Eleventh Brainstorming Week on Membrane Computing

Sevilla, February 4 – 8, 2013

Luis Valencia-Cabrera, Manuel García-Quismondo
Luis F. Macías-Ramos, Miguel Á. Martínez-del-Amor
Gheorghe Păun, Agustín Riscos-Núñez
Editors

Eleventh Brainstorming Week on Membrane Computing

Sevilla, February 4 – 8, 2013

Luis Valencia-Cabrera, Manuel García-Quismondo
Luis F. Macías-Ramos, Miguel Á. Martínez-del-Amor
Gheorghe Păun, Agustín Riscos-Núñez
Editors

RGNC REPORT 1/2013

Research Group on Natural Computing
Sevilla University

Fénix Editora, Sevilla, 2013

ISBN: 978-84-940691-9-2
Edita: Fénix Editora
c/Patricio Sénz, 13 - 41004 Sevilla
info@fenixeditora.com
www.fenixeditora.com
Telf. (+34) 954 90 74 36
Impreso en España - Printed in Spain

Preface

The Eleventh Brainstorming Week on Membrane Computing (BWMC) was organized in Sevilla, from February 4 to February 8, 2013, in the organization of the Research Group on Natural Computing from the Department of Computer Science and Artificial Intelligence of Sevilla University. The first edition of BWMC was organized at the beginning of February 2003 in Rovira i Virgili University, Tarragona, and all the next editions took place in Sevilla at the beginning of February, each year.

In the style of previous meetings in this series, the eleventh BWMC was conceived as a period of active interaction among the participants, with the emphasis on exchanging ideas and cooperation. Several "provocative" talks were delivered, mainly devoted to open problems, research topics, conjectures waiting for proofs, followed by an intense cooperation among the 33 participants – see the list in the end of this preface. The efficiency of this type of meetings was again proved to be very high and the present volume illustrates this assertion.

The papers included in this volume, arranged in the alphabetic order of the authors, were collected in the form available at a short time after the brainstorming; several of them are still under elaboration. The idea is that the proceedings are a working instrument, part of the interaction started during the stay of authors in Sevilla, meant to make possible a further cooperation, this time having a written support.

Selections of the papers from these volumes will be considered for publication in special issues of *International Journal of Unconventional Computing*.

After each BWMC, one or two special issues of various international journals were published. Here is their list:

- BWMC 2003: *Natural Computing* – volume 2, number 3, 2003, and *New Generation Computing* – volume 22, number 4, 2004;
- BWMC 2004: *Journal of Universal Computer Science* – volume 10, number 5, 2004, and *Soft Computing* – volume 9, number 5, 2005;
- BWMC 2005: *International Journal of Foundations of Computer Science* – volume 17, number 1, 2006);

- BWMC 2006: *Theoretical Computer Science* – volume 372, numbers 2-3, 2007;
- BWMC 2007: *International Journal of Unconventional Computing* – volume 5, number 5, 2009;
- BWMC 2008: *Fundamenta Informaticae* – volume 87, number 1, 2008;
- BWMC 2009: *International Journal of Computers, Control and Communication* – volume 4, number 3, 2009;
- BWMC 2010: *Romanian Journal of Information Science and Technology* – volume 13, number 2, 2010;
- BWMC 2011: *International Journal of Natural Computing Research* – volume 2, numbers 2-3, 2011.
- BWMC 2012: *International Journal of Computer Mathematics* – volume 99, number 4, 2013

Other papers elaborated during the eleventh BWMC will be submitted to other journals or to suitable conferences. The reader interested in the final version of these papers is advised to check the current bibliography of membrane computing available in the domain website `http://ppage.psystems.eu`.

***

The list of participants as well as their email addresses are given below, with the aim of facilitating the further communication and interaction:

1. Ludek Cienciala, Silesian University in Opava, Czech Republic, `ludek.cienciala@fpf.slu.cz`
2. Lucie Ciencialová, Silesian University in Opava, Czech Republic, `ciecilka@gmail.com`
3. Mari Angels Colomer Cugat, University of Lleida, Spain, `colomer@matematica.udl.cat`
4. Rudolf Freund, Technological University of Vienna, Austria, `rudifreund@gmx.at`
5. Manuel García Quismondo Fernández, University of Seville, Spain, `mgarciaquismondo@us.es`
6. Adolfo Gastalver-Rubio, University of Seville, Spain, `adolaurion@hotmail.co.jp`
7. Marian Gheorghe, University of Sheffield, United Kingdom, `m.gheorghe@sheffield.ac.uk`
8. Carmen Graciani Díaz, University of Seville, Spain, `cgdiaz@us.es`
9. Miguel A. Gutiérrez Naranjo, University of Seville, Spain, `magutier@us.es`
10. Florentin Ipate, University of Bucharest, Romania, `florentin.ipate@ifsoft.ro`
11. Sergiu Ivanov, Université Paris-Est Créteil, France, `sivanov@colimite.fr`
12. Alica Kelemenová, Silesian University in Opava, Czech Republic, `alice.kelemenova@fpf.slu.cz`
13. Miroslav Langer, Silesian University in Opava, Czech Republic, `miroslav.langer@i.cz`

14. Raluca Lefticaru, University of Bucharest, Romania, raluca.lefticaru@gmail.com
15. Alberto Leporati, University of Milan – Bicocca, Italy, leporati@disco.unimib.it
16. Luis Felipe Macías Ramos, University of Seville, Spain, lfmaciasr@us.es
17. Miguel A. Martínez del Amor, University of Seville, Spain, mdelamor@us.es
18. Giancarlo Mauri, University of Milan – Bicocca, Italy, mauri@disco.unimib.it
19. Santiago Maza López de los Mozos, University of Seville, Spain, omazalopez@gmail.com
20. Niall Murphy, Universidad Politécnica de Madrid, Spain, nmurphy@gmail.com
21. David Orellana Martín, University of Seville, Spain, dorelmar@gmail.com
22. Adam Obtułowicz, Polish Academy of Sciences, Warsaw, Poland, A.Obtulowicz@impan.pl
23. Gheorghe Păun, University of Seville, Spain, and Institute of Mathematics of the Romanian Academy, gpaun@us.es
24. Ignacio Pérez Hurtado de Mendoza, University of Seville, Spain, perezh@us.es
25. Mario de J. Pérez Jiménez, University of Seville, Spain, marper@us.es
26. Antonio Enrico Porreca, University of Milan – Bicocca, Italy, porreca@disco.unimib.it
27. José Luis Pro Martín, University of Seville, Spain, jlpro@modinem.com
28. Agustín Riscos Núñez, University of Seville, Spain, ariscosn@us.es
29. Francisco José Romero Campero, University of Seville, Spain, fran@us.es
30. Álvaro Romero Jiménez, University of Seville, Spain, romero.alvaro@us.es
31. Juan Carlos Soriano Ramírez, University of Seville, Spain, juanki.soriano@gmail.com
32. Luis Valencia Cabrera, University of Seville, Spain, lvalencia@us.es
33. Claudio Zandron, University of Milan – Bicocca, Italy, zandron@disco.unimib.it

As mentioned above, the meeting was organized by the Research Group on Natural Computing from Sevilla University (http://www.gcn.us.es) and all its members were enthusiastically involved in this (not always easy) work.

The meeting was supported from various sources: (i) Ministerio de Economía y Competitividad (grants TIN2008–04487–E, TIN2009 – 13192, TIN2012 – 37434, cofinanced by FEDER funds), (ii) Junta de Andalucía (grant P08 – TIC 04200, also cofinanced by FEDER funds), (iii) Instituto de Matemáticas de la Universidad de Sevilla (IMUS), (iv) Fundación para la Investigación y el Desarrollo de las Tecnologías de la Información en Andalucía (FIDETIA), as well as by the Department of Computer Science and Artificial Intelligence from Sevilla University.

Gheorghe Păun
Mario de Jesús Pérez-Jiménez
(Sevilla, May 2013)

Contents

Flattening P Systems with Active Membranes

Bogdan Aman and Gabriel Ciobanu

Romanian Academy, Institute of Computer Science
Blvd. Carol I no.11, 700506 Iaşi, Romania
baman@iit.tuiasi.ro, gabriel@info.uaic.ro

Summary. Given a P system Π with active membranes having several membranes, we construct a P system Π^f having only one membrane and rules involving catalysts, cooperation and priorities. The evolution of this "flat" P system Π^f simulates the evolution of initial P system with active membranes Π by replacing any rule that changes the configuration in Π by prioritized rules application in a configuration of Π^f.

