda$ erases the nonterminals, thus yielding the string $a^nb^nc^n$.

On the other hand, one membrane is not enough to generate L_4 since $AFSS$ rules alone cannot generate the non-CF language due to Theorem 1, while linear rules alone are also not enough. So if both types of rules are included in the only membrane to generate L_4, then the nonterminals of the linear rules are to be erased or replaced by terminal strings finally. But this will lead to strings not in L_4.

4 $RP_m(REG/AFSR)$ with Extended Initial Objects

In rewriting P systems [12] generating languages, the initial objects in the membranes can be strings of finite languages. This means that strings of length more than one can be the initial objects unlike the objects in $RP_m(LIN/AFSR)$ or $RP_m(REG/AFSR)$ considered here so far, where the initial objects are symbols from the alphabet.

Instead of recalling the formal details of the definition of a rewriting P system [12], we illustrate with an example of a rewriting P system having regular rules in its regions and computing a context-free language. Note that the system works in maximal parallelism manner, but at the level of a membrane the rewriting by a rule is sequential and the rewritten string moves to the membrane indicated by the target.

Example 3. Consider the rewriting P system of degree 2 with regular rewriting rules, given by $(V, T, \mu, L_1, L_2, R_1, R_2, 2)$, where the total alphabet $V = \{A, B, a, b\}$, the terminal alphabet $T = \{a, b\}$, the membrane structure $\mu = [_1\,[_2\,]_2\,]_1$. The sets of initial strings in the membranes are given by $L_1 = \emptyset, L_2 = \{AB\}$. The sets R_1, R_2 with regular rules with associated targets (*here*, *in* or *out*) are given by

$$R_1 = \{B \to bB(in)\}, R_2 = \{A \to aA(out), A \to a(here), B \to b(here)\}.$$

Computation in this P system will take place as follows: Only the membrane 2 has an initial string AB (of length 2) which will evolve by the sequential application of a rule in region 2. If the rule $A \to aA(out)$ is applied to AB, the generated string aAB is sent to region 1, where the application of the rule $B \to bB(in)$ generates $aAbB$ which is sent back to region 2. The process can repeat or terminate by the application of the rules $A \to a(here), B \to b(here)$ yielding words of the form $a^n b^n, n \geq 1$. The language generated is a non-regular context-free language $\{a^n b^n \mid n \geq 1\}$. An application of the rule $B \to b$ followed by the application of the rule $A \to aA(out)$ will send the string to region 1 where it will get stuck. Note that if in region 2, the initial string is allowed to have length not more than 1, this language cannot be generated with regular rules and two membranes.

We now allow in our definition of $RP_m(REG/AFSR)$ initial objects to have length more than 1 but continue to refer to such a system by $RP_m(REG/AFSR)$ itself. A priority relation $>$ on the rules is a well-known notion used in P systems. If a region has rules r_1, r_2 with $r_1 > r_2$, then if both the rules could be applied to strings in that region, only r_1 is applied. We denote such a system with a priority relation on the regular rules in the regions and initial strings in the regions having length one or more by $RP_m(REG/AFSR, pri)$. The corresponding family of languages is denoted by $\mathcal{L}(RP_m(REG/AFSR, pri))$.

Theorem 6. $CFL \subset \mathcal{L}(RP_2(REG/AFSR, pri))$.

Proof. Let L (without the empty word) be a context-free language and $G = (N, \Sigma, P, S)$ be a context-free grammar in Chomsky normal form with P consisting of n rules of the form $A \to BC$ or $A \to a$ generating L. The members of N are the nonterminals and those of T are terminals of G with $S \in N$ as the start symbol. We construct a $RP_2(REG/AFSS, pri)$ Π_3 to generate L. $\Pi_3 = (V, T, [_1\,[_2\,]_2\,]_2, L_1, L_2, (R_1, >), R_2, 2)$, where $V = N \cup \Sigma \cup \{F_i, E, E' \mid F_i, E, E' \notin N, 1 \leq i \leq n\}, T = \Sigma$, such that with each rule r_i, $1 \leq i \leq n$, a distinct symbol F_i is associated. We set

$$L_1 = \{SE, F_j BC, F_k a \mid r_j : A \to BC \in P, r_k : A \to a \in P,$$

$$F_j, F_k \text{ are associated with rules } r_j, r_k\},$$

$$L_2 = \emptyset,$$

$$R_1 = \{(A|F_j - C|\alpha), (A|F_k - a|\alpha) \mid r_j : A \to BC, r_k : A \to a,$$

$$A, B, C \in N, a \in \Sigma, \alpha \in V - \{F_i | 1 \leq i \leq n\}\} \cup \{(E|E' - \lambda|\lambda)(in)\},$$
$$R_2 = \{X \to \lambda \mid X \in N \cup \{F_i, E, E' \mid 1 \leq i \leq n\}\}.$$

As usual, we omit mentioning the target "*here*". The priority relation $>$ on the rules of R_1 is defined as follows: $r > (E|E' - \lambda|\lambda)$ for each rule r in $R_1 - \{(E|E' - \lambda|\lambda)\}$. The construction is very close to a similar result in [7].

It can be seen that a derivation in the context-free grammar G starting from S and leading to a word in L can be simulated by Π_3 as follows: The computation starts with the initial string SE in membrane 1 and the result of rewriting by a rule of the form $A \to BC$ is captured by inserting F_jBC (F_j being the associated symbol) to the immediate right of the symbol A in the currently generated word. Likewise, corresponding to the application of the rule $A \to a$, the word F_ka (F_k being the associated symbol) is inserted again to the immediate right of A. Due to the priority, only when no other rule could be applied, the rule $(E|E' - \lambda|\lambda)$ could be applied which will send the current word to the membrane 2. Here all the nonterminals (symbols not in Σ) are erased and the resulting terminal word is collected in the language, thus generating the context-free language L.

Finally, we note that in Theorem 4, a $RP_2(REG/AFSR)$ is shown to generate a context-sensitive language which is not context-free. This shows that the inclusion is proper.

5 Conclusions and Discussions

In this work, we have considered rewriting P systems with simple rewriting rules and flat splicing rules with a sequential application of rules to an object in a membrane, and the language generative power of such P systems has been investigated. It may be of interest to examine the power of parallel application of rules as in L systems with similar simple rewriting rules.

The language generative power of rewriting P systems with regular rules and flat splicing rules in comparison with flat splicing systems has been investigated in section 3. It is of interest to investigate whether rewriting P systems with regular rules and flat splicing rules can generate any recursively enumerable language.

The priority relation has been considered in rewriting P systems in section 4, which ensures that the context-free grammar is simulated correctly. It remains open whether the result in Theorem 6 still holds if the priority relation is removed.

Acknowledgements

The work of L. Pan and B. Song was supported by National Natural Science Foundation of China (61033003, 61320106005 and 61472154), Ph.D. Programs Foundation of Ministry of Education of China (20120142130008), the Innovation Scientists and Technicians Troop Construction Projects of Henan Province (154200510012). K.G. Subramanian acknowledges support from the Emeritus Fellowship of University Grants Commission, India for the period 2016-17.

References

1. Berstel, J., Boasson, L., Fagnot, I.: Splicing Systems and the Chomsky Hierarchy. Theor. Comput. Sci. 436, 2–22 (2012)
2. Besozzi, D., Ferretti, C., Mauri, G., Zandron, C.: Parallel Rewriting P Systems with Deadlock. In: Hagiya, M., Ohuchi, A. (eds.) LNCS, vol. 2568, pp. 302–314. Springer, Heidelberg (2003)
3. Ciobanu, G., Pérez-Jiménez, M.J., Păun, Gh.: Applications of Membrane Computing. Springer-Verlag, Berlin (2006)
4. Freund, R., Martín-Vide, C., Păun, Gh.: From Regulated Rewriting to Computing with Membranes: Collapsing Hierarchies. Theor. Comput. Sci. 312(2-3), 143–188 (2004)
5. Head, T.: Formal Language Theory and DNA: An Analysis of the Generative Capacity of Specific Recombinant Behaviours. Bull. Math. Biol. 49, 735–759 (1987)
6. Head, T.: Circular Suggestions for DNA Computing. In: Pattern Formation in Biology, Vision and Dynamics (Ed. Carbone, A. et al.), World Scientific, 325–335 (2000)
7. Krassovitskiy, A.: On the Power of Small Size Insertion P Systems. Int. J. Comput. Commun. VI, 266–277 (2001)
8. Krishna, S.N., Lakshmanan, K., Rama, R.: Hybrid P Systems. Rom. J. Inf. Sci. Tech. 4, 11–123 (2001)
9. Krishna, S.N., Lakshmanan, K., Rama, R.: On the Power of P Systems with Contextual Rules. Fund. Informa. 49, 167–178 (2002)
10. Păun, A.: P Systems with Global Rules, Theor. Comput. Syst. 35(5), 471–481 (2002)
11. Păun, A.: On P Systems with Partial Parallel Rewriting. Rom. J. Inf. Sci. Tech. 4, 203–210 (2001)
12. Păun, Gh.: Computing with Membranes. J. Comput. Syst. Sci. 61, 108–143 (2000)
13. Păun, Gh.: Computing with Membranes: An Introduction. Springer-Verlag, Berlin (2002)
14. Păun, Gh., Rozenberg, G., Salomaa, A.: DNA Computing–New Computing Paradigms (Texts in Theoretical Computer Science. An EATCS Series). Springer-Verlag, New York (1998)
15. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, New York (2010)
16. Rozenberg, G., Salomaa, A. (Eds.): Handbook of Formal Languages. Vols. 1-3, Springer-Verlag, Berlin (1997)

The Improved Apriori Algorithm Based on the Tissue-like P System

Yuzhen Zhao, Xiyu Liu*, and Wenxing Sun

School of Management Science and Engineering, Shandong Normal University, Jinan, 250014, China
zhaoyuzhen_happy@126.com
sdxyliu@163.com
373253360@qq.com

Abstract. The Apriori algorithm plays an important role in data mining and data analysis, which can help people to discover useful knowledge from large amounts of data. In this paper, an improved Apriori algorithm based on the tissue-like P system is constructed, which provides new ideas and methods for mining the frequent itemsets. Since the P system has an advantage of great parallelism, it can reduce the computational time complexity and is suitable for the cluster problem. The proposed improved Apriori algorithm is a new attempt in the applications of membrane system and it provides a novel perspective of cluster analysis.

Keywords: Frequent itemsets; the Apriori algorithm; Membrane Computing; P System; Membrane System

1 Introduction

The implicit pattern or knowledge can be extracted from huge amounts of data by data mining techniques. Frequent itemsets mining, as one of the important areas of data mining, is an effective method to pack data and discover useful information from these data. It is a hot area of research in machine learning, statistics, biology and many other fields. Through frequent itemsets mining, the interesting association between the items in the relational data base can be discovered which can help many businessmen make decisions such as classification design, cross-selling and customer buying habits analysis [1]. With the emergence of big data, which shows several characteristics, such as volume, variety, velocity, and value, the data mining is paid a greater attention [2]. People need more efficient algorithms to deal with the big data. Membrane computing, as a new biological computing model, has maximal parallelism and can significantly improve the efficiency of computation.

* Project supported by the Natural Science Foundation of China (No. 61170038, 61472231, 61402187, 61502535, 61572522 and 61572523).

Apriori algorithm, as one of the most influential frequent itemsets mining algorithms, mines the frequent itemsets through two stages: the generation of candidate sets and the downward close detection of plot. It is suitable for frequent itemsets mining over transactional databases. The frequent itemsets the algorithm discovers can be used to determine the association rules which can highlight the general trends in database [3]. Inokuchi et al. [4] proposed an Apriori-based algorithm to mine frequently appearing induced subgraphs in a given graph data set in 2000. Baker and Prasanna [5] reduced time required for processing through the use of a new extension to the systolic array architecture in 2005. Perego et al. [6] used an innovative method for storing candidate itemsets and counting their support in 2001. Lazcorreta et. al [7] proposed a two-step modified Apriori algorithm in 2008. Singh et al. [8] used transaction reduction to improve the efficiency of the algorithm in 2013. Bhandari et al. [9] used a frequent pattern tree to improve the algorithm in 2014. Cheng et al. [10] proposed a differential privacy (DP)-Apriori algorithm which can simultaneously provide a high level of data utility and a high level of data privacy in 2015.

P system, the computing model of the new biological calculation method membrane computing, is abstracted based on the structure and function of the cell. There are three mainly investigated P systems, cell-like P systems [11], tissue P systems [12], and neural-like P systems (also known as spiking neural P systems) [13] (and their variants, see e.g. [14, 15]). P systems are known as powerful computing models, are able do what Turing machine can do [16-19]. The parallel evolution mechanism of variants of P systems, such as numerical P systems [20], spatial P systems [21], spiking neural P systems with anti-spikes [22], have been found to perform well in doing computation, even solving computational hard problems [23].

Since the volume characteristic of big data, it is very difficult for the existing computing model to obtain the calculation results quickly. P system is also suitable for data processing [24, 25]. Membrane computing has been widely applied to many fields such as biology [26, 27], linguistics [28, 29], network communication [30, 31], and graphics [32, 33]. But the application in data mining, especially in frequent itemsets mining, is less. Using the membrane computing to the clustering can reduce the time complexity of clustering, so it has a certain theoretical and practical significance.

The Apriori algorithm and the membrane computing are combined in this study to mine the frequent itemsets over transactional databases. An improved Apriori algorithm is proposed first. This algorithm searches all records synchronously and generates the support number. The membrane structure is initialized next and the rules of algorithm are determined using the improved Apriori algorithm. The computational process is analysed next.

2 The Improved Apriori Algorithm

Apriori algorithm is an frequent itemsets mining algorithm designed for transactional databases specially. The idea is simple which is based on the recursive

statistics to generate frequent itemsets and is easy to be implemented.

In a transaction database D, each transaction is a collection of items. A collection of items is called itemset. An itemset which contains k items is called k itemset. The frequency of an itemset is the number of transactions which consists of this itemset. The frequency of an itemset is also called support count or count. If the support of itemset I meets the corresponding minimum support count threshold, itemset I is called a frequent itemset.

Firstly it identifies the itemset with all frequent individual items of the database. Secondly it repeats the steps below: 1. Derive the candidates with k items by connecting each two $k-1$ items which has only one different item and deleting the items which have subitems of length $k-1$ that do not belong to the $k-1$ itemset. 2. Scan the database to compute the frequencies of each candidate. 3. Delete the infrequent items to gain the frequent itemsets of size k. 4. If no items in k itemset, the circulation stops. Finally all itemsets are generated into one collection. The item sets gained by the Apriori algorithm can be used to find association rules. Association rules are very useful in market basket analysis and other domains [34, 35].

Although this algorithm is very classical and useful, it has a very big weakness: its time complexity is high ($O(D^2 n_h ht)$) [42]. Because the original Apriori algorithm has the deficiency of high time complexity, this paper puts forward an improved Apriori algorithm to reduce it. When we scan the database to compute the frequencies of each candidate, the improved Apriori algorithm searches all records synchronously and the support number will be generated at the same time. (The parallelism here is realized by the parallelism of the P system. The detail of the P system will be introduced below.) One item is added to the corresponding itemset only if its support meets the threshold conditions. That is to say, a parallel Apriori algorithm is proposed based on the tissue-like P system. This improvement will considerably reduce the time complexity of the algorithm.

3 Rules

3.1 Tissue-like P system

Since the concept of membrane computing was proposed, investigators have proposed different types of P systems simulating the different biochemical response mechanisms of the cells or tissue. These P systems can be roughly divided into three categories: cell-like P systems, tissue-like P systems and neural-like P systems. They are abstract models calculated from the cells, tissues, and nervous system [36, 37]. Here we assume that the readers have the basic knowledge about the membrane computing, so we don't introduce in detail. Readers can refer [38] for more.

Tissue-like P system is an important extension model of cell-like P system. Tissue-like P system has many cells which are freely placed in the same environment. Both the cells and the environment can contain objects. Cells and cells or cells and the environment can communicate by rules. If the communication

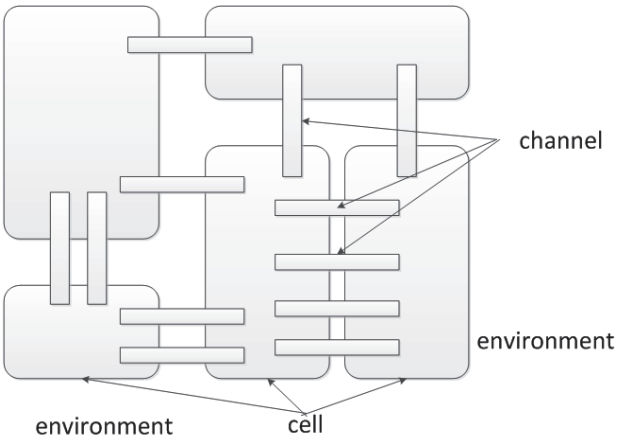


Fig. 1. The basic membrane structure

channels between cells are given in advance by the rules (fixed), such a P system is called basic tissue-like P system. The basic tissue-like P system used below is introduced here [39,40].

The basic membrane structure is shown in Fig. 1. In general, a basic tissue-like P system of degree m is a construct:

$$\prod = (\mathrm{O},\sigma_1,\sigma_2,...,\sigma_m,\mathrm{syn},\mathrm{i}_{out}) \tag{1}$$

where:

(1) O is the alphabet which includes all objects of the system;

(2) $syn \subseteq \{1,2,...,m\} * \{1,2,...,m\}$ shows all channels between cells;

(3) $i_{out} \in \{1,2,...,m\}$ is the output cell which shows the computational result of the system;

(4) $\sigma_1, \sigma_2, ..., \sigma_m$ represent the m cells respectively. Each cell is in the form:

$$\sigma_i = (w_{i,o}, P_i), 1 \leq i \leq m \tag{2}$$

where:

(a) $w_{i,o}$ comprises the initial objects in cell i, we use the object λ to show there is no object in cell i;

(b) P_i is the set of rules in cell i with the form of $(u \to v)_r$, u is a string composed of objects in O and v is a string in the form of $v = v'$ or $v = v'\delta$. v' is a string over $\{a_{here}, a_{out}, a_{in_j} | a \in O, 1 \leq j \leq m\}$. δ is a symbol not in O. It means after executing the rule this membrane will be dissolved. r stands for the promoters or the inhibitors and it is in the form of $r{=}r'$ or $r{=}\ \neg r'$. A rule can execute only when the promoters r' appear and a rule can stop only when the inhibitors appear. The radius of this rule $u \to v$ is the length of u. We use the object λ to show there is no rule in cell i.

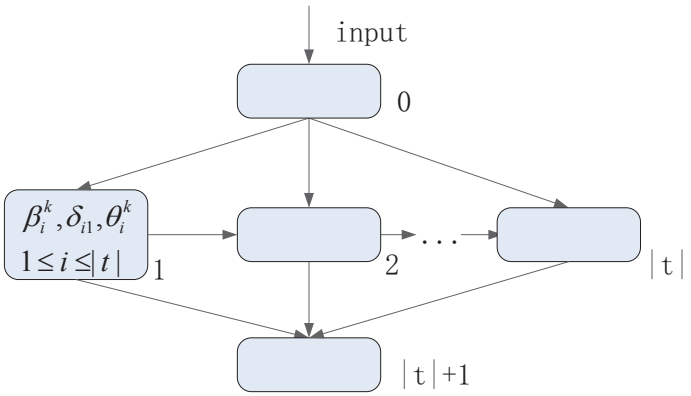


Fig. 2. The tissue-like P system for the Apriori algorithm

Rules are executed in the maximum parallel way and in the uncertain way in each membrane. That is to say, rules should be used in parallel to the maximum degree possible. If more than one rule can be used possibly but the objects in membrane can only support parts of rules to be used, the rules which are used are chosen uncertainly. This is very helpful to solve the computationally hard problems. The P system will halt after some steps if no more rules can be executed or an end mark appears and these objects in output membrane is the final result. The P system will not halt if rules are always executed, then this calculation is invalid, and there is no result being exhibited [41].

3.2 Rules of the improved Apriori algorithm based on the Tissue-like P system

The P system for the improved Apriori algorithm is proposed here. Its structure is depicted in Fig. 2. It uses the subscript i, j of the entries a_{ij} to represent the j-th field of the i-th record in the database and k to represent the threshold of the supports. The database D has $|D|$ records and $|t|$ fields. So the P system for the algorithm is defined as follows:

$$\Pi = (O, \sigma_1, \sigma_2, \ldots, \sigma_{|t|+1}, \mathrm{syn}, \mathrm{i}_{out}) \tag{3}$$

where:

(1) $O = \{a_{jp}, \beta_{i_1}, \beta_{i_1 i_2}, ..., \beta_{i_1 i_2 ... i_{|t|}}, \delta_{i_1}, \delta_{i_1 i_2}, ..., \delta_{i_1 i_2 ... i_{|t|}}, \theta_{i_1}, \theta_{i_1 i_2}, ..., \theta_{i_1 i_2 ... i_{|t|}}, \varphi, \alpha, 1 \le j \le |D|, 1 \le p, i \le |t|\}$
(2) $syn = \{\{0,1\}\{0,2\}, ..., \{0,|t|\}, \{1,|t|+1\}, \{2,|t|+1\}, ..., \{|t|,|t|+1\} \{1,2\}\{2,3\}...\{|t|-1,|t|\}\}$
(3) $i_{out} = |t| + 1$
(4) $\sigma_0 = (w_{0,0}, P_0)$, where:
$w_{0,0} = \lambda$

$P_0 : r_1 = \{a_{ij} \to (a_{ij}, go) \,|1 \le i \le |D|, 1 \le j \le |t|\}$

$\sigma_1 = (w_{1,0}, P_1)$, where:

$w_{1,0} = \{\beta_i^k \; 1 \le i \le |t|, \delta_{11}, \delta_{21}, ..., \delta_{|t|1}, \theta_1^k, \theta_2^k, ..., \theta_{|t|}^k\}$

P_1 :

$r_1 = \{\delta_{1i}a_{i1}\beta_1^j \to \delta_{1(i+1)}a_{i1}\beta_1^{j-1} \cup (\delta_{1i}a_{i1}\theta_1^j)_{\neg\beta_1} \to \delta_{1(i+1)}a_{i1}\theta_1^{(j+1)}$
$\cup\, (\delta_{1i})_{\neg a_{i1}} \to \delta_{1(i+1)}\}$
$\cup\, \{\delta_{2i}a_{i2}\beta_2^j \to \delta_{2(i+1)}a_{i2}\beta_2^{j-1} \cup (\delta_{2i}a_{i2}\theta_2^j)_{\neg\beta_2} \to \delta_{2(i+1)}a_{i2}\theta_2^{(j+1)}$
$\cup\, (\delta_{2i})_{\neg a_{i2}} \to \delta_{2(i+1)}\}$
...
$\cup\{\delta_{|t|i}a_{i|t|}\beta_{|t|}^j \to \delta_{|t|(i+1)}a_{i|t|}\beta_{|t|}^{j-1} \cup (\delta_{|t|i}a_{i|t|}\theta_{|t|}^j)_{\neg\beta_{|t|}} \to \delta_{|t|(i+1)}a_{i|t|}\theta_{|t|}^{(j+1)}$
$\cup\, (\delta_{|t|i})_{\neg a_{i|t|}} \to \delta_{|t|(i+1)}\}(1 \le i, j \le |D|)$

$r_2 = \{\delta_{i(|D|+1)}(\theta_i^j)_{\neg\beta_i} \to \theta_i^j\alpha_i(\alpha_i, go)\} \cup \{\delta_{i(|D|+1)}\beta_i^j\theta_i^k \to \lambda\}$
$(1 \le i \le |t|, 1 \le j \le |D|)$

$r_3 = \{()_{\neg\alpha_i} \to \#|1 \le i \le |t|\}$

$\sigma_2 = (w_{2,0}, P_2)$, where:

$w_{2,0} = \lambda$

P_2 :

$r_1 = \{(\alpha_i\alpha_j)_{\neg\delta_{1ij}} \to \alpha_i\alpha_j\delta_{1ij}\beta_{ij}^k\theta_{ij}^k \,|1 \le i < j \le |t|, 1 \le k \le |D|\}$

$r_2 = \{\alpha_i \to \lambda|1 \le i \le |t|\}$

$r_3 = \{\delta_{tij}a_{ti}a_{tj}\beta_{ij}^p \to \delta_{(t+1)ij}a_{ti}a_{tj}\beta_{ij}^{p-1}\}$
$\cup\, \{(\delta_{tij}a_{ti}a_{tj}\theta_{ij}^p)_{\neg\beta_{ij}} \to \delta_{(t+1)ij}a_{ti}a_{tj}\theta_{ij}^{(p+1)}\}$
$\cup\, \{(\delta_{tij})_{\neg a_{ti}a_{tj}} \to \delta_{(t+1)ij}\}(1 \le t, p \le |D|, 1 \le i, j \le |t|)$

$r_4 = \{\delta_{(n+1)ij}(\theta_{ij}^t)_{\neg\beta_{ij}} \to \theta_{ij}^t\alpha_{ij}(\alpha_{ij}, go) \cup \delta_{(n+1)ij}\beta_{ij}^t\theta_{ij}^k \to \lambda\}$
$(1 \le i, j \le |t|, 1 \le t \le |D|)$

$r_5 = \{()_{\neg\alpha_{ij}} \to \#|1 \le i, j \le |t|\}$

...

$\sigma_{|t|-1} = (w_{|t|-1,0}, P_{|t|-1})$, where:

$w_{|t|-1,0} = \lambda$

$P_{|t|-1}$:

$r_1 = \{(\alpha_{i_1i_2...i_{|t|-3}j_1}\alpha_{i_1i_2...i_{|t|-3}j_2})_{\neg\delta_{1i_1i_2...i_{|t|-3}j_1j_2}} \to$
$\alpha_{i_1i_2...i_{|t|-3}j_1}\alpha_{i_1i_2...i_{|t|-3}j_2}\delta_{1i_1i_2...i_{|t|-3}j_1j_2}\beta_{i_1i_2...i_{|t|-3}j_1j_2}^k\theta_{ii_1i_2...i_{|t|-3}j_1j_2}^k$
$(1 \le i < j \le |t|, 1 \le k \le |D|)$

$r_2 = \{\alpha_{i_1i_2...i_{|t|-2}} \to \lambda|1 \le i \le |t|\}$

$r_3 = \{\delta_{ti_1i_2...i_{|t|-3}j_1j_2}a_{i_1}a_{i_2}...a_{i_{|t|-3}}a_{j_1}a_{j_2}\beta_{i_1i_2...i_{|t|-3}j_1j_2}^p \to$
$\delta_{(t+1)i_1i_2...i_{|t|-3}j_1j_2}a_{i_1}a_{i_2}...a_{i_{|t|-3}}a_{j_1}a_{j_2}\beta_{i_1i_2...i_{|t|-3}j_1j_2}^{p-1}\}$
$\cup\, \{(\delta_{ti_1i_2...i_{|t|-3}j_1j_2}a_{i_1}a_{i_2}...a_{i_{|t|-3}}a_{j_1}a_{j_2}\theta_{i_1i_2...i_{|t|-3}j_1j_2}^p)_{\neg\beta_{ij}} \to$
$\delta_{(t+1)i_1i_2...i_{|t|-3}j_1j_2}a_{i_1}a_{i_2}...a_{i_{|t|-3}}a_{j_1}a_{j_2}\theta_{i_1i_2...i_{|t|-3}j_1j_2}^{p+1}\}$
$\cup\, \{(\delta_{ti_1i_2...i_{|t|-3}j_1j_2})_{\neg a_{i_1}a_{i_2}...a_{i_{|t|-3}}a_{j_1}a_{j_2}} \to \delta_{(t+1)i_1i_2...i_{|t|-3}j_1j_2}$
$(1 \le t, p \le |D|, 1 \le i, j \le |t|)$