1 Introduction

The family of membrane systems (also called P systems) is presented in the monograph [6] and in the handbook [7], while several applications of membrane computing are presented in [4]. Membrane systems are distributed, parallel and non-deterministic computing models inspired by biological entities. The structure of the cell is represented by a set of hierarchically embedded regions, each one delimited by a surrounding boundary (called membrane), and all of them contained inside a skin membrane. A membrane without any other membrane inside it is said to be elementary, while a membrane with other membranes inside is said to be composite. Multisets of objects are distributed inside these regions, and they can be modified or communicated between adjacent compartments. Objects represent the formal counterpart of the molecular species (ions, proteins, etc.) floating inside cellular compartments, and they 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 membrane structure.

A membrane system can perform computations in the following way. Starting from an initial configuration that is defined by multisets of objects initially placed inside the compartmentalized structure and by sets of evolution rules, the system evolves by applying evolution rules in a non-deterministic and maximally parallel manner (every rule that is applicable inside a region *has to* be applied in that region). 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 (there are not used by other

rules applied in the same step). Due to the competition for resources, some rules are applied non-deterministically. A halting configuration is reached when no rule is applicable.

A flat membrane system Π^f can be constructed to simulate a membrane system Π with multiple membranes and dissolution [1]. The central idea of this construction is to use pairs of objects and labels from Π as objects in Π^f. Each rule r of Π is translated into sets of rules r^f for Π^f. This simulation is at no extra cost: the flat-membrane system does not use more time or space than the membrane system with multiple membranes.

In order to simulate a P system Π with multiple active membranes by a flat membrane P system Π^f, a bigger amount of rules and more time is needed to perform the simulation. The increase in the number of rules derives from the fact that Π has a 'higher dimension' (the changes in the topology) in working with its symbols: for instance, it can change the polarization of one compartment affecting in this way all the symbols in it. The larger amount of time depends on the fact that in Π configuration changes are done in one step (one rule application), while in Π^f this cannot be achieved in one step.

A motivation of this study is this one: many researcher classify computational complexity classes in terms of the kinds of rules present in P systems with active membranes. For instance, NP problems can be solved with these kind of systems when rules of types (a), (b), (c) and (d) or when rules of kind (a), (c), (e) and polarities are there. These kinds of rules are rather powerful and they could hide some power in their definition. We aim to unify these rules in terms of simple rules, hoping to compare their power. For instance, we may prove that the simple rules needed to simulate rules of the kind (a), (b), (c) and (d) are the same as the simple rules needed to simulate rules of the (a), (c), (e) and polarities.

2 P Systems with Active Membranes

Biological membranes are not completely passive: when a molecule passes through a membrane, the molecule and the membrane itself can be modified. These ideas lead to the introduction of P systems with active membranes where the central role in computation is played by membranes. Each membrane is supposed to have an electrical polarization (also called charge), one of the three possible: positive $(+)$, negative $(-)$ and neutral (0).

Definition 1 ([6]). *A* P system with active membranes *is a construct*

$$\Pi = (V, T, H, \mu, w_1, \ldots, w_n, \alpha_1, \ldots, \alpha_n, R)$$

where:

1. $n \geq 1$ *represents the number of membranes;*
2. V *is an alphabet (the* total alphabet *of the system);*
3. $T \subseteq V$ *(the* terminal *alphabet);*
4. H *is a finite set of* labels *for membranes;*

5. μ *is a* membrane structure*, consisting of* n *membranes, labelled in a one-to-one manner with elements of* H*;*
6. $w_1, \ldots, w_n$ *are strings over* V*, describing the* multisets of objects *placed in the* n *regions of* μ*;*
7. $\alpha_1, \ldots, \alpha_n$*, with* $\alpha_i \in \{+, -, 0\}$ *for* $i \in \{1, \ldots, n\}$*, are the initial polarizations of the membranes;*
8. R *is a finite set of* developmental rules*, of the following forms:*

   a) $[a \to v]_h^\alpha$*, for* $h \in H$*,* $\alpha \in \{+, -, 0\}$*,* $a \in V$*,* $v \in V^*$

   object evolution

   An object a *placed inside a membrane evolves into a multiset of objects* v*, depending on the label* h *and the charge* α *of the membrane. The membrane does not take part in the application of the rule and is not modified by it.*

   b) $a[\ ]_h^{\alpha_1} \to [b]_h^{\alpha_2}$*, for* $h \in H$*,* $\alpha_1, \alpha_2 \in \{+, -, 0\}$*,* $a, b \in V$

   communication

   An object a *is introduced in the membrane labelled* h *and with charge* α_1*. However, the object* a *may be modified to* b *and the polarization may be changed from* α_1 *to* α_2 *during the operation. The label of the membrane remains unchanged.*

   c) $[a]_h^{\alpha_1} \to [\ ]_h^{\alpha_2} b$*, for* $h \in H$*,* $\alpha_1, \alpha_2 \in \{+, -, 0\}$*,* $a, b \in V$

   communication

   An object a *is removed from the membrane labelled* h *and with charge* α_1*. However, the object* a *may be modified to* b *and the polarization may be changed from* α_1 *to* α_2 *during the operation. The label of the membrane remains unchanged.*

   d) $[a]_h^\alpha \to b$*, for* $h \in H$*,* $\alpha \in \{+, -, 0\}$*,* $a, b \in V$

   dissolving

   An object a *dissolves the surrounding membrane labelled* h *and with charge* α*. However, the object* a *may be modified to* b *during the operation.*

   e) $[a]_h^{\alpha_1} \to [b]_h^{\alpha_2} [c]_h^{\alpha_3}$*, for* $h \in H$*,* $\alpha_1, \alpha_2, \alpha_3 \in \{+, -, 0\}$*,* $a, b, c \in V$

   division of elementary membranes

   In reaction with an object a*, a membrane labelled* h *and with charge* α *is divided into two membranes with the same label, maybe of different polarizations. However, the object* a *may be replaced in the two new membranes by possibly new objects.*

   f) $[\ [\ ]_{h_1}^{\alpha_1} \ldots [\ ]_{h_k}^{\alpha_1} [\ ]_{h_{k+1}}^{\alpha_2} \ldots [\ ]_{h_n}^{\alpha_2}]_{h_0}^{\alpha_0} \to [\ [\ ]_{h_1}^{\alpha_3} \ldots [\ ]_{h_k}^{\alpha_3}]_{h_0}^{\alpha_5} [\ [\ ]_{h_{k+1}}^{\alpha_4} \ldots [\ ]_{h_n}^{\alpha_4}]_{h_0}^{\alpha_6}$ *for* $k \geq 1$*,* $n > k$ *,* $h_i \in H$*,* $0 \leq i \leq n$*, and* $\alpha_0, \ldots, \alpha_6 \in \{+, -, 0\}$ *with* $\{\alpha_1, \alpha_2\} = \{+, -\}$

   division of non-elementary membranes

   If a membrane labelled h_0 *contains other membranes than those with the labels* $h_1, \ldots, h_n$ *specified above, then they should have neutral charge for this rule* t *be applicable. The division of non-elementary membranes is possible only if a membrane contains two immediately lower membranes of opposite polarization,* $+$ *and* $-$*. The membranes of opposite polarizations are separated in the two new membranes, but the polarization can change. All*

membranes of opposite polarizations are separated always by applying this rule.

It can be noticed that the objects interact indirectly by means of membranes and their polarizations. The above rules are applied in the usual non-deterministic maximally parallel manner, with the following details:

- Any object can be subject of only one rule of any type and any membrane can be subject of only one rule of types (b)-(f);
- Rules of type (a) are not counted as applied to membranes, but only to objects;
- If a membrane is dissolved, then all the objects and membranes in its region are left free in the surrounding region;
- The rules are applied in a bottom-up manner;
- The skin membrane cannot be dissolved or divided, but it can be the subject of in/out operations.