$r_4 = \{\delta_{(|D|+1)i_1i_2...i_{|t|-3}j_1j_2}(\theta_{i_1i_2...i_{|t|-3}j_1j_2}^t)_{\neg\beta_{i_1i_2...i_{|t|-3}j_1j_2}} \to$
$\theta_{i_1i_2...i_{|t|-3}j_1j_2}^t\alpha_{i_1i_2...i_{|t|-3}j_1j_2}(\alpha_{i_1i_2...i_{|t|-3}j_1j_2}, go)\}$
$\cup\, \{\delta_{(|D|+1)i_1i_2...i_{|t|-3}j_1j_2}\beta_{i_1i_2...i_{|t|-3}j_1j_2}^t\theta_{i_1i_2...i_{|t|-3}j_1j_2}^k \to \lambda\}$

$$(1 \le i, j \le |t|, 1 \le t \le |D|)$$
$$r_5 = \{()_{\neg\alpha_{i_1 i_2 \dots i_{|t|-1}}} \to \#|1 \le i \le |t|\}$$
$\sigma_{|t|} = (w_{|t|,0}, P_{|t|})$, where:
$$w_{|t|,0} = \lambda$$
$$P_{|t|}:$$
$$r_1 = \{(\alpha_{i_1 i_2 \dots i_{|t|-2} j_1} \alpha_{i_1 i_2 \dots i_{|t|-2} j_2})_{\neg\delta_{1 i_1 i_2 \dots i_{|t|-2} j_1 j_2}} \to$$
$$\alpha_{i_1 i_2 \dots i_{|t|-2} j_1} \alpha_{i_1 i_2 \dots i_{|t|-2} j_2} \delta_{1 i_1 i_2 \dots i_{|t|-2} j_1 j_2} \beta^k_{i_1 i_2 \dots i_{|t|-2} j_1 j_2} \theta^k_{i i_1 i_2 \dots i_{|t|-2} j_1 j_2}\}$$
$$(1 \le i < j \le |t|, 1 \le k \le |D|)$$
$$r_2 = \{\alpha_{i_1 i_2 \dots i_{|t|-1}} \to \lambda | 1 \le i \le |t|\}$$
$$r_3 = \{\delta_{t i_1 i_2 \dots i_{|t|-2} j_1 j_2} a_{i_1} a_{i_2} \dots a_{i_{|t|-2}} a_{j_1} a_{j_2} \beta^p_{i_1 i_2 \dots i i_{|t|-2} j_1 j_2} \to$$
$$\delta_{(t+1) i_1 i_2 \dots i_{|t|-2} j_1 j_2} a_{i_1} a_{i_2} \dots a_{i_{|t|-2}} a_{j_1} a_{j_2} \beta^{p-1}_{i_1 i_2 \dots i_{|t|-2} j_1 j_2}\}$$
$$\cup \{(\delta_{t i_1 i_2 \dots i_{|t|-2} j_1 j_2} a_{i_1} a_{i_2} \dots a_{i_{|t|-2}} a_{j_1} a_{j_2} \theta^p_{i_1 i_2 \dots i_{|t|-2} j_1 j_2})_{\neg\beta_{ij}} \to$$
$$\delta_{(t+1) i_1 i_2 \dots i_{|t|-2} j_1 j_2} a_{i_1} a_{i_2} \dots a_{i_{|t|-2}} a_{j_1} a_{j_2} \theta^{p+1}_{i_1 i_2 \dots i_{|t|-2} j_1 j_2}\}$$
$$\cup \{(\delta_{t i_1 i_2 \dots i_{|t|-2} j_1 j_2})_{\neg a_{i_1} a_{i_2} \dots a_{i_{|t|-2}} a_{j_1} a_{j_2}} \to \delta_{(t+1) i_1 i_2 \dots i_{|t|-2} j_1 j_2}\}$$
$$(1 \le t, p \le |D|, 1 \le i, j \le |t|)$$
$$r_4 = \{\delta_{(|D|+1) i_1 i_2 \dots i_{|t|-2} j_1 j_2} (\theta^t_{i_1 i_2 \dots i_{|t|-2} j_1 j_2})_{\neg\beta_{i_1 i_2 \dots i_{|t|-2} j_1 j_2}} \to$$
$$\theta^t_{i_1 i_2 \dots i_{|t|-2} j_1 j_2} \alpha_{i_1 i_2 \dots i_{|t|-2} j_1 j_2} (\alpha_{i_1 i_2 \dots i_{|t|-2} j_1 j_2}, go)\}$$
$$\cup \{\delta_{(|D|+1) i_1 i_2 \dots i_{|t|-2} j_1 j_2} \beta^t_{i_1 i_2 \dots i_{|t|-2} j_1 j_2} \theta^k_{i_1 i_2 \dots i_{|t|-2} j_1 j_2} \to \lambda\}$$
$$(1 \le i, j \le |t|, 1 \le t \le |D|)$$
$$r_5 = \{()_{\neg\alpha_{i_1 i_2 \dots i_{|t|}}} \to \#|1 \le i \le |t|\}$$
$\sigma_{|t|+1} = (w_{|t|+1,0}, P_{|t|+1})$, where:
$$w_{|t|+1,0} = \lambda$$
$$P_{|t|+1} = \lambda$$

The computational process of the proposed algorithm is described in the next section.

4 Computational Process

As defined above, the database D has $|D|$ records and $|t|$ fields. And the frequent itemsets with k support or higher are expected to be found. Because this is a transactional database, the object a_{ij} is used to represent the j-th field of the i-th record existing in the database. So a_{jp} $(1 \le j \le |D|, 1 \le p \le |t|)$ is put to membrane 0. And this membrane puts all the input objects into membrane 1 to membrane $|t|$, so each of these membranes has the original data of the database.

When membrane 1 gets the objects a_{ij}, the rules in it are activated. δ_{ij} are used to control the circulations. The first subscript i of δ_{ij} represents the i-th field of the database and the second one represents the j-th record of the database. Membrane 1 has $\delta_{11}, \delta_{21}, ..., \delta_{|t|1}$ in the beginning. So the $|t|$ sub-rules of r_1 run at the same time to count the supports of all $|t|$ fields starting from record 1.

The first sub-rule of r_1 is taken as the example. The first subscript 1 of δ_{1j} represents this sub-rule counts the support of the first field of the database.

The second subscript of δ_{1j} is 1. So the first record is checked. If a_{11} exists, the number of object β_1 decreases 1 and the second subscript of δ_{11} increases 1 to check the object a_{21} and so on. If a_{i1} does not exist, the number of object β_1 does not change and second subscript j of object δ_{1j} increases 1 to check $a_{(i+1)1}$. The number of object β_1 is k. So when all the objects β_1 are disappeared, the number of a_{i1} is k. If the number of a_{i1} increases still this time, the number of object θ_1 increases. There are k θ_1 in the beginning. So the number of θ_1 can show the support of a_{i1}. Other $|t|-1$ field are similar to the first field. So these are not repeated here.

Rule r_2 is activated when the second subscript of δ_{ij} reaches $|D|+1$. If there is no object β_i at this time, the support count of the i-th field meets the threshold. Item i is a frequent itemset of size 1. The object α_i is generated to show that and α_i goes to membrane 2 for the next step of the computation and membrane $|t|+1$ for storing the result of computation (membrane $|t|+1$ is the membrane which shows the result of the computation). By these rules, the set of itemsets with frequent items of size 1 is generated. If there is no item that meets the requirement, the computation stops.

Then membrane 2 gets the object α_i. Firstly, the corresponding objects $\delta_{1ij}, \beta_{ij}^k, \theta_{ij}^k$ are generated if α_i, α_j exist. The three subscripts of δ_{tij} represent that the i-th and the j-th fields of the t-th record in the database. The meaning of β_{ij} and θ_{ij} are similar with β_i and θ_i. Then the computation in membrane 2 is similar to that in membrane 1. Finally, the set of itemsets with frequent items of size 2 will be generated. And so on until the set of itemsets with frequent items of size $|t|$ is generated by membrane $|t|$.

There are some objects α in the output membrane $|t|+1$ which subscript shows the frequent items of the database.

5 Experiments and Analysis

In order to illustrate how the proposed algorithm runs, an illustrative example is considered. Table 1 shows the transaction data of one branch office of All-Electronics [1]. There are 9 transactions in this table and each transaction has 5 fields. Suppose the support count threshold is 2.

When computation stops, objects α_1,α_2,α_3,α_4,α_5,α_{12},α_{13}, α_{15},α_{23},α_{24},α_{25}, $\alpha_{123}, \alpha_{125}$ are in cell 6, which means $\{I1\}\{I2\}\{I3\}\{I4\}\{I5\}\{I1, I2\}\{I1, I3\}$ $\{I1, I5\}\{I2, I3\}\{I2, I4\}\{I2, I5\}\{I1, I2, I3\}\{I1, I2, I5\}$ are all frequent itemsets in this database.

6 Conclusions

With the advent of the era of big data, the traditional way of data processing is more and more difficult to meet peoples requirement to efficiency. Profit from the great parallelism, P system can decrease the time complexity of computing and improve the computational efficiency. Recent years, as a new biological computing method, the theory of membrane computing has been adequately

Table 1. The transaction data of one branch office of AllElectronics

TID	items
$T100$	I1,I2,I5
$T200$	I2,I4
$T300$	I2,I3
$T400$	I1,I2,I4
$T500$	I1,I3
$T600$	I2,I3
$T700$	I1,I3
$T800$	I1,I2,I3,I5
$T900$	I1,I2,I3

studied. Due to its great parallelism, it has been applied into many fields like combinatorial problem, finite state problems and graph theory but not enough. An improved frequent itemsets mining algorithm based on the Tissue-like P system is constructed in this paper. This paper applies membrane computing into the typical frequent itemsets mining algorithm Apriori algorithm enlarges the research field of P system. This paper is focused in a first step on denotation of the algorithm using a tissue P system while practical studies will follow. Additionally, there are many data mining methods and membrane computing can be applied to a variety of other data mining methods.

References

1. Han J, Kamber M, Pei J. Data mining: concepts and techniques. Elsevier, 2011.
2. Che D, Safran M, Peng Z. From big data to big data mining: challenges, issues, and opportunities. Database Systems for Advanced Applications. Springer Berlin Heidelberg, 2013: 1-15.
3. Agrawal R, Srikant R. Fast algorithms for mining association rules. Proc. 20th int. conf. very large data bases, VLDB. 1994, 1215: 487-499.
4. Inokuchi A, Washio T, Motoda H. An apriori-based algorithm for mining frequent substructures from graph data. Principles of Data Mining and Knowledge Discovery. Springer Berlin Heidelberg, 2000: 13-23.
5. Baker Z K, Prasanna V K. Efficient hardware data mining with the Apriori algorithm on FPGAs. Field-Programmable Custom Computing Machines, 2005. FCCM 2005. 13th Annual IEEE Symposium on. IEEE, 2005: 3-12.
6. Perego R, Orlando S, Palmerini P. Enhancing the apriori algorithm for frequent set counting. Data Warehousing and Knowledge Discovery. Springer Berlin Heidelberg, 2001: 71-82.
7. Lazcorreta E, Botella F, Fernández-Caballero A. Towards personalized recommendation by two-step modified Apriori data mining algorithm. Expert Systems with Applications, 2008, 35(3): 1422-1429.
8. Singh J, Ram H, Sodhi D J S. Improving efficiency of apriori algorithm using transaction. International Journal of Scientific and Research Publications, 2013, 3(1): 1-4.

9. Bhandari A, Gupta A, Das D. Improvised Apriori Algorithm using frequent pattern tree for real time applications in data mining. arXiv preprint arXiv:1411.6224, 2014.
10. Cheng X, Su S, Xu S, et al. DP-Apriori: A differentially private frequent itemset mining algorithm based on transaction splitting. Computers and Security, 2015, 50: 74-90.
11. Păun. Gh. Computing with membranes. Journal of Computer and System Sciences, 2000, 61(1): 108-143.
12. Marti. C, Păun. Gh, Pazos. J. Tissue P systems. Theoretical Computer Science, 2003, 296(2): 295-326.
13. Ionescu. M, Păun. Gh, Yokomori. T. Spiking neural P systems. Fundamenta Informaticae, 2006, 71(2): 279-308.
14. Song. T, Pan. L, Jiang. K, Song. B, Chen. W. Normal forms for some classes of sequential spiking neural P systems. IEEE Transactions on NanoBioscience, 2013, 12(3): 255-264.
15. Song. T, Pan. L, Păun. Gh. Asynchronous spiking neural P systems with local synchronization. Information Sciences, 2012, 219: 197-207.
16. Zeng. X, Zhang. X, Pan. L. Homogeneous spiking neural P systems. Fundamenta Informaticae, 2009, 97(1): 275-294.
17. Song. T, Wang. X, Zhang. Z, Chen. Z. Homogenous spiking neural P systems with anti-spikes. Neural Computing and Applications, 2014, 24(7): 1833-1841.
18. Ibarra. O. H, Păun. A, Rodríguez-Patón. A. Sequential SNP systems based on min/max spike number. Theoretical Computer Science, 2009, 410(30-32): 2982-2991.
19. Song. T, Pan. L, Păun. Gh. Spiking neural P systems with rules on synapses. Theoretical Computer Science, 2014, 529: 82-95.
20. Romero-Campero. F. J, Pérez-Jiménez. M. J. Modelling gene expression control using P systems: The Lac Operon, a case study. Biosystems, 2008, 91(3): 438-457.
21. Enguix. G. B. Unstable P systems: applications to Linguistics. Membrane Computing, 2005, 3365: 190-209.
22. Song. T, Pan. L, Jiang. K, Song. B, Chen. W. Normal forms for some classes of sequential spiking neural P systems. IEEE Transactions on NanoBioscience, 2013, 12(3): 255-264.
23. Díaz-Pernil D, Berciano A, Peña-Cantillana F, et al. Segmenting images with gradient-based edge detection using Membrane Computing. Pattern Recognition Letters, 2013, 34(8): 846-855.
24. Freund R, Oswald M, Păun G. Catalytic and purely catalytic P systems and P automata: control mechanisms for obtaining computational completeness. Fundamenta Informaticae, 2015, 136(1-2): 59-84.
25. Freund R, Kari L, Oswald M, et al. Computationally universal P systems without priorities: two catalysts are sufficient. Theoretical Computer Science, 2005, 330(2): 251-266.
26. Gheorghe M, Krasnogor N, Camara M. P systems applications to systems biology. Biosystems, 2008, 91(3):435-7.
27. Romero-Campero F J, Pérez-Jiménez M J. Modelling gene expression control using P systems: the Lac Operon, a case study. BioSystems, 2008, 91(3): 438-457.
28. Enguix G B. Preliminaries about some possible applications of P systems in linguistics. Membrane Computing. Springer Berlin Heidelberg, 2002: 74-89.
29. Enguix G B. Unstable p systems: applications to linguistics. Membrane Computing. Springer Berlin Heidelberg, 2004: 190-209.

30. Andrei O, Ciobanu G, Lucanu D. A rewriting logic framework for operational semantics of membrane systems. Theoretical Computer Science, 2007, 373(3): 163-181.
31. Idowu R K, Maroosi A, Muniyandi R C, et al. An application of membrane computing to anomaly-based intrusion detection system. Procedia Technology, 2013, 11: 585-592.
32. Díaz-Pernil D, Berciano A, Peña-Cantillana F, et al. Segmenting images with gradient-based edge detection using Membrane Computing. Pattern Recognition Letters, 2013, 34(8): 846-855.
33. Peng H, Wang J, Pérez-Jiménez M J, et al. The framework of P systems applied to solve optimal watermarking problem. Signal Processing, 2014, 101: 256-265.
34. Păun G, Rozenberg G, Salomaa A. The Oxford handbook of membrane computing. Oxford University Press, Inc., 2010.
35. Păun G, Păun R. Membrane computing and economics: Numerical P systems. Fundamenta Informaticae, 2006, 73(1, 2): 213-227.
36. Pavel A, Arsene O, Buiu C. Enzymatic numerical P systems-a new class of membrane computing systems. Bio-Inspired Computing: Theories and Applications (BIC-TA), 2010 IEEE Fifth International Conference on. IEEE, 2010: 1331-1336.
37. Barbuti R, Maggiolo-Schettini A, Milazzo P, et al. Spatial P systems. Natural Computing, 2011, 10(1): 3-16.
38. Verlan S. Using the formal framework for P systems. Membrane Computing, 2013: 56-79.
39. Gheorghe M, Ipate F. Kernel P Systems - A Survey. Membrane Computing, 2013: 9-10.
40. Gazdag Z. Solving SAT by P systems with active membranes in linear time in the number of variables. Membrane Computing, 2013: 189-205.
41. Cienciala L, Ciencialova L, Langer M. Modelling of Surface Runo using 2D P colonies. Membrane Computing, 2013: 81-94.
42. Banu-Demergian I T, Stefanescu G. The geometric membrane structure of finite interactive systems scenarios. Membrane Computing, 63-80.

Extended Abstracts

Traces of Computations by Generalized Communicating P Systems (Extended Abstract)

Ákos Balaskó[1] and Erzsébet Csuhaj-Varjú[2]

[1] Computer and Automation Research Institute, Hungarian Academy of Sciences
Kende u. 13-17, H-1111 Budapest, Hungary
balasko@sztaki.hu
[2] Department of Algorithms and Their Applications, Faculty of Informatics,
Eötvös Loránd University, Pázmány Péter sétány 1/c, 1117 Budapest, Hungary
csuhaj@inf.elte.hu

1 Introduction

Generalized communicating P systems (GCPSs, for short), introduced in [9], are unconventional, Turing equivalent models of computation. Nevertheless, as is shown in [1], a computation of a GCPS can be used as an interpreter of a language in which complex compositions of web-services and/or applications (e.g. workflows) are defined. This option indicates several requirements to be satisfied by a workflow interpreter: one of them is reproducibility. As nondeterminism is one of the main characteristic of GCPSs, we cannot be sure that the same result can be reproduced from time to time if the computation is executed with the same input configuration. Hence, the best the system is able to provide is the trace of all the interactions among the cells during the computation noting that which rule applications were produced the next configuration. As a rule to be applied is selected non-deterministically from the multiple optional ones, the traces can be used to calculate the probability of obtaining this subsequent configuration.

2 Generalized Communicating P Systems

A generalized communicating P system (a GCPS, for short) is a variant of tissue-like P systems [6, 7]. Roughly speaking, it corresponds to a hypergraph where each node represents a cell and each edge is represented by a rule. Every node contains a multiset of objects which can be communicated, that is, objects may move between the cells according to communication rules (also called interaction rules). The form of a communication rule is $(a, i)(b, j) \to (a, k)(b, l)$ where a and b are objects and i, j, k, l are labels identifying the source and the target cells. Such a rule means that an object a from cell i and an object b from cell j move synchronously to cell k and cell l, respectively. The system is embedded in an environment (represented by cell 0) which may have certain types of objects in

an infinite number of copies and certain types of objects only in a finite number of copies. The GCPS and the environment interact by using the communication (interaction) rules given above, with the restriction that at every computation step only a finite number of objects may enter any cell from the environment. This is due to the fact that the set of communication rules is defined in such way that if two objects from the environment are moved to some other cell or cells, then at least one of them must not appear in the environment in an infinite number of copies. We note that although a GCPS realizes a graph structure, the cells are defined implicitly, since the system is given as a set of communication rules over an alphabet. For the formal details of the notion the reader is referred to [9].

The communication rules, by default, are applied in the maximally parallel manner, possibly implying changes in the configuration of the GCPS, i.e., changing the multisets representing the contents of the cells. A computation in a GCPS is a sequence of configurations directly following each other. If this sequence is finite, then we speak of a finite computation. A computation is called halting if starting from the initial configuration it ends in a configuration to which no communication rule can be applied. The result of the computation is the number of objects found in a distinguished cell, e.g., in the output cell in the halting configuration. Obviously, any halting computation is finite. GCPSs compute numbers (nonnegative integers), but they can be used for computing vectors of such numbers as well.

For more information on generalized communicating P systems consult [9, 4, 3, 5, 2] and [7].

3 Traces of Finite Computations in GCPSs

As it can easily be seen, each finite computation of a GCPS is realized by an ordered list of configurations which can be considered as a list of snapshots that records the contents of the cells in each step. Nevertheless, this list does not provide information on which types of rules and in how many copies they were applied in each step, thus the computation process cannot be reproduced. Thus, the following questions are open: Is it decidable whether or not a computation of length k in a GCPS is deterministic at the level of the applied rules, e.g., starting from a certain configuration there exists only one computation of length k or not, the multiset of communication rules applied in each step is the same or not, and, starting from the same configuration how many computations, different according to the multiset of applied rules, lead to the same result.

To provide more information on the process of computation, we add further information to the description by configuration sequences.

Suppose that the GCPS Π has l communication rules. A trace of the ith computation step of Π is a vector of $2l$ components, where the first l components specify the applicable communication rules in a given configuration: if the ith rule is applicable, then ith component of the vector is 1, otherwise it is 0. The other components, from $l+1$ to $2l$, are nonnegative numbers, indicating how

many copies of the given rule are performed in the given step. Notice that for a given configuration of Π, all traces resulting in a subsequent configuration can be computed.

By a trace matrix of Π of size s, we mean a matrix of s rows where each row is a trace of Π and for each j, $1 \leq j \leq s-1$, the $j+1$th row is the trace of a computation step that resulted from the trace of the jth step. The first row represents a trace of a computation step starting from the initial (first) configuration of this finite computation. In this way, we obtain finite sequences of matrices which describe the whole process of computations of length l, $l \geq 1$.

These representations make possible to study the functioning of GCPSs from several points of view. For example, since a GCPS can be considered as an application using its initial configuration as inputs and counts the result in the output cell, the comparison of trace matrices of different computations provide information on how many different computations provide the same result for the same input. Furthermore, which of them requires the minimal number of steps of computation? Some other question for further investigation is how many different computations are possible from a given configuration? Finally, robustness of a given GCPS with respect to change of the initial configuration would also be of interest, which could serve for classifying generalized communicating P systems.

Our work in progress deals with these kinds of questions. Our research concentrates on studying GCPSs computing Parikh vectors of D0L and DT0L sequences. D0L systems (deterministic interactionless Lindenmayer systems) are pure contex-free grammars where the rule set has exactly one rule for each symbol in the system, and by performing a direct derivation step all symbols in the word in generation are simultaneously rewritten (for details concerning these notions and their properties consult [8]). This implies that having a word w of a D0L sequence (a word obtained by performing a finite sequence of derivation steps from an initial word), the (only one) multiset of rules to be applied to obtain the next word in the sequence and the Parikh vector of the new word can be computed. We study the trace matrices of the simulating GCPSs and compare the results to the corresponding results obtained in the theory of Lindenmayer systems.

References

1. Balasko, A.: On a workflow model based on generalized communicating P systems, Journal of Computer Science, 17 (1), 45–68, AGH University of Science and Technology, Krakow (2016)
2. Balaskó, Á., Csuhaj-Varjú, E., Vaszil, Gy.: Dynamically Changing Environment for Generalized Communicating P Systems. Rozenberg, G. et al. (Eds.): Membrane Computing. LNCS 9504, pp. 92–105, Springer (2015)
3. Csuhaj-Varjú, E., Vaszil, Gy., Verlan, S.: On Generalized Communicating P Systems with One Symbol. M. Gheorghe et al. (Eds.): CMC 2010, LNCS 6501, pp. 160-174, Springer (2010)
4. Csuhaj-Varjú, E., Verlan, S.: On Generalized Communicating P systems with Minimal Interaction Rules. Theoretical Computer Science 412, 124-135 (2011)

5. Krishna, S.N., Gheorghe, M., Dragomir, C.: Some Classes of Generalised Communicating P Systems and Simple Kernel P Systems. P. Bonizzoni et. al (Eds.): CiE 2013, LNCS 7921, pp. 284-293, Springer (2013)
6. Păun, Gh.: Membrane Computing. An Introduction. Springer, Berlin (2002)
7. Păun, Gh., Rozenberg, G., Salomaa, A. (Eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
8. Rozenberg, G., Salomaa, A.: Handbook of Formal Languages, Vol. 1–3. Springer (1997)
9. Verlan, S., Bernardini, F., Gheorghe, M., Margenstern, M.: Generalized Communicating P systems. Theoretical Computer Science, 404 (1-2) 170–184 (2008)

Chemical Term Reduction with Active P Systems

Péter Battyányi, György Vaszil

Department of Computer Science, Faculty of Informatics
University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
{battyanyi.peter,vaszil.gyorgy}@inf.unideb.hu,

Extended Abstract

1 Introduction

In the following, we continue the investigations concerning the relationship of the "chemical calculus" of Banâtre and Le Métayer [5, 3] and membrane systems. In [6, 1] we have studied the possibilities of describing membrane system computations with chemical terms and their reduction sequences. Here we present an attempt to attack the problem from the "opposite direction", that is, to describe chemical terms and their reduction sequences by membrane system computations. Our chosen model is a variant of P systems with active membranes as presented in papers, such as [2].

The following brief presentation of the chemical calculus is based on [3] and [4]. Like Gamma programming, the chemical calculus is also inspired by the chemical metaphor: data are represented by γ-terms, which are called molecules, and reactions between them are represented by rewrite rules.

The syntactical elements are *molecules, reaction conditions*, and *patterns*, denoted by M, C and P, respectively. The special type of molecule

$$\gamma(P)[C].M$$

is called a γ-*abstraction* with pattern P, reaction condition C, result M. It encodes a rewriting rule: when the pattern P is respected and the condition C is met, a substituted variant of M is created as a result.

The way patterns are matched is defined by the notion of substitution, a mapping ϕ from the set of variables to the set of molecules, but instead of giving the formal details, we present the idea through an example below.

A (γ-)redex is a term of the form $(\gamma(P)[C].M, N)$. The reaction rule (γ-rule) is defined as $\gamma(P)[C].M, N \to \phi M$, where ϕ assigns values to variables in such a way that $\phi(C)$ reduces to *true*.

As an example, consider the γ-term $(\gamma(x, y)[x = y].x,\ 1, 2, 3, 1, 4)$ where (x, y) is a pattern, and $x = y$ is a reaction condition. In order for the condition to evaluate to *true*, the pattern needs to be matched so that $\phi(x) = \phi(y)$, and then the result is $(\gamma(x, y)[x = y].x,\ 1, 2, 3, 1, 4) \to (1, 2, 3, 4)$.

During the translation we are concerned with a simplified version of the γ-calculus: we consider only abstractions where the conditional part is constantly true. This simplification has no effect on the computational strength of the calculus.

2 P Systems Associated to γ-terms

Let M be a γ-calculus term. We define the P system associated with M as

$$\Pi = (O, \mu, w_1, \ldots, w_n, H, R_1, \ldots, R_n, \rho_1, \ldots, \rho_n),$$

a P system with active mebranes and the usual ingredients, using membrane dissolution, division, and so called subordination rules. We also use promoters and inhibitors assigned to rules. Moreover, besides the usual parities we introduce new values ? and ?? for unknown parities. $H = Var \cup \{\sigma, \rho, \rho_1, \rho_2\}$ is the set of labels, where Var is the set of variables occurring in M and $Atom$ is the set of atomic objects in M different from variables.

Intuitively, solutions are coded by the label σ, patterns and abstractions are coded as pairs, the first elements of which are the patterns labelled by ρ_1 and the second ones are the bodies of the abstractions with labels ρ_2. The variables occurring in patterns also appear as labels. Labels in the pattern part of an abstraction get an additional subscript p. The membrane implicitly containing the whole structure is the skin membrane. For example,

$$\gamma\langle x, y\rangle.(x, y, z) \rightsquigarrow [[[[]_{x_p}, []_{y_p}]_{\sigma_p}]_{\rho_1}, [[]_x, []_y, []_z]_{\rho_2}]_{\rho}.$$

We simulate one step of reduction in the γ-calculus with possible several steps of computation of the membrane system. The process is governed by control elements, most of the construction will follow the general pattern below:

$$control, condition \to result \quad > \quad control \to newcontrol,$$

where condition is usually a membrane or atom or a multiset of them and *newcontrol* triggers some new operations. First of all, we are going to find an abstraction for a possible redex, nondeterministically choose its arguments and then start the matching process. When two solutions matched inertness of the solution not in the pattern must be ensured. This compels us to embed a process for checking for inertness in the course of the reduction. When everything was successfully matched we begin substitutions for the variables in the non-pattern part of the abstraction. This highly leans on the rule of subordination.

To start the computation of the membrane system, we have a control element $?start$, which immediately creates the control element $?copy$. The element $?copy$ produces a copy of the original system and is rewritten for the new control element $?findredex$:

$$[?copy]^0_s \to [?findredex]^0_s, []^0_s.$$

The control element $?findredex$ finds an abstraction nondeterministically: it can permeate solutions so as to find redexes in subterms also. The new control element $?findarguments$ collects arguments by subordination in order to prepare for the reduction. Then reduction continues with the matching process by calling the new control element $?match$. As an illustration we give the process in detail.

$$?findredex, [\,]^0_\sigma \to [?findredex]^0_\sigma,$$
$$?findredex, [\,]^0_\rho \to [?findarguments]^+_\rho$$

Abstractions with parity + can accept other membranes in subordinations.

$$[\,]^+_\rho, [\,]^0_\alpha \to [[\,]^0_\alpha]^+_\rho$$

The element $?findarguments$ can intervene to prevent an abstraction from accepting more arguments:

$$[?findarguments \to ?findarguments]^+_\rho,$$
$$[?findarguments]^+_\rho \to [?match]^0_\rho.$$

The element $?match$ removes the membrane of the pattern part of the abstraction labelled by ρ_1, creates a new control element $?correct$, which is in charge for the correct termination of the matching process, then the matching process begins. A pattern variable can be matched to every type of term, while matching a solution with a solution needs a check for the inertness of the argument.

$$[\,]^0_{\sigma_p}, [\,]^0_\sigma \to [\,]^0_{\sigma_p}, [?startinert]^?_\sigma,$$
$$[\,]_{x_p}, [\,]^a_\alpha \to [[\,]^a_\alpha]^+_{x_p}, \quad \text{where } \alpha \in Var \cup \{\sigma, \rho\}.$$

If the solution with parity ? proves to be inert, then $?match$ removes the membranes with labels σ_p and the one with label σ and parity +, and the matching process can be continued inside the solutions. Otherwise, when the solution argument proves to be non-inert, then the whole reduction step fails, which manifests itself by clearing the duplicate of the skin membrane with parity ? and starting the whole process again with the copy of the whole membrane system reserved before.