3 Flattening P Systems with Active Membranes

We may reduce an initial P system Π with active membranes, as described in Section 2, to the following P system Π^f with only one membrane, priorities, catalysts and cooperative rules as described below.

$$\Pi^f = (V^f, T^f, [\,], w^f, R^f)$$

where

1. $V^f = \{a_h \mid a \in V, h \in H \cup H_{\mathrm{N}}\}$
   $\cup\, \{\alpha_h, \alpha_h^{di}, \alpha_h^m \mid \alpha \in \{+,-,0\}, h, i \in H \cup H_{\mathrm{N}}\}$
   $\cup\, \{p_{ij}, p'_{ij} \mid i, j \in H \cup H_{\mathrm{N}}\}$
   - a_h - models an object a from membrane h;
   - α_h - models the polarization α of membrane h;
   - p_{ij} - represents the fact that membrane i is included in membrane j; this object is used to model the membrane structure of the initial system;
   - $H_{\mathrm{N}} = \{h_i \mid h \in H, i \in \mathrm{N}\}$ is used to uniquely track the copies of the membranes from H created by division;
2. $T^f = \{a_h \mid a \in T, h \in H \cup H_{\mathrm{N}}\}$
   - the terminal alphabet is obtained by considering all combination of terminal objects and locations from initial system;
3. $w^f = \{a_h, a'_h \mid a \in w_h, h \in H\}$
   $\cup\, \{\alpha_h \mid h \in H\} \cup \{p_{ij} \mid \text{membrane } i \text{ is included in membrane } j \text{ in } \mu\}$
   - a_h - models an object a from the initial multiset w_h;
   - α_h - models the initial polarization α of membrane h;
- R^f is a finite set of evolution rules:
  a) a rule $[a \to v]_h^\alpha$ is simulated with the rule:

i. $\alpha_h a_h \to \alpha_h v'_h$;
v'_h denotes the fact that to all objects from the multiset v a label h is added and that the newly created objects cannot be used by any other rules in the current evolution step;

b) a rule $a[\ ]_h^{\alpha_1} \to [b]_h^{\alpha_2}$ is simulated with the rule:
   i. $p_{hi} a_i \alpha_{1h} \to p_{hi} b'_h \alpha_{2h}$;
   a_i is an object from the membrane i surrounding membrane h; this relation between membranes is captured by the object p_{hi}.

c) a rule $[a]_h^{\alpha_1} \to [\ ]_h^{\alpha_2} b$ is simulated with the rule:
   i. $p_{hi} a_h \alpha_{1h} \to p_{hi} b'_i \alpha_{2h}$;
   b_i is an object from the membrane i surrounding membrane h; this relation between membranes is captured by the object p_{hi}.

d) a rule $[a]_h^{\alpha} \to b$ is simulated with the rules:
   i. $p_{hi} a_h \alpha_h \to p'_{hi} \alpha_h^{di} b'_i$
   α_h^{di} represents a special object that models the fact that the membrane labelled h in the initial membrane structure is marked to be dissolved (d - symbolizes dissolution, i - the parent membrane of dissolved membrane h), and to signal that objects that in the initial membrane structure were in membrane h should move to membrane i. When a membrane is marked dissolution, some objects p_{ij} need to be modified in order to keep track with the modification of the initial structure. To do this the object p_{hi} is replaced with the object p'_{hi} in order to announce any children of membrane h, if any, to change the parent to i.
   ii. $p'_{hi} p_{jh} \to p'_{hi} p_{ji}$
   In the presence of the object p'_{hi} any membrane contained in the dissolved membrane h, if any, changes its parent from h to i, namely the object p_{jh} is replaced with the object p_{ji}.
   iii. $p'_{hi} \to \varepsilon$
   If there are no more membranes with parent h, the intermediate object p'_{hi} is removed.
   iv. $\alpha_h^{di} a_h \to \alpha_h^{di} a'_i$
   In the presence of the object α_h^{di} any object from the initial membrane h is moved to membrane i, namely an object a_h is replaced with an object a'_i.
   v. $\alpha_h^{di} \to \varepsilon$
   If there are no more objects in the dissolving membrane h, then intermediate object α_h^{di} is removed.

   The rules are applied according to the following sequence of priorities:
   $$(d).i > (d).ii > (d).iii > (d).iv > (d).v$$

e) a rule $[a]_h^{\alpha_1} \to [b]_h^{\alpha_2} [c]_h^{\alpha_3}$ is simulated with the rules:
   i. $p_{hi} a_h \alpha_{1h} \to p_{h_1 i} b'_{h_1} p_{h_2 i} c'_{h_2} \alpha_h^m$
   α_h^m represents a special object that models the fact that the membrane labelled h in the initial membrane structure is multiplied (m - symbolizes multiplication). In order to keep track which objects belong to

which membrane in the initial system, we consider that the new copies of the membrane and its inner objects have labels that uniquely identify them, namely h_1 and h_2. To be able to apply similar rules as for the initial membrane h also to the newly created membranes labelled by h_1 and h_2 we duplicate the rules from h in the newly created membranes by replacing the objects a_h by the objects a_{h_1} or a_{h_2}, respectively. As an example, for the rule $\alpha_h a_h \to \alpha_h v_h$ of membrane h, an instance of it is created for each membrane h_i, $i \in \{1, 2\}$, namely $\alpha_{h_i} a_{h_i} \to \alpha_{h_i} v_{h_i}$.

ii. $\alpha_h^m a_h \to \alpha_h^m a'_{h_1} a'_{h_2}$
In the presence of the object α_h^m any object from the initial membrane h is duplicated in the two new copies of membrane h, namely the object a_h is replaced with the objects a'_{h_1} and a'_{h_2}.

iii. $\alpha_h^m \to \alpha_{h_1} \alpha_{h_2}$
After all objects of membrane h are replicated into the new copies of membrane h, namely h_1 and h_2, the object α_h^m is replaced with the objects α_{h_1} and α_{h_2}, representing the polarizations of the newly created membranes.

The rules are applied according to the following sequence of priorities:

$$(e).i > (e).ii > (e).iii$$

f) a rule $[\,[\,]_{h_1}^{\alpha_1} \dots [\,]_{h_k}^{\alpha_1} [\,]_{h_{k+1}}^{\alpha_2} \dots [\,]_{h_n}^{\alpha_2}]_{h_0}^{\alpha_0} \to [\,[\,]_{h_1}^{\alpha_3} \dots [\,]_{h_k}^{\alpha_3}]_{h_0}^{\alpha_5} [\,[\,]_{h_{k+1}}^{\alpha_4} \dots [\,]_{h_n}^{\alpha_4}]_{h_0}^{\alpha_6}$ is simulated with the rules:

i. $p_{h_i h_0} +_{h_i} p_{h_j h_0} -_{h_j} \to \alpha_{0h_0}^m p_{h_i h_{01}} +'_{h_i} p_{h_j h_{02}} -'_{h_j}$
The division of the non-elementary membrane h_0 is possible only if it contains two immediately lower membranes h_i, h_j of opposite polarization, $+$ and $-$. In this case, the membranes of opposite polarizations are separated in the two new membranes h_{01} and h_{02} together with all the membranes of similar charge, but for which the polarizations can change. The membranes with neutral charge are copied in both obtained membranes. The newly created object $\alpha_{h_0}^m$ indicates that the membrane h_0 will multiply. The labels h_{01} and h_{02} are used to indicate uniquely the two instances of membrane h_0. We shall use the subscript 1 for the membrane containing the membranes initially charged with $+$, and the subscript 2 for the membrane containing the membranes initially charged with $-$.

ii. $\alpha_{h_0}^m p_{h_i h_0} +_{h_i} \to \alpha_{h_0}^m p_{h_{i1} h_{01}} +'_{h_i}$

iii. $\alpha_{h_0}^m p_{h_i h_0} -_{h_i} \to \alpha_{h_0}^m p_{h_{i2} h_{02}} -'_{h_i}$

iv. $\alpha_{h_0}^m p_{h_i h_0} 0_{h_i} \to \alpha_{h_0}^m p_{h_{i1} h_{01}} p_{h_{i2} h_{02}} 0'_{h_i}$
Using these rules, is modelled the change of the membrane structure in the initial membrane system such that the membranes with charge $+$ are included in membrane h_{01} and the label is changed from h_i to h_{i1}, while the membranes with charge $-$ are included in membrane h_{02} and the label is changed from h_i to h_{i2}. For the membranes with charge 0 a fresh copy is included both in membranes h_{01} and h_{02} and the label

h_i is changed to h_{i1} and h_{i2}. The charge is primed so no other rule can be used for the obtained membranes until the division ends.