When the matching terminates properly, only membranes with labels of pattern variables and a membrane with label ρ_2 should remain in the examined abstraction. In other words, every membrane with label σ should disappear. The control element $?correct$ sees to it. If this is not the case, again a $?matchfails$ process is called, otherwise we still have to check whether the matching assigns to equal variables equal membrane subsystems. That is, if $[\ulcorner M_1 \urcorner]^+_{x_p}$ and $[\ulcorner M_2 \urcorner]^+_{x_p}$ are elements obtained by virtue of the matching process, then $\ulcorner M_1 \urcorner = \ulcorner M_2 \urcorner$, where $\ulcorner M_i \urcorner$ is the membrane encoding of the term M_i. The control element $?equal$ is responsible for this investigation. The basic idea is: we order the variables occurring in our membrane system and we accomplish this equality check

for the variables separately, treating first the variable preceding the others in the ordering. Assuming the variable in question is x and we have n copies of membranes with label x_p, say $[^\ulcorner M_1 ^\urcorner]^+_{x_p}, \ldots, [^\ulcorner M_n ^\urcorner]^+_{x_p}$. We delete one outermost membrane at a time with label α in one of the membranes labelled x_p and then delete one outermost membrane with the same label α in all other membranes labelled x_p. The process terminates with empty membranes if and only if $^\ulcorner M_1 ^\urcorner, \ldots, ^\ulcorner M_n ^\urcorner$ were the same membrane subsystems. In every other case the element $?matchfails$ is created.

When we reach this point, either the reduction is failed or we have a set of substitutions labelled with pattern variables. The only task remaining is to accomplish the substitution in the membrane subsystem labelled by ρ_2 contained in the same abstraction. This is a straightforward task implemented with subsequent applications of the subordination rule. In the meantime, by dissolution, we have to make sure that we have an ample supply of elements labelled with pattern variables, which convey the representation of the substitution, and every variable of the membrane subsystem encoding the non-pattern part of the abstraction is substituted for. This can be checked again by a straightforward traversal of the membrane system labelled ρ_2.

What is left is the process of checking inertness of a solution. A solution $\langle N \rangle$ is inert, if there are no redexes in N only inside of other solutions. Thus checking for inertness in a membrane $[^\ulcorner N ^\urcorner]^a_\sigma$ involves examining every abstraction and every possible set of arguments so that they do not form a redex of the γ-calculus.

References

1. B. Aman, P. Battyányi, G. Ciobanu, Gy. Vaszil, Simulating P systems with membrane dissolution in a chemical calculus. *Natural Computing*, accepted.
2. A. Atanasiu, C. Martin-Vide, Recursive calculus with membranes. *Fundamenta Informaticae*, 49(1-3): 45-59.
3. J.P. Banâtre, P. Fradet, Y. Radenac, Principles of chemical programming. *Electronic Notes in Theoretical Computer Science* 124(1) (2005) 133–147.
4. J.P. Banâtre, P. Fradet, Y. Radenac, Generalized multisets for chemical programming. *Mathematical Structures in Computer Science* 16 (2006) 557–580.
5. J.P. Banâtre, D. Le Métayer, A new computational model and its discipline of programming. *Technical Report RR0566*, INRIA (1986).
6. P. Battyányi, Gy. Vaszil, Describing membrane computations with a chemical calculus. *Fundamenta Informaticae*, 134 (2014) 39–50.

Extensions of P Colonies (Extended Abstract)

Erzsébet Csuhaj-Varjú

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

Recently, there has been a growing interest in unconventional Turing equivalent computing devices and in computational models which "go beyond" Turing, i.e., which are able to compute more than recursively enumerable sets of strings or numbers. In membrane computing, we can find examples for both types of such constructs.

1 Red-Green Turing Machines

Red-green Turing machines, introduced in [9], are computing devices with computational power exceeding that of the standard Turing machines, since they recognize exactly the Σ_2-sets of the Arithmetical Hierarchy. These machines are deterministic and their state sets are divided into two disjoint sets, called the set of red states and the set of green states. They work on finite inputs with the recognition criterion on infinite runs that no red state is visited infinitely often and one or more green states are visited infinitely often. A change of the "color", i.e., a change from a green state to a red state or reversely is called a mind change. In [9], it was shown that any recursively enumerable language can be recognized by a red-green Turing machine with one mind change and if more than one mind changes may take place, then they are able to recognize the complement of any recursively enumerable language.

In [1], in the analogy of the concept of red-green Turing machines red-green counter machines (red-green register machines) were introduced and examined.

It was shown that the computations of a red-green Turing machine TM can be simulated by a red–green register machine RM with two registers and with string input in such a way that during the simulation of a transition of TM leading from a state p with color c to a state p' with color c' the simulating register machine uses instructions with labels (states) of color c and only in the last step of the simulation changes to a label (state) of color c'. Furthermore, the computations of a red–green register machine RM with an arbitrary number of registers and with string input can be simulated by a red–green Turing machine TM in such a way that during the simulation of a computation step of RM leading from an instruction with label (state) p with color c to an instruction with label (state) p' with color c' the simulating Turing machine stays in states of color c and only in the last step of the simulation changes to a state of color c'.

In [1] the notions of a red-green P automaton and its variants, as counterparts of red-green Turing machines in membrane computing, have also been defined and examined, demonstrating that these variants of P systems "go beyond" Turing.

2 P Colonies

P colonies [8] which combine features of tissue-like P systems and distributed systems of formal grammars called colonies [7] are examples for non-standard Turing equivalent computational models. A P colony consists of a finite number of agents (also called cells) and an environment. In the basic model, every agent is represented by a finite multiset of objects (the current state of the agent) and a finite number of programs. Each program consists of a finite number of rules for processing the objects of the agent. A rule of the agent is either an evolution rule (it changes an object of the agent to some other object) or a communication rule (it exchanges an object of the agent and an object of the environment). The environment is represented by a multiset of objects as well; at the beginning it consists of an infinite number of copies of a special object, e, called the environmental object.

P colonies work with direct changes of their configurations, also called computational steps. At each computational step every agent attempts to use one of its programs. The rules of the program have to be applied in parallel. If there is at least one applicable program, then the agent non-deterministically chooses one of them. At every step of the computation, the maximal possible number of agents has to perform a program and at every computational step each agent is represented by the same number of objects as at the beginning. This number is called the capacity of the agent. In the standard case, agents of the P colony are with capacity two. By applying programs, the P colony passes from one configuration to some other configuration. A sequence of direct configuration changes starting from the initial configuration is called a computation; if the P colony has no applicable program to the obtained configuration, then the computation is called halting. The result of the computation is the number of a distinguished object, f (called the final object), that can be found in the environment at halting. P colonies and their variants have been examined in detail during the years, it has been shown that they are computationally complete computing devices even with very restricted size parameters and programs with restricted forms (see, for example, [6]).

3 APCol Systems

According to the generic model of P colonies, the environment is represented by a multiset of objects. In [2] and [3] a new variant of P colonies was introduced, called an APCol system, where the environment is given as a string. The model provides the options of erasing and inserting symbols (even contexts) into and

from the string by communication rules; the environmental object (the basic object), e, plays the role of the empty word.

Formally, an APCol system is a construct $\Pi = (O, e, A_1, \ldots, A_n)$, $n \geq 1$, where O is an alphabet (the alphabet of objects), $e \in O$ (the basic object) and A_i, $1 \leq i \leq n$, are agents. Each agent is a triplet $A_i = (ee, P_i, F_i)$, where ee is the multiset of objects describing the initial state (initial content) of the agent, and $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$, $k_i \geq 1$, is a finite set of programs associated with the agent, where each program is a pair of rules. Each rule is in one of the following forms: $a \to b$ where $a, b \in O$ (an evolution rule) and $c \leftrightarrow d$ where $c, d \in O$ (a communication rule). $F_i \subseteq O^*$ is a finite set of multisets with cardinality two, called the set of final states (final contents) of agent A_i.

During the work of the APCol system, the agents perform programs. If an evolution rule $a \to b$ is applied, then object a in the state of the agent is rewritten to the object b. If a communication rule $c \leftrightarrow d$ is performed, then the object c inside the agent and a symbol d in the string are exchanged, the agent rewrites symbol d to symbol c in the string representing the environment. If $c = e$, then the agent erases d from the input string and if $d = e$, symbol c is inserted into the string. Since both rules in a program can be communication rules, an agent can work with two objects in the string representing the environment in the same computational step. In the case of a program $\langle a \leftrightarrow b; c \leftrightarrow d \rangle$, where $a, b, c, d \in (O \setminus \{e\}^*)$, a substring bd of the environment is replaced by string ac. The order of the rules in the program is significant, that is, in one computational step the agent can act only at one place and the change of the string depends both on the order of the rules in the program and on the interacting objects.

The program is said to be *restricted* if it consists of one rewriting and one communication rule. The APCol system is restricted if all of its agents have only restricted programs.

At the beginning of the work of the APCol system, the environment is given by a string $\omega \in (O \setminus \{e\})^*$ of objects. This string represents the initial state of the environment. Thus, an initial configuration of the APCol system is an $(n+1)$-tuple $c = (\omega; ee, \ldots, ee)$ where ω is the initial state of the environment and multiset ee is initial state of agent i, $1 \leq i \leq n$.

A configuration of an APCol system Π is given by $(w; w_1, \ldots, w_n)$, where w_i, $1 \leq i \leq n$, is a multiset of objects with two elements representing the state of the i-th agent and $w \in (O - \{e\})^*$ is the environment to be processed.

A configuration $c = (w; w_1, \ldots, w_n)$ of Π is directly changed to configuration $c' = (w'; w'_1, \ldots, w'_n)$ if c' is obtained from c in such way that in configuration c a maximal number of agents performs one of its own programs simultaneously and the resulting configuration is c'. The direct change of a configuration is also called a computational step.

A sequence of configurations starting from the initial configuration is called a computation. A configuration is halting if no agent of the APCol system has any applicable program.

The result of a computation depends on the working mode of the APCol system. In the case of the accepting mode, the string ω (the initial state of the

environment) is accepted by the APCol system Π if there exists a computation by Π such that it starts in the initial configuration $(\omega; ee, \ldots, ee)$ and the computation ends by halting in the configuration $(\varepsilon; w_1, \ldots, w_n)$, where $w_i \in F_i$ for at least one w_i, $1 \leq i \leq n$. The language accepted by Π is the set of all (environmental) words over $(O \setminus \{e\})$ which are accepted by Π. The other working mode of an APCol system is the generating mode. The string w_F is generated by Π if there exists a computation starting in an initial configuration $(\varepsilon; ee, \ldots, ee)$ and the computation ends by halting in configuration $(w_F; w_1, \ldots, w_n)$, where $w_i \in F_i$ for at least one w_i, $1 \leq i \leq n$. The language generated by Π is the set of all (environmental) words over $(O \setminus \{e\})$ which are generated by Π.

In [2] it was shown that any recursively enumerable language can be obtained as a projection of a language of an APCol system with two agents working in the accepting mode. In [3] it was proved that any recursively set of numbers can be computed by a restricted APCol system with two agents working in the generating mode.

4 Extensions of P Colonies

We develop the concept of P colonies in two directions. First, we propose the concept of *P colonies with teams* ((APCol systems with teams), where the team is a finite number of agents of the P colony (APCol system). These teams can be so-called prescribed teams (given together with the components of the P colony) or so-called free teams where only the size of the teams, i.e., number of agents in the team is given in advance. The notion is inspired by the concept of team grammar systems (see [5]). P colonies (APCol systems) with prescribed or with free teams function in the following manner: at any computation step only one team is allowed to work (only one team is active) and all of its components should perform a program in parallel. P colonies (APCol systems) with prescribed teams are computationally complete computing devices.

In the analogy of red-green Turing machines (red-green register machines), we can introduce *red-green APCol systems*. These constructs are APCol systems with teams where the set of teams is divided into two disjoint subsets, the set of red teams and the set of green teams. A red-green APCol system recognizes a string if during infinite runs on the string no red team is active infinitely often and one or more green teams are active infinitely often. The APCol system performs a mind change if an action of a red team is followed by an action of a green team and vice versa. The proofs of the computational completeness of APCol systems in [2] and in [3] are based on simulations of counter machines (register machines) with APCol systems. Modifying the constructions accordingly, it can be shown that the transitions of a red-green counter (register) machines can be simulated by action sequences of teams of red-green APCol systems, furthermore, during the simulating action sequence of the red-green APCol system as many mind changes take place as the simulated transition of the red-green counter (register) machine means. Thus, we obtain that the computational power of red-green APCol systems "goes beyond" Turing.

We plan further investigations in these topics.

References

1. Aman, B., Csuhaj-Varjú, E., Freund, R.: Red-Green P Automata. In: Gheorghe, M. et al. (eds.): CMC 2014, LNCS, vol. 8961, pp. 139–157, Springer (2014)
2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: P Colonies Processing Strings. Fundamenta Informaticae 134(1-2), 51–65 (2014)
3. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: A class of restricted P colonies with string environment. Submitted, 2015.
4. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, Gy.: Computing with cells in environment: P colonies. Journal of Multi-Valued Logic and Soft Computing 12, 201–215 (2006)
5. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, Vol. I-III. Springer Verlag, Berlin, Heidelberg, New York (1997)
6. Kelemenová, A.: P Colonies. Chapter 23.1, In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 584–593. Oxford University Press (2010)
7. Kelemen, J., Kelemenová, A.: A grammar-theoretic treatment of multi-agent systems. Cybernetics and Systems 23, 621–633 (1992)
8. Kelemen, J., Kelemenová, A., Păun, Gh.: Preview of P colonies: A biochemically inspired computing model. In: Bedau, M. et al. (eds.) Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX), pp. 82–86. Boston Mass. (2004)
9. van Leeuwen, J., Wiedermann, J.: Computation as an Unbounded Process. Theoretical Computer Science 429, 202–212 (2012)

On minimal multiset grammars

Giuditta Franco[1]

Dipartimento di Informatica, Università di Verona, Italy.

Abstract. A biological network modeled by a multiset grammar may be investigated from a dynamical viewpoint by a recurrence system. An interesting connection among computation by a multiset grammar, linear algebra, and recurrent dynamics is here discussed along with questions of minimality and covering.

Biochemical networks (usually having much more interactions/edges than substances/nodes) may be modeled by a set of (thousands of) rewriting rules (see, for example [11]). A computationally efficient study of such a model could start from a reduction of the number of rules to a minimal set which guarantees the dynamical properties of interest, for all variables involved in the system. Similarly, in recent literature of synthetic biology, synthetic genomes are generated by suitable assemblying of a minimal set of genes necessary to keep the cell working, or performing a given specific function. For instance, the genome of *Mycoplasma mycoides* was replaced with a synthetic genome in 2010 [5], while more complex organisms (of yeast species) have been synthetically generated more recently [1, 6]. In these cases, original genomic sequences are "edited", namely by specific deletion of "dispensable" DNA sequences and by replacement of particular regions by others performing the same task. Deletion of the "non-essential genes" allows the genome size to be reduced: a feature designed to determine the smallest cohorts of genes required to perform a given function (or necessary for survival under a particular growth condition).

Motivated by the above approaches to model biological dynamics and functions, here we would like to search for the smallest cohorts of rewriting rules required to exhibit the dynamics of a given multiset grammar. In this extended abstract a discussion is proposed, to define a *minimal* grammar, that is, having a minimal set of rules, which guarantees a given systemic dynamical pattern (e.g., periodic, oscillating, increasing, decreasing ...). Observations in the vectorial spaces where substances and rules may be seen allow us to consider also some covering issues.

This is a preliminary work ispired by problems opened up in [3], in the context of metabolic computing [7]. Moreover, according to a current trend of membrane computing, P systems are investigated having integer vectors as states (where object multiplicities do not need to be positive numbers), which means that substances are present in arbitrary quantity.

The intent to employ multiset processing as a framework to study real biological systems is vivid in literature, along with variants of P systems enriched with other features, often inspired by biology [4], and having a specific rule application

strategy different from the traditional nondeterministic maximal parallelism [2]. Models for metabolism as MP systems [8] may be mentioned as a very applicative trend of membrane computing. They are mono-membrane multiset rewriting grammars, where rules are regulated by specific state functions.

1 A minimization problem

Deterministic evolution of macrostates of an abstract container of objects, with integer multiplicities, transformed by given rewriting rules, may be naturally modeled by a multiset grammar, where all rules are applied in parallel at each step. An application strategy is established for the number of times that each rule is applied. Indeed, on multisets with integer multiplicities rules have no conflict to be applied, as in some traditional P models, where an environment was assumed to provide our system with unlimited resourses. Our interest is focused on the dynamics of the system, that is, on the integer vectors (related to the objects) variation.

In other terms, we consider a one-membrane P system with integer multiplicities, having numerical vectors emerging as a sequence of states computed by transitions, where rules are applied according to a given systemic strategy.

Example 1. As a toy example, the reactions $r_1 : ab \to aa$, $r_2 : bcc \to a$, $r_3 : abc \to bb$ may be all applied in one transition, *each of them once*, to modify an arbitrary (initial) state (α, β, γ), which is an integer vector. The transition is computed by means of the stoichiometric matrix $\mathcal{M} = \begin{pmatrix} 1 & 1 & -1 \\ -1 & -1 & 1 \\ 0 & -2 & -1 \end{pmatrix}$, according to the equation $\mathcal{M} \times (1,1,1) + (\alpha, \beta, \gamma) = (\alpha + 1, \beta - 1, \gamma - 3)$, where $\times$ denotes the ordinary matrix product.

In general, given a multiset grammar over the alphabet $X = \{x_1, x_2, \ldots, x_n\}$, with rules $R = \{r_1, r_2, \ldots, r_m\}$, with $m \geq n$, if (in one transition) we apply the first rule k_1 times, the second k_2 times, and so on, then the following recurrent dynamics describes such a grammar computation: $X[i+1] = k_1 v_1 + k_2 v_2 + k_m v_m + X[i]$, where $v_1, v_2, \ldots, v_m$ are the m columns of the stochiometric matrix $\mathcal{M}$, $U = (k_1, k_2, \ldots, k_m) \in \mathcal{N}^m$, i the computational step, and $X[i]$ the state of the system (n-dimensional integer vector related to the symbols of the alphabet). Each n-dimensional column v_i reports the single effect of one application of the rule r_i on the variation of the substances, while k_i represents the number of times the rule r_i is applied.

In conclusion, the dynamics of the system is given by a recurrence equation, which updates the current state $X[i]$ by adding the vector $\mathcal{M} \times U$, given by the specific linear combination $k_1 v_1 + k_2 v_2 + k_m v_m$ of the m vectors of Z^n corresponding to the rules. Our goal is to minimize the number of rules in the system, by keeping unaltered such a state variation (i.e., the dynamics).

If the vector U is constant, the dynamics of X is linear, that is, each variable increases or decreases monotonically. In order to have more complex curves, we

need a vector U which depends on the state, as it is usually the case for multiset grammars in both traditional P systems (where the number of times to apply rules depends on the current multiset) and MP systems, where U is given by state functions computed by data-driven regression [9]. It has been shown [10, 7] that very complex functions may be computed by such a discrete model of computation.

Let us first discuss the minimality question under the assumption of U constant (m-dimensional vector of positive natural numbers) and linear dynamics for the variables. To be able to reduce the number of rules (i.e., vectors v_i, $i = 1, \ldots, m$) of course we need to change the given U into another positive vector U' with a minor number of components.

Then, our open question has been now reduced to a problem of linear algebra: *which is the minimum number p of rules, with $p \leq m$, such that: $k_{j_1} v_{j_1} + k_{j_2} v_{j_2} + \ldots, k_{j_p} v_{j_p} = k_1 v_1 + k_2 v_2 \ldots + k_m v_m$ and $j_i \in \{1, 2, \ldots, m\}$ for $i \in \{1, 2, \ldots, p\}$?*

In the toy system above, we have a state variation of $(1, -1, -3)$, with $U = (1, 1, 1)$, and by basic algebra one realizes that $4v_1 + 3v_3 = v_1 + v_2 + v_3$. The minimal set of rules to get our linear dynamics is $\{r_1, r_3\}$. In fact, the cardinality of such a set cannot be further riduced, as no column of $\mathcal{M}$ is a multiple of $(1, -1, -3)$.

A conjecture here is that the minimum number p is greater or equal to the number of linearly independent columns of $\mathcal{M}$, that is $rank_{\mathcal{M}} \leq p \leq m$. A more general conjecture claims that this property is true even in the case of U being a positive state function (rather than a constant vector). In this general case, we have a possible variation of p at each computational step, and we look for an interval of possible values for p, which would be included in $[rank_{\mathcal{M}}, m]$.

2 A covering issue

Looking at a multiset grammar as a couple of sets, one in Z^n (substances x) and the other in Z^m (associated to rules r of the type $\alpha_r \to \beta_r$), recalls the notion of duality between vectorial spaces of functions, since substances and rewriting rules are clearly connected. For instance, any substance x may be associated to the set of rules having x as a reactant (i.e., occurring in α_r), denoted by $A(x)$, which is a singleton $\{r\}$ iff the substance x is a reactant only for the rule r.

Definition 1. *A set $S \subseteq R$ is called covering set if $\forall x \in X \; \exists r \in R(x)$ such that $r \in S$.*

In other terms, S is a covering set iff $\forall x \in X \quad A(x) \cap S \neq \emptyset$. If both the sets of substances and rules are not empty, the cardinality of a covering set is at least 1 and at most m (R is obviously a covering set).

A minimal set of rules (in the sense defined in previous section) needs to have such a *covering property*, which is necessary for the system to compute a dynamics for each substance. Viceversa of couse is not true: in our toy example, $\{r_3\}$ is a covering but is not a minimal set.

Any set of n linearly independent rules is a covering (as each substance is involved by at least one rule) [3]. If our conjecture is correct, then a covering set is included in a minimal set.

If we add a rule to a covering set we still have a covering set (by definition). In this context, if we have to choose between the covering sets $\{xyz \to a\}$ and $\{xy \to b,\ z \to c\}$, then we prefer the first choice.

Let us recall a notion related to context sensitive rules that is employed by the algorithm presented in the following.

Definition 2. *Given X and $r \in R$, we call* radius of r with respect to X *the number $\eta_r^{(X)}$ of distinct symbols of X present in α_r.*

We notice that a rule r has only one occurrence in the sets $A(x)$ with $x \in X$ iff $\eta_r^{(X)} = 1$, and in general it occurs as many times as the number of distinct symbols occurring in α_r. Hence, in this context, the radius with respect to X measures the multiplicity of a rule in the multiset $\cup_{x \in X} R(x)$.

The basic idea of the following algorithm finding covering sets (in worst-case polynomial time) is to pick up the most frequent rule (i.e., having the maximal number of occurrences) in the sets $A(x)$ with $x \in X_0$ (initially $X_0 = X$), and then to update the set X_0 by deleting all the substances which have been covered with that rule. It is quite straightforward that by iterating this process until X_0 is empty, one gets a covering set S, having a number of elements which belongs to $\{1, 2, \ldots, n\}$.

Input: X, R from a given MP system.

Let L, ξ, l_r, η_r, with $r \in R$, be integers initialized to zero.
$X_0 := X$; $W := \emptyset$; $S := \emptyset$;

while $X_0 \neq \emptyset$ **do**

begin

1. For each $x \in X_0$, compute $R(x)$;
2. For each rule $r \in \cup_{x \in X_0} R(x)$, compute its radius $\eta_r := \eta_r^{(X_0)}$, and let $\xi := max_{r \in \cup_{x \in X_0} R(x)} \{\eta_r\}$;
3. $W := \{r \mid \eta_r = \xi\}$;
4. For each rule $r \in W$, compute $l_r = |\cup_{r \in R(x), x \in X_0} R(x)|$, and let $L = max_{r \in W} \{l_r\}$.
5. Choose one $r \in W$ such that $l_r = L$, and
   (a) $S := S \cup \{r\}$;
   (b) $X_0 := X_0 \backslash \{x \mid r \in R(x)\}$;

end

Output: S.

In words, the algorithm keeps the set X_0 updated with all the substances which still need to be covered (initially it is equal to X, while in each cycle at least one of its element is deleted). Referring to the current set of substances (step 1), the most frequent rules are chosen and settled down in the work-set W (steps 2 and 3).

Among the rules of W, those related to a maximum number of other rules are selected (by step 4), in order to avoid useless computation in the next steps (also intuitively: since we want to take a minimal number of rules, we choose those which eliminate most of the other candidates). This step only exists to limit computational steps.

One out of the rules which are most frequent and which eliminate most of the other rules is finally picked up (step 5): it is added to the set S, by 5(a), and the set X_0 is updated accordingly to this choice, by 5(b). That is, all the substances covered by the rule just chosen are deleted, as they do not need to be covered anymore.

Remark: Whatever rule we choose at step 5, the same number of substances is deleted in X_0, as the choice is performed among rules of maximal radius. Therefore, both the number of steps of the algorithm and the cardinality of the output set are not affected by the nondeterministic choice at step 5.

Proof. (Algorithm correctness). The output set S is a covering set by construction, as it contains rules which exhausted the set X (i.e., which covered all the substances of X). For each rule which is added in S, at the end of each cycle (step 5(a)), at least one substance is deleted in X_0 (step 5(b)) which initially has cardinality n. Then, X_0 is empty after $k \leq n$ cycles, and the output set S has cardinality k. □

This algorithm computes only one of the covering sets, but others with the same cardinality can be computed, actually as many as the product of the number of rules having maximum radius and maximum L after each cycle (that is, after each updating of X_0). In other words, there are as many covering sets with the cardinality of S as the number of possible combinations of choices the algorithm performs at step 5.

We point out that the algorithm finds a covering set of cardinality n only if the system has n rules with radius one and acting on different substances.

References

1. N. Annaluru et al., *Total synthesis of a functional designer eukaryotic chromosome*, Science **344(6186)**:816, 2014.
2. A. Castellini et al., *Data analysis pipeline from laboratory to MP models*, Natural Computing **10**:55-76, 2011.
3. G. Franco et al., Regulation and covering problems in MP systems, LNCS 5957, pp 242-251, 2010.
4. P. Frisco et al. eds, *Applications of Membrane Computing in Systems and Synthetic Biology*, Springer, 2014.

5. D. G. Gibson et al., *Creation of a bacterial cell controlled by a chemically synthesized genome*, Science **329(5987)**:52-6, 2010.
6. D. G. Gibson et al., *Synthetic Biology: Construction of a Yeast Chromosome*, Nature **509**:168-169, 2014.
7. V. Manca, *Infobiotics*, Springer-Verlag, 2013.
8. V. Manca et al., *Metabolic P Systems: A Discrete Model for Biological Dynamics*, Chinese Journal of Electronics **22**:717–723, 2013.
9. V. Manca et al. *Recurrent Solutions to Dynamics Inverse Problems: A Validation of MP Regression*, J. of Applied & Computational Mathematics **3**:1–8, 2014.
10. V. Mante et al., *Context-dependent computation by recurrent dynamics in prefrontal cortex*, Nature **503**(7474):78–84, 2013.
11. R. Pagliarini et al., *Combining flux balance analysis and model checking for metabolic network validation and analysis*, Natural Computing **14**(3):341–354, 2015.

Transmission Line Fault Classification Based on Fuzzy Reasoning Spiking Neural P Systems

Kang Huang

School of Electrical Engineering, Southwest Jiaotong University,
Chengdu, 610031, P.R. China
email: luodanhk@126.com

Abstract. A scheme for fault classification of transmission lines based on rFRSN P systems is presented. The fuzzy production rules of fault classification are first described. Then the fault classification model based on rFRSN P systems is established to identify fault types. Finally, the introduced method is verified to be effective with high classification accuracy and is not affected by various fault conditions, such as fault inception angles, fault resistance, and fault locations, etc.

Keywords: Fault classification, rFRSN P systems, fuzzy production rules

1 Introduction

Fault classification is the most important task involved in transmission line protection, which must be accomplished in a way which is as fast and accurate as possible to isolate the system from a fault point and recover it after a fault occurs [1–3]. Of various fault classification methods, a fuzzy reasoning spiking neural P system is a promising choice [4–10]. This is also considered as a promising practical use of spiking neural P systems [11–14], even of P systems in general[15, 16]. In this paper, a novel fault classification scheme based on fuzzy reasoning spiking neural P systems with real numbers (rFRSN P systems) , which are applied to build a fault classification model to identify fault types on the basis of fault features, is presented.