v. $+'_{h_i} a_{h_i} \rightarrow +'_{h_i} a'_{h_{i1}}$
vi. $-'_{h_i} a_{h_i} \rightarrow -'_{h_i} a'_{h_{i2}}$
vii. $0'_{h_i} a_{h_i} \rightarrow 0'_{h_i} a'_{h_{i1}} a'_{h_{i2}}$
The objects are relocated to the new obtained membranes, and when necessary duplicated.
viii. $+'_{h_i} \rightarrow \alpha_{h_{i1}}$
ix. $-'_{h_i} \rightarrow \alpha_{h_{i2}}$
x. $0'_{h_i} \rightarrow \alpha_{h_{i1}} \alpha_{h_{i2}}$
xi. $\alpha^m_{h_0} \rightarrow \alpha_{h_{01}} \alpha_{h_{02}}$
This rules have a lower priority than the one above. When there are no more objects to be transferred to the newly obtained membranes, the primed polarizations are changed to new polarizations. If there are no primed polarizations remained, then modelling of the division process is finished by changing the label $\alpha^m_{h_0}$ to two new membranes $\alpha_{h_{01}}$ and $\alpha_{h_{02}}$. To be able to apply similar rules as for the initial membranes $h_0, h_1, \ldots, h_n$ also to the newly created membranes the rules for h_i are duplicated for the newly created membranes using instead of objects a_{hi}, objects $a_{h_{i1}}$ or $a_{h_{i2}}$, respectively. As an example, for the rule $\alpha_{h_i} a_{h_i} \rightarrow \alpha_{h_i} v_{h_i}$ of membrane h_i, an instance of it is created for each membrane h_{ij}, $j \in \{1, 2\}$ namely $\alpha_{h_{ij}} a_{h_{ij}} \rightarrow \alpha_{h_{ij}} v_{h_{ij}}$.

The rules are applied according to the following sequence of priorities:

$$(f).ii > (f).i > (f).xi$$
$$(f).iii > (f).i > (f).xi$$
$$(f).iv > (f).i > (f).xi$$
$$(f).v > (f).viii$$
$$(f).vi > (f).ix$$
$$(f).vii > (f).x$$

g) $a'_h \rightarrow a_h$
After applying all possible rules in a step, in order to mve the next step of the evolution, the primed objects (e.g., a'_h) are transformed into simple objects (e.g., a_h).

The rules are simulated using the application details for P systems with active membranes, except for rule (g) that has the lowest priority among all rules and is applied last in order to prepare the system for a new evolution step.

Remark 1. We end by emphasizing the size of the P system Π^f with respect to that of Π. The cardinality depends on the size of a halting configuration of the initial membrane system. Consider that the largest configuration has m membranes. Thus, the cardinality of the alphabet V^f is,

$$card(V^f) = m \times (card(V) + 10 \times m)$$

while the cardinality of the rule set R^f is,:

$$card(R^f) = m \times (card(R(a)) + card(R(a)) + card(R(c)) +$$

$+5 \times card(R(d)) + 3 \times card(R(e)) + 11 \times card(R(f)) + card(V^f))$,
where $R(a)$ denotes the number of (a) rules from the set of rules R.

4 Conclusion

The idea of using a flat membrane system to simulate P systems with multiple membranes has previously appeared in several papers; a formal presentation can be found in [5]. However, the P systems with multiple membranes had a static structure, not involving dissolution. The dissolution aspects where tackled in [1].

In this paper we presented a general approach for P systems with active membranes, based on the use of catalysts, cooperation and priorities.

The translation presented here can be used to simplify proofs of statements involving general P systems with active membranes by using only flat P systems. However, concerns may appear regarding the increasing number of objects and rules in the P system Π^f, according to Remark 1.

References

1. O. Agrigoroaiei, G. Ciobanu. Flattening the Transition P Systems with Dissolution. *Lecture Notes in Computer Science*, vol.6501, 53–64, 2011.
2. G. Ciobanu. *Membrane Computing and Biologically Inspired Process Calculi.* "A.I.Cuza" University Press, Iaşi, 2010.
3. G. Ciobanu. Semantics of P Systems. *Oxford Handbook of Membrane Computing*, Oxford University Press, 413–436, 2010.
4. G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez. *Applications of Membrane Computing*, Springer, Natural Computing Series, 2006.
5. R. Freund, S. Verlan. A Formal Framework for Static (Tissue) P Systems. *Lecture Notes in Computer Science*, vol.4860, 271–284, 2007.
6. Gh. Păun. *Membrane Computing. An Introduction.* Springer, 2002.
7. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.) *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.

Studying the Chlorophyll Fluorescence in Cyanobacteria with Membrane Computing Techniques

Ioan Ardelean[1], Daniel Díaz-Pernil[2], Miguel A. Gutiérrez-Naranjo[3], Francisco Peña-Cantillana[3], Iris Sarchizian[4]

[1]Institute of Biology Bucharest, Romanian Academy, 060031 Bucharest, Romania
ioan.ardelean57@yahoo.com

[2]Research Group on Computational Topology and Applied Mathematics
Department of Applied Mathematics - University of Sevilla, 41012, Spain
sbdani@us.es

[3]Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla, 41012, Spain
magutier@us.es, frapencan@gmail.com

[4]Scoala Gimnaziala nr.38 Dimitrie Cantemir, Constanta, Romania
irissarchizian@yahoo.com

Summary. In this paper, we report a pioneer study of the decrease in chlorophyll fluorescence produced by the reduction of MTT (a dimethyl thiazolyl diphenyl tetrazolium salt) monitored using an epifluorescence microscope coupled to automate image analysis in the framework of P systems. Such analysis has been performed by a family of tissue P systems working on the images as data input.

1 Introduction

Membrane Computing has many features that makes it suitable for the study and the implementation of algorithms of digital images. One of them is that, usually, these algorithms can be parallelized and locally solved. Regardless how large the picture is, many algorithmic processes can be performed in parallel in different local areas. Another interesting feature is that the local information needed for a pixel transformation can also be easily encoded in the data structures used in Membrane Computing.

Recently, a new research line has been open by applying well-known Membrane Computing techniques for solving problems from digital images. For example, segmentation is a well-known problem in the process of digital images which tries to

assign a label to every pixel in an image in such way that pixels with the same label share certain visual characteristics. Segmentation has shown its utility, for example, in bordering tumors and other pathologies or computer-guided surgery. In [11, 12, 13, 15, 31] we can find several approaches to this problem with Membrane Computing techniques. Other problems as *thresholding* [10] or *smoothing* [25] has also been considered in the framework of membrane computing. Special attention deserves [18], where the *symmetric dynamic programming stereo* (SDPS) algorithm [19] for stereo matching was implemented by using simple P modules with duplex channels or [33], where the authors combine Membrane Computing and quantum-inspired algorithms for image processing.

In [2], a first approach of the application of Membrane Computing techniques to the study of images from Microbiology was presented. Automated image analysis is increasingly used in Microbiology to quantify important parameters for research and application. The most studied so far are the cell numbers, cell volumes, frequencies of dividing cells, *in situ* classification of bacteria, enumeration of actively respiring bacteria, characterization of bacterial growth on solid medium, viability and physiological activity in biofilms (e.g. [9, 17, 29, 30]).

In [2], the focus was the study of the application of Membrane Computing techniques to the problem of counting cells. The whole process is a combination of different techniques of processing images (binarization, segmentation, noise reduction ...) which can be performed by different families of P systems. The final algorithm is a sequence of partial processes which can be performed by Membrane Computing techniques, and the application of such processes can be seen as a global machine which takes as input a digital image showing a biological entity (usually, a photograph taken with a microscopy in a *wet lab*) and the output is the number of cells in the picture.

In this paper, we focus on the problem of considering the intensity of color in cyanobacteria, a phylum of bacteria that obtain their energy through photosynthesis, by using algorithms based on Membrane Computing techniques. We report a new study of the decrease in chlorophyll fluorescence produced by the reduction of MTT (a dimethyl thiazolyl diphenyl tetrazolium salt) monitored using an epifluorescence microscope coupled to automate image analysis in the framework of P systems The different stages of the analysis have been performed by a family of tissue P systems working on the images as data input.

Such families of P systems used in the stages of the process have inspired parallel software programs which have been developed by using a device architecture called CUDA™, (Compute Unified Device Architecture). CUDA™ is a general purpose parallel computing architecture that allows the parallel NVIDIA[1] Graphics Processors Units (GPUs) to solve many complex computational problems in a more efficient way than on a CPU. GPUs constitute nowadays a solid alternative for high performance computing, and the advent of CUDA allows programmers a friendly model to accelerate a broad range of applications. This novel architecture has been previously used to implement parallel software that simulates the behav-

[1] `http://www.nvidia.com`.

ior of P systems [5, 6, 7, 8, 24, 25], and, in a similar way to other implementations, the obtained results in the problem of detecting cyanobacteria are quite promising.