2 Fault Classification Models Based on rFRSN P Systems

2.1 Fuzzy Production Rules of Fault Classification

When a fault occurs, the fault component current of the fault phase changes significantly, while the fault component current of the sound phase changes little. And in case of phase to phase faults and three phase fault, the zero sequence current is theoretically zero, while in case of ground faults, the zero sequence current is significant than that under normal conditions, and the fault component zero-sequence current of phase to phase fault is very small. It should be noted that the ***large*** and ***small*** above-mentioned are fuzzy knowledge which is on behalf of size of transient feature value. Therefore, the fuzzy production rules of fault classification are described as follows. In addition,

let s_a, s_b, s_c and s_0 stand for phase a, b, c and 0, respectively

R_1:IF (s_a is ***large***) AND (s_b is ***small***) AND (s_c is ***small***) AND (s_0 is ***large***) THEN A_g
R_2:IF (s_a is ***small***) AND (s_b is ***large***) AND (s_c is ***small***) AND (s_0 is ***large***) THEN B_g
R_3:IF (s_a is ***small***) AND (s_b is ***small***) AND (s_c is ***large***) AND (s_0 is ***large***) THEN C_g
R_4:IF (s_a is ***large***) AND (s_b is ***large***) AND (s_c is ***small***) AND (s_0 is ***large***) THEN AB_g
R_5:IF (s_a is ***small***) AND (s_b is ***large***) AND (s_c is ***large***) AND (s_0 is ***large***) THEN BC_g
R_6:IF (s_a is ***large***) AND (s_b is ***small***) AND (s_c is ***large***) AND (s_0 is ***large***) THEN CA_g
R_7:IF (s_a is ***large***) AND (s_b is ***large***) AND (s_c is ***small***) AND (s_0 is ***small***) THEN AB
R_8:IF (s_a is ***small***) AND (s_b is ***large***) AND (s_c is ***large***) AND (s_0 is ***small***) THEN BC
R_9:IF (s_a is ***large***) AND (s_b is ***small***) AND (s_c is ***large***) AND (s_0 is ***small***) THEN CA
R_{10}:IF (s_a is ***large***) AND (s_b is ***large***) AND (s_c is ***large***) AND (s_0 is ***small***) THEN ABC

2.2 Fault Classification Models

On the basis of fuzzy production rules, a fault classification model based on rFRSN P systems is built according to the above mentioned fuzzy production rules, where A_l, A_s, B_l, B_s, C_l, C_s, 0_l and 0_s are on behalf of propositions "s_a is ***large***", "s_a is ***small***", "s_b is ***large***", "s_b is ***small***", "s_c is ***large***", "s_c is ***small***", "s_0 is ***large***", "s_0 is ***small***", respectively. The corresponding rFRSN P system for fault classification is defined as follows:

$$\Pi_1 = (O, \sigma_1, \sigma_2, \ldots, \sigma_{28}, syn, in, out)$$

where

1) O=$\{a\}$ is the singleton alphabet ($\{a\}$ is called spike);

2) $\sigma_1, \sigma_2, \ldots, \sigma_{18}$ are proposition neurons corresponding to the propositions with fuzzy truth values $\theta_1, \theta_2, \ldots, \theta_{18}$;

3) $\sigma_{19}, \sigma_{20}, \sigma_{28}$ are *and* rule neurons;

4) $syn = \{(1,19), (1,22), (1,23), (1,25), (1,26), (1,28), (2,20), (2,21), (2,24), (2,27), (3,20), (3,22), (3,24), (3,25), (3,27), (3,28), (4,19), (4,21), (4,23), (4,26), (5,21), (5,23), (5,24), (5,26), (5,27), (5,28), (6,19), (6,20), (6,22), (6,25), (7,19), (7,20), (7,21), (7,22), (7,23), (7,24), (8,25), (8,26), (8,27), (8,28), (19,9), (20,10), (21,11), (22,12), (23,13), (24,14), (25,15), (26,16), (27,17), (28,18)\}$;

5) in=$\{\sigma_1, \sigma_2, \ldots, \sigma_8\}$, out=$\{\sigma_9, \sigma_{10}, \ldots, \sigma_{18}\}$.

3 Simulation and Robustness Analysis

The considered two-machine three-phase power system is simulated on PSCAD/EMTDC for producing fault samples to test the performance of our rFRSN P System-based fault classifier. The Bergeron line model of PSCAD/EMTDC is considered for transmission line. The proposed fault classification method only uses the current signals of three phases. The sampling frequency is set to 50kHz.

To test the effectiveness of different transmission line parameters on the accuracy of the presented classification method, two transmission lines with different parameters

have been simulated to produce 100 test samples. The classification results show that the accuracy is 100% even when the transmission line is changed. In other words, the presented method in this paper has advantages in robustness to line parameters, and is suitable for other transmission lines.

4 Conclusions

In this paper, rFRSN P systems are applied to classify fault types for power system transmission line. A reasoning algorithm for the rFRSN P systems is used for fault reasoning to obtain confidence levels of different fault types. Simulation tests show that the presented method is effective in fault classification under various fault inception angles, various fault resistance, various fault locations.

References

1. Silva, K.M., Souza, B.A., Brito, N.S.D.: Fault Detection and Classification in Transmission Lines Based on Wavelet Transform and ANN. IEEE Transactions on Power Delivery, 21(4), 2058–2063 (2006)
2. Wang, H.S., Keerthipala, W.W.L.: Fuzzy-neuro Approach to Fault Classification for Transmission Line Protection. IEEE Transactions on Power Delivery, 13(4), 1093–1104 (1998)
3. Youssef, O.A.S.: Combined Fuzzy-Logic Wavelet-based Fault Classification Technique for Power System Relaying. IEEE Transactions on Power Delivery, 19(2), 582–589 (2004)
4. Wang, T., Zhang, G., Pérez-Jiménez, M.J.: Fuzzy membrane computing: theory and applications. International Journal of Computers, Communications & Control, 10(6), 904–935 (2015).
5. Peng, H., Wang, J., Pérez-Jiménez, M.J., Wang, H., Shao, J., Wang,T.: Fuzzy reasoning spiking neural P system for fault diagnosis, Information Sciences, 235, 106–116 (2013), .
6. Wang, J., Shi, P., Peng, H., Pérez-Jiménez, M.J., Wang,T.: Weighted Fuzzy Spiking Neural P Systems. IEEE Transactions on Fuzzy Systems 21(2), 209-220 (2013)
7. Zhang, G.X., Cheng, J.X., Wang, T., Wang, X.Y., Zhu, J.: Membrane Computing: Theory and Applications. Beijing: The Science Publishing Company, 2014.
8. Wang, T., Zhang, G.X., Rong, H.N., Pérez-Jiménez, M.J.: Application of Fuzzy Reasoning Spiking Neural P Systems to Fault Diagnosis. International Journal of Computers, Communications and Control, 9(6), 786-799 (2014)
9. Zhang, G.X., Rong, H.N., Neri, F., Pérez-Jiménez, M.J.: An Optimization Spiking Neural P system for Approximately Solving Combinatorial Optimization Problems. International Journal of Neural Systems, 24(5), 1440006 (16 pages) (2014)
10. Wang, T., Zhang, G.X., Zhao, J.B., He, Z.Y., Wang, J., Pérez-Jiménez, M.J.: Fault Diagnosis of Electric Power Systems Based on Fuzzy Reasoning Spiking Neural P Systems. IEEE Transactions on Power Systems, 30(3), 1182–1194 (2015)
11. Ionescu, M., Păun, Gh., Yokomori, T.: Spiking Neural P Systems. Fundameta Informaticae, 71(2-3), 279–308 (2006)
12. Song, T., Pan, L., Păun, Gh.: Asynchronous Spiking Neural P Systems with Local Synchronization, Information Sciences, 219, 197–207 (2013)
13. Zeng, X., Zhang, X., Song, T., Pan, L.: Spiking Neural P Systems with Thresholds, Neural Computation, 26(7), 1340–1361 (2014).
14. Zhang, X., Wang, B., Pan, L.: Spiking neural P systems with a generalized use of rules, Neural Computation, 26(12), 2925–2943 (2014).

15. Păun, Gh.: Computing with Membranes. Journal of Computer and System Sciences 61(1), 108-143 (2000)
16. Zhang, G., Gheorghe, M., Pan, L., Pérez-Jiménez, M.J.: Evolutionary membrane computing: a comprehensive survey and new results, Information Sciences, 279, 528–551 (2014).

Recent Results and Problems Concerning P Colony Automata

Kristóf Kántor, György Vaszil

Department of Computer Science, Faculty of Informatics
University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
{kantor.kristof, vaszil.gyorgy}@inf.unideb.hu

Extended Abstract

1 Introduction

P colonies are variants of very simple membrane systems, similar to so-called colonies of simple grammars, see [7]. These are collections of very simple generative grammars, but as systems, they are able to generate complicated languages.

Similarly to the grammar systems variant, P colonies also consist of a collection of very simple computing agents which interact in a shared environment, see [8, 9]. The environment and the computing agents are both described by multisets of objects which are processed by the colony members using rules which enable the transformation of the objects and the exchange of objects between the colony members and the environment. The rules are grouped into programs, which execute the rules they contain in parallel. A computation consists of a sequence of computational steps during which the colony members execute their programs in parallel, until the system reaches a halting configuration.

P colony automata, a variant of P colonies characterizing string languages instead of multiset collections were introduced in [2] where several of its variants were shown to be computationally complete.

Generalized P colony automata were introduced in [5] in order to make the model resemble more to the standard models of membrane computing, in particular, to the model of P automata, introduced in [4]. In this case, the computation of the colony defines an accepted multiset sequence, which is turned into an accepted string (a set of accepted strings, in general) by a non-erasing mapping (as in P automata).

2 Definitions

Definition 1. A *genPCol automaton* of capacity k and with n cells, $k, n \geq 1$, is a construct $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ where

- V is the *alphabet* of the automaton, its elements are called *objects*;
- $e \in V$ is the *environmental object* of the automaton;

- $w_E \in (V - \{e\})^*$ is a string representing the multiset of objects different from e which is found in the environment initially;
- $(w_i, P_i), 1 \leq i \leq n$, specifies the i-th *cell* where w_i is a multiset over V, it determines the initial contents of the cell, and its cardinality $|w_i| = k$ is called the *capacity* of the system. (Note that the cells may also contain the environmental object e initially.) The sets P_i of *programs* are formed from k rules of the following types:
  - *tape rules* of the form $a \xrightarrow{T} b$, or $a \overset{T}{\leftrightarrow} b$, called rewriting tape rules and communication tape rules, respectively; or
  - *nontape rules* of the form $a \to b$, or $a \leftrightarrow b$, called rewriting (nontape) rules and communication (nontape) rules, respectively,

  where $a, b \in V$.
  A program is called a *tape program* if it contains at least one tape rule.
- F is a set of *accepting configurations* of the automaton which we will specify in more detail below.

A genPCol automaton reads an input word during a computation. A part of the input (possibly consisting of more than one symbols) is read during each configuration change: the processed part of the input corresponds to the multiset of symbols introduced by the tape rules of the system. The application of a tape program containing the rule $a \xrightarrow{T} b$ or the rule $a \overset{T}{\leftrightarrow} b$ introduces the object a into the multiset of symbols read by the system. For a program p, we denote by $read(p)$ the multiset of all such objects, that is, the multiset of objects from the left sides of the tape rules contained by p.

A *configuration* of a genPCol automaton is an $(n+1)$-tuple $(u_E, u_1, \ldots, u_n)$, where $u_E \in (V - \{e\})^*$ represents the multiset of objects different from e in the environment, and $u_i \in V^*,\ 1 \leq i \leq n$, represents the contents of the i-th cell. The *initial configuration* is given by $(w_E, w_1, \ldots, w_n)$, the initial contents of the environment and the cells. The elements of the set F of *accepting configurations* are given as configurations of the form $(v_E, v_1, \ldots, v_n)$.

Definition 2. Let $\Pi = (V, e, w_E, (w_1, P_1), \ldots, (w_n, P_n), F)$ be a genPCol automaton. The *set of input sequences accepted by* Π is defined as

$$A(\Pi) = \{u_1 u_2 \ldots u_s \mid u_i \in (V - \{e\})^*,\ 1 \leq i \leq s, \text{ and there is a configuration sequence } c_0, \ldots, c_s, \text{ with } c_0 = (w_E, w_1, \ldots, w_n),\ c_s \in F, \text{ and } c_i \Longrightarrow c_{i+1} \text{ with } \bigcup_{p \in P_{c_i}} read(p) = u_{i+1} \text{ for all } 0 \leq i \leq s-1\}$$

where P_{c_i} is the set of programs applied by the components in the computational step from c_i to c_{i+1}.

The idea behind genPCol automata is that instead of the different computational modes used in [2], we have a system with programs and we apply the programs in the maximally parallel way as usual in P colonies, that is, in each computational step, every component cell must non-deterministically choose and

apply one of its applicable programs. Then we look at those rules which were tape rules (in the applied set of programs) and collect all the symbols that they "read": this multiset (of the collected symbols) is the multiset read by the system in the given computational step. A successful computation defines in this way an accepted sequence of multisets: the sequence of multisets entering the system during the steps of the computation.

The words accepted by the automaton are otained by applying an input mapping to the accepted mutliset sequences.

Definition 3. Let Π be a genPCol automaton, and let $f : (V - \{e\})^* \to 2^{\Sigma^*}$ be a mapping, such that $f(u) = \{\varepsilon\}$ if and only if u is the empty multiset.

The *language accepted by* Π with respect to f is defined as

$$L(\Pi, f) = \{f(u_1)f(u_2)\dots f(u_s) \in \Sigma^* \mid \ u_1u_2\dots u_s \in A(\Pi)\}.$$

Definition 4.

- $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{com-tape}(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where all the communication rules are tape rules,
- $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{all-tape}(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where all the programs must have at least one tape rule,
- $\mathcal{L}(\text{genPCol}, \mathcal{F}, *(k))$ is the class of languages accepted by generalized PCol automata with capacity k and with mappings from the class $\mathcal{F}$ where programs with any kinds of rules are allowed.

For all-tape and com-tape languages we also define their *restricted* variants, $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted all-tape}(k))$ and $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted com-tape}(k))$, respectively. These are accepted by systems with programs not having any rules of types

$$e \xrightarrow{T} e, \ a \xrightarrow{T} e, \text{ and } e \xleftrightarrow{T} e, \ e \xleftrightarrow{T} a,$$

for arbitrary $a \in V$, where e is the special environmental object. Note that systems which accept languages of these restricted classes must read nonempty multisets in each computational step.

We denote the mapping f defined for any multiset $x \in (V - \{e\})^*$ by f_{perm}, if $f(x) = \{y \in (V - \{e\})^* \mid y \in perm(x)\}$ where $perm(x) \subseteq V^*$ denotes the set of strings representing the multiset composed of the symbols of x, or in other words, $perm(x)$ is the set of strings obtained by a permutation of the symbols of the multiset x. We denote the languages of systems with this type of mapping as $\mathcal{L}_{perm}(\text{genPCol}, X(k))$, where $X \in \{\text{com-tape}, \text{all-tape}, *\}$.

Example 1. Let $M = (\Sigma, Q, \delta, q_0, F)$ be a finite automaton with input alphabet Σ, set of states Q, initial state q_0, set of final states F, and transition function $\delta : \Sigma \times Q \to Q$.

Consider the genPCol automaton $\Pi = (\Sigma \cup \{q_I\} \cup Q, e, w_E, (w, P), F')$ of capacity two, where $q_I \notin \Sigma \cup Q$, $w_E = \bigcup_{a \in \Sigma}\{a, a\}$, and $w = eq_I$. If we have the set of rules

$$P = \{\langle e \overset{T}{\leftrightarrow} a,\ q_I \to q_0\rangle \mid a \in \Sigma\} \cup$$
$$\{\langle x \overset{T}{\leftrightarrow} a,\ q \to q'\rangle \mid \text{for all } x \in \Sigma \text{ such that } \delta(x, q) = q' \text{ for some } q, q' \in Q\},$$

and set of final configurations $F' = \{(w_E - \{x\}, xq_f) \mid \text{for all } x \in \Sigma, \text{ and } q_f \in F\}$, then Π simulates the computations of M, that is, $L(\Pi) = L(M)$. To see this, note that if and only if $\delta(x, q) = q'$, then there is a transition of Π from $(w_E - \{x\}, qx) \Rightarrow (w_E - \{y\}, q'y)$ for some $y \in \Sigma$. The computation goes on until the an object $q_f \in F$ is introduced, thus, the accepted multiset sequences are sequences of singleton multisets $\{a_1\}\{a_2\}\ldots\{a_n\}$ such that $a_1a_2\ldots a_n \in L(M)$ with $a_i \in \Sigma,\ 1 \leq i \leq n$. If we have an input mapping $f(\{x\}) = x$ for any $x \in \Sigma$, then $L(\Pi, f) = L(M)$.

3 Results and Open Problems Concerning the Power of genPCol Automata

The computational capacity of genPCol automata was investigated in [5] and [6].

Proposition 1 ([5], [6]) *For any class of mappings $\mathcal{F}$, we have*

1. $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{com-tape}(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, *(k))$ *and* $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{all-tape}(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, *(k)$ *for any $k \geq 1$;*
2. $\mathcal{L}(\text{genPCol}, \mathcal{F}, \text{restricted } X(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, X(k))$ *for any $k \geq 1$ and $X \in \{\text{com-tape}, \text{all-tape}, *\}$; and*
3. $\mathcal{L}(\text{genPCol}, \mathcal{F}, X(k)) \subseteq \mathcal{L}(\text{genPCol}, \mathcal{F}, X(k+1))$ *for any $k \geq 1$ and $X \in \{\text{com-tape}, \text{all-tape}, *\}$.*

In the case of P colonies, all recursively enumerable sets of integers can be characterized by systems of capacity one, see [1]. This is also true for genPCol automata with languages obtained by permutation mappings, if programs with any kind of rules are allowed.

Theorem 2. ([6]) $\mathcal{L}_{perm}(\text{genPCol}, *(1)) = \mathcal{L}(RE)$.

The power of systems with capacity one decreases considerably if not all kinds of programs are allowed. The next theorem examines the relationship of regular languages and languages of genPCol automata with all-tape programs.

Theorem 3. ([6]) $\mathcal{L}_{perm}(\text{genPCol}, \text{all-tape}(1))$ *is incomparable with the class of regular languages.*

Concerning genPCol automata languages that can be accepted by systems of capacity two, we have the following result. r-1LOGSPACE denotes the class of languages characterized by so-called *restricted one-way logarithmic space* bounded Turing machines, see [3] for more on this complexity class.

Theorem 4. ([5])

$$\mathcal{L}_{perm}(\text{genPCol}, \text{restricted all-tape}(2)) \cup \mathcal{L}_{perm}(\text{genPCol}, \text{restricted com-tape}(2)) \subseteq \text{r-1LOGSPACE}.$$

As the class of languages characterized by P automata is strictly included in r-1LOGSPACE, the above theorem does not give any information on the relationship of the power of P automata and genPCol automata. We know, however, that genPCol automata with restricted all-tape or restricted com-tape programs are more powerful than P automata using the mapping f_{perm}.

If we denote by $\mathcal{L}_X(f_{perm}, PA)$ the class of languages characterized by P automata with $X \in \{seq, par\}$ for parallel or sequential rule application, then we have the following.

Theorem 5. ([6]) $\mathcal{L}_{perm}(\text{genPCol}, \text{restricted all-tape}(2)) \setminus \mathcal{L}_X(f_{perm}, PA) \neq \emptyset$ *for* $X \in \{seq, par\}$.

We conjecture, that the class $\mathcal{L}_{perm}(\text{genPCol}, \text{restricted all-tape}(2))$ and the class $\mathcal{L}_{perm}(\text{genPCol}, \text{restricted com-tape}(2))$ are both strictly included in the class r-1LOGSPACE, but the relationship of these classes is subject to further investigations.

If we allow arbitrary programs, then genPCol automata of capacity two characterize the class of recursively enumerable languages.

Theorem 6. ([5]) $\mathcal{L}_{perm}(\text{genPCol}, *(2)) = \mathcal{L}(\text{RE})$.

The same result holds if we require that all programs contain at least one tape rule (but unlike in the restricted case, they can also use the environmental symbol e).

Theorem 7. ([6]) $\mathcal{L}_{perm}(\text{genPCol}, \text{all-tape}(2)) = \mathcal{L}(\text{RE})$.

We do not know, however, what is the power of systems with com-tape type of programs.

If we consider systems with capacity at least three, their all-tape and com-tape languages include any recursively enumerable language. Given a recursively enumerable language L, the idea is to take a system of capacity two which, when any kind of programs are allowed, accepts L (we refer to [5] for such a system), and transform it to a system of capacity three having a communication tape rule in each program by adding "dummy" tape rules which do not interfere with the work of the rest of the system.

Proposition 8 ([6])

$$\mathcal{L}_{perm}(\text{genPCol}, X(3)) = \mathcal{L}(\text{RE})$$

for $X \in \{\text{com-tape}, \text{all-tape}\}$.

4 Conclusion

As we have seen, even with capacity one, if we do not place additional restrictions on the types of programs allowed to be used by the system, genPCol automata characterize the class of recursively enumerable languages. On the other hand, for systems with capacity three, even the use of most restrictive program types does not result in any decrease of the computational power. The most interesting cases are the ones in between these two: the restricted variants of capacity one and capacity two. These require further study; especially interesting would be to refine the relationship of the model with P automata, as they are very closely related, but not as similar as one might expect at the first glance.

Further, the effect of using *checking rules*, as defined in [8] for P colonies, is also an interesting topic for further investigations, just as the investigation of systems with other classes of input mappings besides f_{perm}. Results for other kinds of input mappings, similarly to those existing for P automata, would also be interesting.

References

1. L. Ciencialová, E. Csuhaj-Varjú, A. Kelemenová, Gy. Vaszil, Variants of P colonies with very simple cell structure. *International Journal of Computers Communication and Control*, **4** (2009), 224–233.
2. L. Cienciala, L. Ciencialová, E. Csuhaj-Varjú, Gy. Vaszil, *PCol automata: Recognizing strings with P colonies.* In: M. A. Martínez del Amor, Gh. Păun, I. Pérez Hurtado, A. Riscos Nuñez (eds.), Eighth Brainstorming Week on Membrane Computing, Sevilla, February 1-5, 2010, Fénix Editora, 2010, 65–76.
3. E. Csuhaj-Varjú, M. Oswald, Gy. Vaszil, P automata. In [10], chapter 6, 144–167.
4. 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 Worskhop, WMC-CdeA 2002, Curtea de Arges, Romania, August 19-23, 2002, Revised Papers.* LNCS 2597, Springer Berlin Heidelberg, 2003, 219–233.
5. K. Kántor and Gy. Vaszil Generalized P Colony Automata *Journal of Automata, Languages and Combinatorics* **19** (2014), 145–156.
6. K. Kántor and Gy. Vaszil On the Class of Languages Characterized by Generalized P Colony Automata *Submitted.*
7. J. Kelemen, A. Kelemenová, A grammar-theoretic treatment of multiagent systems. *Cybernetics and Systems*, **23** (1992), 621–633.
8. J. Kelemen, A. Kelemenová, Gh. Păun, Preview of P colonies: A biochemically inspired computing model. In: M. Bedau et al. (eds.), *Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX).* Boston Mass., 2004, 82–86.
9. A. Kelemenová, P Colonies. In [10], chapter 23.1, 584–593.
10. Gh. Păun, G. Rozenberg, A. Salomaa, editors, *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

A characterization of symport/antiport P systems through Information Theory ⋆ (*extended abstract*)

José M. Sempere

Departamento de Sistemas Informáticos y Computación.
Universidad Politécnica de Valencia.
email: jsempere@dsic.upv.es

Abstract. In this work we analyze communication P systems under the framework of Information Theory. Given a cell-like P system with communication and evolution rules, we analyze the amount of information that it holds as the result of symbol movements across the membranes. Hence, *antiport* rules can be defined as bidirectional information channels, while *symport* and evolution rules can be viewed as unidirectional information channels. Under this approach, we propose some results about the information of a P system and its entropy.

Keywords: Communication P systems, Information Theory, Entropy

Some basic concepts

P systems were introduced as a computational model inspired by the information and biochemical product processing of living cells through the use of membrane communication. In most of the works about P systems, information is represented as multisets of symbol/objects which can interact and evolve according to predefined rules. From the beginning, the most important component of the system has been the kind of rules that it holds. There have been different proposals to define the rules of the system such as evolution rules, communication rules, active rules to create/dissolve membrane structures, active rules with polarization, and so on and so forth.

Here, we pay our attention to the following fact: the rules of a P system produce/consume new symbols in different regions of the system, so they can be considered information regulators that act over a region, which can be considered information senders and receivers in a pure communication system. Hence, the behavior of the P system can be analyzed with Information Theory tools. Mainly, we can analyze any P system through the characterization of the entropies at every region according to its membrane structure and rules.

⋆ Work partially supported by the Spanish Ministry of Economy and Competitiveness under EXPLORA Research Project SAF2013-49788-EXP

We will introduce basic concepts related to multisets, Information Theory and P systems. We suggest to the reader the references [3, 4] to introduce membrane computing, and the books [2, 5] to introduce Information Theory. We will provide some definitions from multiset theory as exposed in [7].

Let D be a set. A multiset over D is a pair $\langle D, f\rangle$ where $f : D \longrightarrow \mathbb{N}$ is a function. If $A = \langle D, f\rangle$ and $B = \langle D, g\rangle$ are two multisets then the removal of multiset B from A, denoted by $A \ominus B$, is the multiset $C = \langle D, h\rangle$ where for all $a \in D$ $h(a) = max(f(a) - g(a), 0)$, and their sum, denoted by $A \oplus B$, is the multiset $C = \langle D, h\rangle$, where for all $a \in D$ $h(a) = f(a) + g(a)$. We will say that $A = B$ if the multiset $(A \ominus B) \oplus (B \ominus A)$ is empty that is $\forall a \in D$ $f(a) = 0$.

The size of any multiset M, denoted by $|M|$ will be the number of elements that it contains.

We can suggest to the reader the books [2, 5] and the classical work by C.E. Shannon [6] in order to have a full view of Information Theory.

An *information source* is defined by the tuple (S, P) where S is an alphabet and P a probability distribution over S. A cornerstone of Information Theory is the concept of *entropy* which is attached to information sources. The entropy of an information source I, with an alphabet S and probability distribution $P : S \to [0, 1]$ is defined as

$$H(I) = -\sum_{a \in S} P(a) \cdot \log_2 P(a)$$

Observe that we are working with trivial codes where the alphabet of an information source is its encoding. In addition, we have fixed the base 2 for the logarithm, so the information entropy is described in bits. The change from a binary base to a different one can be easily carried out in a logarithm base change.

A *cell-like* P system of degree m with *communication* rules is a construct

$$\Pi = (V, \mu, w_1, \cdots, w_m, (R_1, \rho_1), \cdots, (R_m, \rho_m), i_0),$$

where:

- V is an alphabet (the *objects*)
- μ is a membrane structure consisting of m membranes
- w_i, $1 \leq i \leq m$, is a string representing a multiset over V associated to the region i
- R_i, $1 \leq i \leq m$, is a finite set of rules of the form (u, v) with $u \neq \lambda$ and $v \neq \lambda$ (evolution rules), $(u, out; v, in)$ with $u \neq \lambda$ and $v \neq \lambda$ (antiport rule) and (x, out) or (x, in) with $x \neq \lambda$ (symport rule). The strings u, v and x are defined over the alphabet V.
- i_0 is a number between 1 and m and it specifies the *output* membrane of Π (in the case that it equals to ∞ the output is read outside the system).

Observe that, in the previous definition, we have omitted an output alphabet, a catalyst alphabet and dissolution rules. In addition, we have omitted priorities in the rule sets and other communication rules with explicit address. The main

reason is that we want to establish a preliminary analysis with the most simple systems.

The entropy of a P system

We will define the entropy of a P system by analyzing how the multisets at every region evolve according to the rules of the system. First, we define the entropy of the multisets of the regions and, then, the entropy of a P system.

Definition 1. *(**self-referred entropy of a multiset**). Let us consider a multiset $A = \langle D, f \rangle$. The self-referred entropy of A is defined as*

$$H_s(A) = -\sum_{a \in D} fr(a) \cdot log_2 fr(a)$$

where $fr(a) = \frac{f(a)}{|D|}$.

Observe that the self-referred entropy of a multiset is a static concept given that we have substitute the probability distribution by the frecuency of appearance of every object at the region ($fr(a)$). Observe that there is no external probability distribution over the rules and the objects.

In the following, we analyze the evolution of self-referred entropies according to the system computations.

Definition 2. *Let Π be a P system of degree m and $c_t = (\mu, w_1^t, \cdots, w_m^t)$ be a configuration of the system during a computation at time t. Then*

1. *The absolute entropy of Π at time t is $H_{abs}^t(\Pi) = \sum_{1 \leq i \leq m} H_s(w_i^t)$*
2. *The maximal entropy of Π at time t is $H_{max}^t(\Pi) = max\{H_s(w_1^t), \cdots, H_s(w_m^t)\}$*

The question about the computation of the entropy of a P system is completely based on the calculation of the different multisets at every region, according to the rules that affect to that region. Hence, at time t the multiset w_i^t will evolve, in the next transition, to the multiset $w_i^t \ominus l(R,i) \oplus r(R,i)$, where $l(R,i)$ is a multiset based on the left hand side of the rules that affect to the region i, and $r(R,i)$ is a multiset based on the right hand side of the rules that affect to the region i.

In this work we will overview, among others, the following questions and aspects :

1. What is the relationship between the kind of rules at every region (symport, antiport or evolution) and the evolution of its entropy ?
2. What is the definition of confluence under an information theory point of view ?
3. How does the operational mode (i.e. maximal/minimal parallelism) affect to entropy ?
4. How is the entropy defined in a stochastic/probabilistic P system ?
5. What is the definition of the entropy of a P system, if the external output is defined ?

References

1. C. Calude, Gh. Păun, G. Rozenberg and A. Salomaa, *Multiset Processing* LNCS 2235. Springer. 2001.
2. T.M. Cover and J.A. Thomas. *Elements of Information Theory* John Wiley & Sons, Inc. 1991
3. Gh. Păun. *Membrane Computing. An Introduction.* Springer. 2002.
4. Gh. Păun, G. Rozenberg and A. Salomaa (editors). *The Oxford Handbook of Membrane Computing.* Oxford University Press. 2010.
5. S. Roman. *Introduction to Coding and Information Theory.* Springer. 1997.
6. C.E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, Vol. 27, pp 379-423, 623-656, July, October, 1948.
7. A. Syropoulos. *Mathematics of Multisets.* In [1] pp 347-358.

Author Index