The paper is organized as follows: Next, we recall the computational model used to design the different families of P systems that performs the stages of the algorithm. In Section 3, a short presentation of the biological experiment on cyanobacteria is presented. Section 4 outlines the steps of the analysis via a family of P systems with takes as input data the images taken in the wet lab. Section 5 provides some details of the implementation of such families on CUDA and finally, the paper ends with some conclusions and open lines for future research.

2 Formal Framework

Next, we recall some basics on the P system model chosen for implementing the solution described below. The model is tissue-like P systems with promoters. Promoters are usually defined on cell-like models [20] and its extension to tissue-like is quite natural. Next, we recall the formal definition.

Definition 1. *A tissue-like P system with promoters of degree $q \geq 1$ is a tuple of the form*

$$\Pi = (\Gamma, \Sigma, \mathcal{E}, w_1, \ldots, w_q, \mathcal{R}, i_{in}, i_{out})$$

where

1. *Γ is a finite alphabet, whose symbols will be called objects;*
2. *$\Sigma \subseteq \Gamma$ is the input alphabet;*
3. *$\mathcal{E} \subseteq \Gamma$ is a finite alphabet representing the set of the objects in the environment available in an arbitrary large amount of copies;*
4. *$w_1, \ldots, w_q$ are strings over Γ representing the multisets of objects associated with the cells in the initial configuration;*
5. *$\mathcal{R}$ is a finite set of rules of the following form:*

$$(pro \,|\, i, u/v, j), \quad \text{for } 0 \leq i \neq j \leq q,\ pro, u, v \in \Gamma^*$$

   In these rules, the labels $1, \ldots, q$ correspond to the q cells and the label 0 corresponds to the environment;
6. *$i_{in} \in \{1, 2, \ldots, q\}$ denotes the input region;*
7. *$i_{out} \in \{1, 2, \ldots, q\}$ denotes the output region.*

The rule $(pro \,|\, i, u/v, j)$ can be applied over two cells (or a cell and the environment) i and j such that u (contained in cell i) is traded against v (contained in cell j). The rule is applied if in i the objects of the promoter pro are present. The promoter is not modified by the application of the rule. If the promoter is empty, we will write $(i, u/v, j)$ instead of $(\emptyset \,|\, i, u/v, j)$.

Rules are used as usual in the framework of membrane computing, that is, in a maximally parallel way (a universal clock is considered). In one step, each

object in a membrane can only be used for one rule (non-deterministically chosen when there are several possibilities), but any object which can participate in a rule of any form must do it, viz., in each step we apply a maximal multiset of rules. A *configuration* is an instantaneous description of the system Π, and it is represented as a tuple $(w_0, w_1, \ldots, w_q)$, where $w_1, \ldots, w_q$, where represent the multiset of objects contained in the q cells and w_0 represent the multiset of objects from $\Gamma - \mathcal{E}$ placed in the environment (initially $w_0 = \emptyset$). Given a configuration, we can perform a computation step and obtain a new configuration by applying the rules in a parallel manner as it is shown above. A sequence of computation steps is called a *computation*. A configuration is *halting* when no rules can be applied to it.

3 Cyanobacteria

The object of study of our research are cyanobacteria. It is a phylum of bacteria that obtain their energy through photosynthesis. The ability of cyanobacteria to perform oxygenic photosynthesis is the reason why the primitive reducing atmosphere has became an oxidizing one. This new atmosphere sustained the emergence of living beings depending of oxygen, and changed the face of the Earth. It is thought that chloroplasts in plants and eukaryotic algae evolved from cyanobacterial ancestors.

Cyanobacteria are the most diversified, ecologically most successful and evolutionary most important group of prokaryotes [27] clearly defined by the ability to carry out oxygenic photosynthesis in the thylakoid membranes and respiration both in plasma membrane and thylakoid membrane [26].

Oxygenic photosynthesis, the ability to use the light energy to synthetize glucides from carbon dioxide and water, and to evolve oxygen from water molecules is essential for all the other forms of life on Earth. Historically, cyanobacteria were the first organisms to perform oxygenic photosynthesis and this metabolic ability of early cyanobacteria have converted the early reducing atmosphere of Earth (when no free molecular oxygen was available) into an oxidizing one. This process emerged approximately 3.5 billion years and had an essential effect on the evolution of life on our planet. There is a general agreement that the oxic atmosphere allowed the emergence and evolution of aerobic microorganisms, this is the occurrence of one of the greatest evolutive events on Earth, the emergence of eukaryotic cells most probably by endosymbiotic association between different types of prokaryotic cells.

The early cyanobacteria participated to this endosymbiosis thus all photosynthetic organisms on Earth have some cyanobacteria as ancestors; together with cyanobacteria (50 % contribution at planetary level) all these photosynthetic eukaryotes, including higher plants, contribute today to the synthesis of organic matter and oxygen production, the basis of all life forms here.

As an example of the importance of cyanobacteria for the life on our planet, one can remember that *Prochorococcus* -the most abundant cyanobacterium on

Earth- is responsible for 20 % of the molecular oxygen evolved (and, corresponding for 20 % of the consumed carbon dioxide and 20 % organic matter synthetized during oxygenic photosynthesis). Some cyanobacteria have also the ability to use atmospheric nitrogen as nitrogen source for growth, thus being able to live in environments where the concentrations of organic or inorganic nitrogen are very low. Cyanobacteria being very versatile microorganisms can live in very different environments for example from warmer springs to many cold sites, including glaciers. The important functions in Nature make cyanobacteria very strong candidates for the development of bio(nano)technologies the most known topics being the photoproduction of molecular hydrogen or electricity, biomass (and related processes, including valuable products) production and removal of different pollutants (petroleum hydrocarbon, heavy metals, nitrogen and phosphorus etc.,) from the environment.

The concentration of metalimnetic populations of *Planktothrix* sp. can be measured by epifluorescence microscopy of filaments collected on membrane filters. Computer image analysis is used to determine the length of filaments whose phycoerythrin fluoresces strongly in green light [32]. Similar methods have been used for enumeration of picoplanktonic cyanobacteria [1]. Image analysis was used in a previous work done on color analysis of cyanobacteria under labelling with quantum dots [3] and the cells within filamentous cyanobacteria were counted with tissue-like P systems [2].

Sarchizian *et al.* [28] investigated the ability of a cyanobacterial strain- isolate IS-H- to reduce MTT [(3-(4,5-Dimethylthiazol-2-yl)-2,5-diphenyltetrazolium bromide)], an artificial electron acceptor, with special emphasis on quantitative determinations at single cell level using automated image analysis for precise color measurement of cells within the filaments of this strain. Up to our best knowledge this is the first report on the use of automated image analysis for the measurement of reduction of artificial redox carriers at single cell level in cyanobacteria or any other levels. The results show a strong decrease in the blue signal during MTT reductions by each individual analyzed cell, as a consequence of orange light absorption by reduced MTT.

Cyanobacterial filaments (actually each filament is one biological specimen) contains chlorophyll a which has a characteristic red fluorescence. This red fluorescence can be seen using different physical instruments, as fluorescence microscopes. This fluorescence is related to the light initially absorbed by the cell. In constant experimental conditions the fluorescence as one can be seen using a fluorescence microscope is practically constant. In our experiments cyanobacterial culture were challenged with a special chemical, namely MTT. MTT (3-(4,5-Dimethylthiazol-2-yl)-2,5-diphenyltetrazolium bromide, a yellow tetrazole), a chemical belonging to tetrazolium salts that is largely used to measure the metabolic activity in living cells (see, e.g., [4] or [28]).

The rationale design of our experiments is the following: the interaction of living cyanobacteria with MTT causes the reduction of MTT with electrons coming from cyanobacterial metabolism (photosynthesis, respiration and intermedi-

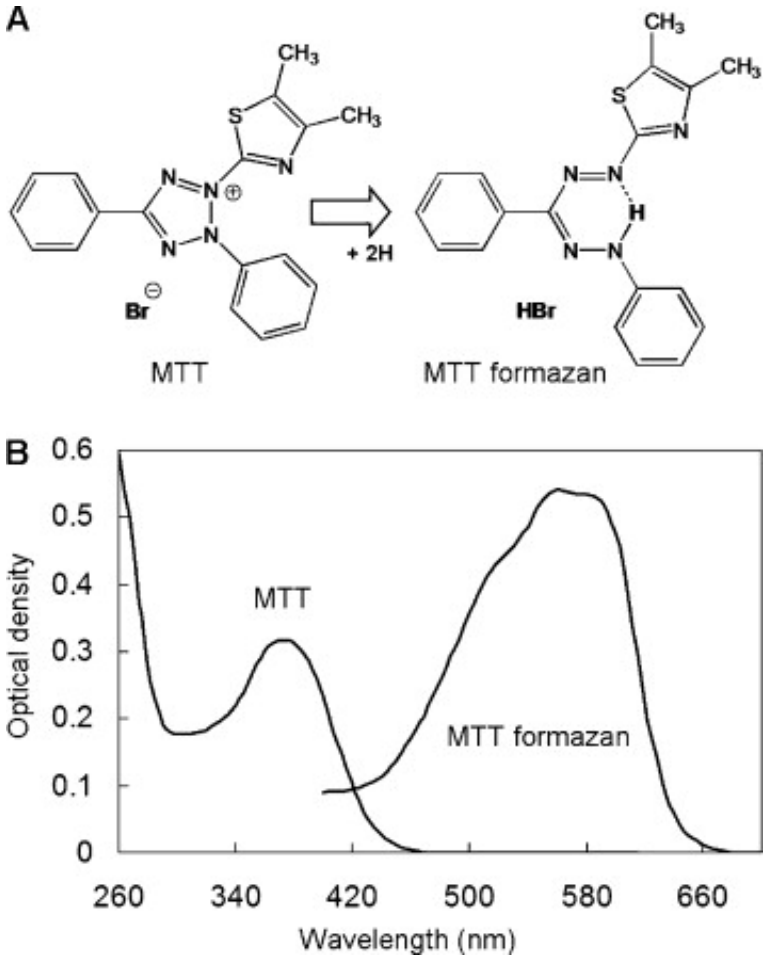


Fig. 1. The decrease of chlorophyll red fluorescence as a consequence of the accumulation inside the cell of MTT formazans crystal can be used to measure the intensity of MTT metabolic, light -dependent, reduction by cyanobacteria

ary metabolism). The reduction of MTT further decrease the intensity of chlorophyll fluorescence. The chemical reduction of MTT changes some of its properties, namely the color and the physical state of the molecule. The oxidized molecule is yellowish and water soluble whereas the reduced molecule (the so-called formazan) is dark brown, having a specific absorption spectrum, and it is insoluble in water (Fig. 1 (A)).

Fig. 1 (B) shows how MTT is reduced by enzymes called reductase (acting in photosynthesis, respiration and intermediary metabolism) to formazan. MTT is yellowish, soluble in water, and reduced MTT (MTT formazan) is insoluble (not

shown); when an appropriate chemical is added to dissolve the insoluble purple formazan product into a colored solution, this colored solution can be quantified by measuring the optical density at a certain wavelength (usually between 500 and 600 nm, as one can see, by a spectrophotometer).

Thus the reduction of MTT generate crystals of formazan which remain inside the cyanobacterial cell covering the intracellular structures of cyanobacteria. The most important intracellular structures of cyanobacteria involved in the reduction of MTT (as well as in the reduction of other artificial electron acceptors) are the thylakoides. Thylakoides are the sites where some of the light energy absorbed by thylakoids is converted in chemical stable energy found in molecules such as ATP and reduced form of chemical compounds (e.g NADPH etc.,); other part of some of the light energy absorbed by thylakoids is re-emitted as fluorescence.

During the process of MTT reduction by living cyanobacteria in light, the reduced MTT (formazan) accumulate inside the cell; this accumulation can be seen microscopically using light microscopy and even quantificated by automated image analysis [28]. This accumulation of reduced MTT inside the cell physically covers intracellular structures, including thylakoides thus acting as a shield which blocks the access of light to thylakoides, thus decreasing the intensity of chlorophyll fluorescence.

Up to our best knowledge, this is for the first time when the decrease in chlorophyll fluorescence produced by the reduction of MTT is monitored using an epifluorescence microscope. However, the inhibition of important metabolic processes in cyanobacteria during tetrazolim salt reduction is documented in literature. Paerl and Bland [23] show the effects of localized reduction of five tetrazolium salts has strong negative impact on three important metabolic processes in cyanobacteria : N_2 fixation (acetylene reduction), CO_2 fixation, and H_2 consumption.

During short-term (within 30 min) exposures in the cyanobacterium *A. oscillarioides*, salt reduction in heterocysts occurred simultaneously with inhibition of acetylene reduction

Conversely, when salts failed to either penetrate or be reduced in heterocysts, no inhibition of acetylene reduction occurred. When salts were rapidly reduced in vegetative cells, $^{14}CO_2$ fixation and $^{3}H_2$ utilization rates decreased [23] .

The type of experiment presented in this paper has a deeper biological theoretical significance and a stronger practical application than our previous work done on color analysis of cyanobacteria under labelling with quantum dots [3] because of metabolic background. The decrease of chlorophyll fluorescence as a consequence of the accumulation inside the cell of MTT formazans crystal can be used to (indirectly) measure the intensity of MTT reduction at the level of filaments or even at the level of individual cells within each filament, cells which are subcomponents of the biological individual (the filament in the case of filamentous cyanobacteria).

4 Analyzing the Images

The study of the chlorophyll fluorescence in cyanobacteria has been split in several stages. In each stage, an image is provided as input and it is processed by a tissue P system with promoters described above. The result is an automatized image process performed by a sequence of P systems.

The target is to obtain information about the central cyanobacteria of the image of Fig. 2 (a). To do that, the following stages are performed:

Stage 1: Grey Scale. The image is transformed into a grey scale one (Fig. 2 (b)). We only keep the information on the red plane to do this.

Stage 2: Sampling (Fig. 3 (a)). Before being processed by a computer, the images greater than an specific size must be sampled. The aim of this is a basic process is to obtain images of the same size before comparing them.

Stage 3: Dynamical AGP Segmentator (First threshold, Fig 3 (b)). This is an iterative stage. We apply, in each iteration, a variant of the AGP segmentator (See [14]). In order to blur the image, each pixel on the boundary turns on white or it takes the smallest gray value of their neighbours. As usual in P systems, this process finishes when no more segmentation rules can be applied.

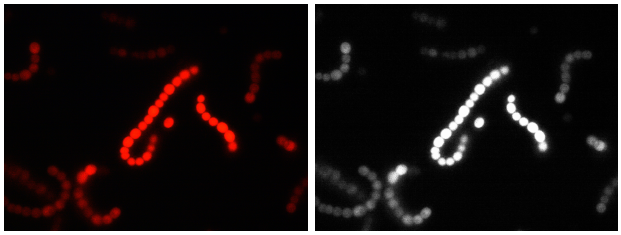
Stage 4: Iterative Edge Erosion Fig 3 (c). In each iteration, rules of the following type are applied:

$$\left(\begin{array}{ccc} K_1 & K_2 & K_3 \\ 1, K_8 & B & K_4 \\ K_7 & K_6 & K_5 \end{array} \quad / \quad \begin{array}{ccc} K_1 & K_2 & K_3 \\ K_8 & K' & K_4 \\ K_7 & K_6 & K_5 \end{array} , 0 \right)$$

where $K' = min\{K_i : i = 1, \ldots, 8 \wedge K_i \neq B\}$

Stage 5: Segmentation (Second threshold, Fig 4 (a)). Again, the P system implementation of the Sobel segmentator is used, but in this stage it is combined with the AGP segmentator [14] in order to obtain a sharper definition of the boundary.

Stage 6: Quantization (Third threshold, Fig 4 (b)) Quantization is a lossy compression technique achieved by compressing a range of values to a single quantum value. In our study we apply the tissue P system implementation presented in



(a) (b)

Fig. 2. (a) An example image (b) Grey Scale Stage

(a) (b) (c)

Fig. 3. (a) Sampling Stage (b) Dynamical AGP Segmentator Stage (c) Iterative Edge Erosion Stage

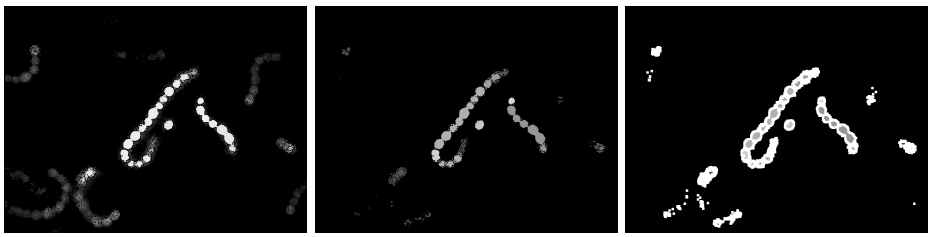
[24]. So, we propagate the background, black pixels, between the dark areas (with color early to black).

Stage 7: Spot Erasing: In this stage, two copies of the image provided as output of the previous stage are taken as input. This new stage is split into two steps:

- *White Marking:* Rules of type $(1, K\,N/B\,N, 0)$ are applied. Eight rules are defined, one for each neighbour of the pixel (K). (Fig 4 (c)).
- *White Unmarking:* Rules of the same type as the used ones in the iterative edge erosion perform this step, but in this case, the neighbours are taken from the second copy of the image. (Fig 5 (a)).

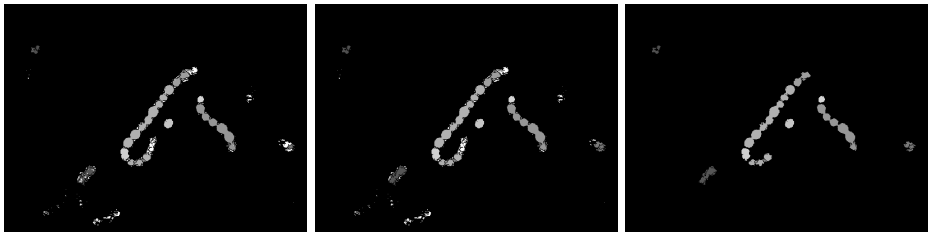
Stage 8: Edge Erasing

- *Edge Erasing type 1:* The white edges adjacent to pixels with a associated color different to black are deleted. Rules as the ones in the iterative edge erosion stage are used (Fig 5 (b)).
- *Edge Erasing type 2:* Edges are eliminated by using the same type of rules of the iterative edge erosion stage. (Fig 5 (c)).



(a) (b) (c)

Fig. 4. (a) Segmentation Stage (b) Quantization Stage (c) Spot Erasing Stage: White Marking



(a) (b) (c)

Fig. 5. (a) Spot Erasing Stage: White Unmarking (b) Edge Erasing Stage: type 1 (c) Edge Erasing Stage: type 2

Stage 9: Dealing with Connected Components. Labelled connected components are seeked (Fig 6). In this case, two pixels are considered *connected* when their distance is less than 3. See [16] for a detailed description of a family of tissue P systems for finding connected components in a binary image.

After finishing all the stages of the previous algorithm, some information of each connected component, i.e., each cyanobacteria, like the area, medium intensity, etc. can be obtained, and, of course, the number of studied Cyanobacteria too.

For example, the algorithm can be applied to study the medium intensity in two images where the same Cyanobacteria appears with a little difference of time (See Fig 7 (a) and (b)). After applying the algorithm, the images in Fig 7 (c) and (d) are obtained, where the different connected components are shown.

Finally, some statistical information is obtained. On one hand, for the first image, we have detected 6 connected components. We have kept the greatest of them (the chosen cyanobacterium for the study) whose area has 6375 pixels and a size of 1570244. Moreover, its average intensity is $246,31$. On the other hand, the image taken in the second place has the same 6 connected components. The greater

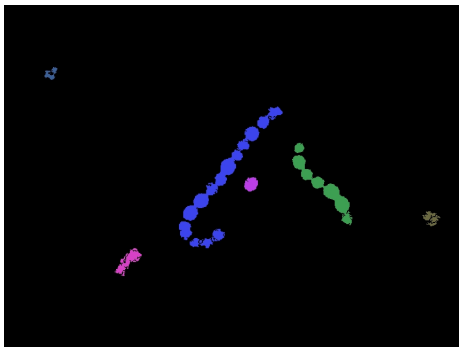
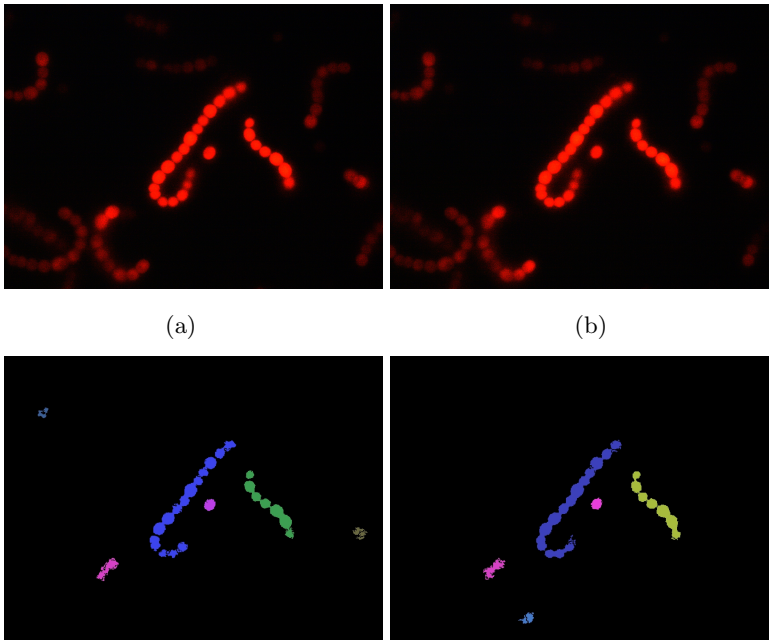


Fig. 6. Dealing with Connected Components Stage



(a) (b)

(c) (d)

Fig. 7. (a) Original Images of the previous example (b) Image taken an briefly later instant (c) Different Cyanobacteria detected by our software in the first image (d) Different Cyanobacteria detected by our software in the second image

of them (our cyanobacterium) has a greater area with respect to the previous. In this case it has 7227 pixels and a size of 1784567 and the average intensity is similar or lightly greater to the cyanobacteria of the previous image. In this case, it is $246,93$.

5 Implementation

Inspired in the families of tissue-like P systems that perform the stages of the process of counting cells, a software tool has been implemented by using CUDA™, (Compute Unified Device Architecture) [21, 22]. CUDA™ is a general purpose parallel computing architecture that allows the parallel NVIDIA Graphics Processors Units (GPUs) to solve many complex computational problems in a more efficient way than on a CPU.

The experiments have been performed on a computer with a CPU AMD Athlon II x4 645, which allows to work with four cores of 64 bits to 3.1 GHz. The computer has four blocks of 512KB of L2 cache memory and 4 GB DDR3 to 1600 MHz of

main memory. The used graphical card (GPU) is an NVIDIA Geforce GT240 composed by 12 *Stream Processors* with a total of 96 cores to 1340 MHz. It has 1 GB DDR3 main memory in a 128 bits bus to 700 MHz. So, the transfer rate obtained is by 54.4 Gbps. The used Constant Memory is 64 KB and the Shared Memory is 16 KB. Its Compute Capability level is 1.2 (from 1.0 to 2.1). The implementation deals with N blocks of threads for the complete image in our GPU of 96 cores.

6 Conclusions

The discovery of new application areas of Membrane Computing is a powerful engine for future research. In parallel, the new hardware architectures, as CUDA, allows a real implementation of the inherent parallelism of P systems. In this paper, we report a new step in the applications of Membrane Computing techniques to Digital Images. As pointed above, Membrane Computing techniques allow a natural treatment of the parallelism of the flow of information in Digital Images algorithms where the information can be encoded with simple data structures.

From a practical point of view, such techniques are a real innovation in the study of biological images. In this paper, the case study has been the chlorophyll fluorescence in cyanobacteria and its use for computing their density. A deep study of these cyanobacteria can contribute for the development of future bio(nano)technologies as the production of electricity or pollutants removal.

In near future we intend to simultaneously measure on the same sample both the formation of MTT formazan (using bright field microscopy, as in [28]) and the decrease in chlorophyll fluorescence (using epifluorescence microscopy, as in this report) and to quantitatively analyse the correlation between the two processes.

Acknowledgements

MAGN acknowledges the support of the Project of Excellence with *Investigador de Reconocida Valía* of the Junta de Andalucía, grant P08-TIC-04200 and the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain, both cofinaced by FEDER funds.

References

1. Albertano, P., Somma, D.D., Capucci, E.: Cyanobacterial picoplankton from the central baltic sea: cell size classification by image-analyzed fluorescence microscopy. Journal of Plankton Research 19(10), 1405–1416 (1997)
2. Ardelean, I., Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Peña-Cantillana, F., Reina-Molina, R., Sarchizian, I.: Counting cells with tissue-like P systems. In: Martínez-del-Amor, M.A., Păun, G., Pérez-Hurtado, I., Romero-Campero, F.J. (eds.) Tenth

Brainstorming Week on Membrane Computing. vol. I, pp. 69–78. Fénix Editora, Sevilla, Spain (2012)
3. Armăşelu, A., Popescu, A., Apostol, I., Ardellean, I., Damian, V., Iordache, I., Sarchizian, I., Apostol, D.: Passive nonspecific labeling of cyanobacteria in natural samples using quantum dots. Optoelectronics and Advanced Materials - Rapid Communications 5(10), 1084 – 1090 (October 2011)
4. Berridge, M.V., Herst, P.M., Tan, A.S.: Tetrazolium dyes as tools in cell biology: New insights into their cellular reduction. Biotechnology Annual Review 11, 127 – 152 (2005)
5. Carnero, J., Díaz-Pernil, D., Gutiérrez-Naranjo, M.A.: Designing tissue-like P systems for image segmentation on parallel architectures. In: Martínez-del-Amor, M.A., Păun, G., Pérez-Hurtado, I., Romero-Campero, F.J., Cabrera, L.V. (eds.) Ninth Brainstorming Week on Membrane Computing. pp. 43–62. Fénix Editora, Sevilla, Spain (2011)
6. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Implementing P systems parallelism by means of GPUs. In: Păun, G., Pérez-Jiménez, M.J., Riscos-Núñez, A., Rozenberg, G., Salomaa, A. (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5957, pp. 227–241. Springer, Berlin Heidelberg (2009)
7. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. Journal of Logic and Algebraic Programming 79(6), 317–325 (2010)
8. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with active membranes on CUDA. Briefings in Bioinformatics 11(3), 313–322 (2010)
9. Chávez de Paz, L.E.: Image analysis software based on color segmentation for characterization of viability and physiological activity of biofilms. Applied and Environmental Microbiology 75(6), 1734–9 (2009)
10. Christinal, H.A., Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Pérez-Jiménez, M.J.: Thresholding of 2D images with cell-like P systems. Romanian Journal of Information Science and Technology (ROMJIST) 13(2), 131–140 (2010)
11. Christinal, H.A., Díaz-Pernil, D., Real, P.: Segmentation in 2D and 3D image using tissue-like P system. In: Bayro-Corrochano, E., Eklundh, J.O. (eds.) Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications 14th Iberoamerican Conference on Pattern Recognition, CIARP 2009, Guadalajara, Jalisco, Mexico, November 15-18, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5856, pp. 169–176. Springer, Berlin Heidelberg (2009)
12. Christinal, H.A., Díaz-Pernil, D., Real, P.: Region-based segmentation of 2D and 3D images with tissue-like P systems. Pattern Recognition Letters 32(16), 2206 – 2212 (2011)
13. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Molina-Abril, H., Real, P.: A bio-inspired software for segmenting digital images. In: Nagar, A.K., Thamburaj, R., Li, K., Tang, Z., Li, R. (eds.) Proceedings of the 2010 IEEE Fifth International Conference on Bio-Inspired Computing: Theories and Applications BIC-TA. vol. 2, pp. 1377 – 1381. IEEE Computer Society, Beijing, China (2010)
14. Díaz-Pernil, D., Berciano, A., Peña-Cantillana, F., Gutiérrez Naranjo, M.A.: Segmenting Images with Gradient-based Edge Detection Using Membrane Computing. Pattern Recognition Letters 34(8), 846–855 (2013)

15. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Molina-Abril, H., Real, P.: Designing a new software tool for digital imagery based on P systems. Natural Computing pp. 1–6 (2011)
16. Díaz-Pernil, D., Gutiérrez-Naranjo, M.A., Real, P., Sánchez-Canales, V.: Computing homology groups in binary 2D imagery by tissue-like P systems. Romanian Journal of Information Science and Technology 13(2), 141–152 (2010)
17. Fero, M., Pogliano, K.: Automated quantitative live cell fluorescence microscopy. Cold Spring Harb Perspectives in Biology 2(8), a000455 (2010)
18. Gimel'farb, G., Nicolescu, R., Ragavan, S.: P systems in stereo matching. In: Real, P., Díaz-Pernil, D., Molina-Abril, H., Berciano, A., Kropatsch, W. (eds.) Computer Analysis of Images and Patterns, Lecture Notes in Computer Science, vol. 6855, pp. 285–292. Springer Berlin / Heidelberg (2011)
19. Gimel'farb, G.L.: Probabilistic regularisation and symmetry in binocular dynamic programming stereo. Pattern Recognition Letters 23(4), 431–442 (2002)
20. Ionescu, M., Sburlan, D.: On P systems with promoters/inhibitors. Journal of Universal Computer Science 10(5), 581–599 (may 2004)
21. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue 6, 40–53 (March 2008)
22. Owens, J.D., Houston, M., Luebke, D., Green, S., Stone, J.E., Phillips, J.C.: GPU Computing. Proceedings of the IEEE 96(5), 879–899 (May 2008)
23. Paerl, H.W., Bland, P.T.: Localized tetrazolium reduction in relation to N_2 fixation, CO_2 fixation, and H_2 uptake in aquatic filamentous cyanobacteria. Applied and Environmental Microbiology 43(1), 218–226 (January 1982)
24. Peña-Cantillana, F., Díaz-Pernil, D., Berciano, A., Gutiérrez-Naranjo, M.A.: A parallel implementation of the thresholding problem by using tissue-like P systems. In: Real, P., Díaz-Pernil, D., Molina-Abril, H., Berciano, A., Kropatsch, W.G. (eds.) CAIP (2). Lecture Notes in Computer Science, vol. 6855, pp. 277–284. Springer (2011)
25. Peña-Cantillana, F., Díaz-Pernil, D., Christinal, H.A., Gutiérrez-Naranjo, M.A.: Implementation on CUDA of the smoothing problem with tissue-like P systems. International Journal of Natural Computing Research 2(3), 25–34 (2011)
26. Peschek, G.A.: Structure-function relationships in the dual-function photosynthetic-respiratory electron-transport assembly of cyanobacteria (blue-green algae). Biochemical Society Transactions 24(3), 729–733 (August 1996)
27. Peschek, G.A., Obinger, C., Fromwald, S., Bergman, B.: Correlation between immuno-gold labels and activities of the cytochrome-c oxidase (aa_3-type) in membranes of salt stressed cyanobactria. FEMS Microbiology Letters 124(3), 431–437 (1994)
28. Sarchizian, I., Cîrnu, M., Ardelean, I.I.: Isolation of a heterocyts - forming cyanobacterium and quantification of its biotechnological potential with respect to redox properties at single cell level. Romanian Biotechnological Letters 16(6), 3–9 (2011)
29. Selinummi, J., Ruusuvuori, P., Podolsky, I., Ozinsky, A., Gold, E., Yli-Harja, O., Aderem, A., Shmulevich, I.: Bright field microscopy as an alternative to whole cell fluorescence in automated analysis of macrophage images. PLoS ONE 4(10), 1 9 (2009)
30. Selinummi, J., Seppälä, J., Yli-Harja, O., Puhakka, J.A.: Software for quantification of labeled bacteria from digital microscope images by automated image analysis. Biotechniques 39(6), 859–863 (2005)

31. Sheeba, F., Thamburaj, R., Nagar, A.K., Mammen, J.J.: Segmentation of peripheral blood smear images using tissue-like P systems. Sixth International Conference on Bio-Inspired Computing: Theories and Applications (BIC-TA), 2011, 257–261 (sept 2011)
32. Walsby, A.E., Avery, A.: Measurement of filamentous cyanobacteria by image analysis. Journal of Microbiological Methods 26(1-2), 11 – 20 (1996)
33. Zhang, G.X., Gheorghe, M., Li, Y.: A membrane algorithm with quantum-inspired subalgorithms and its application to image processing. Natural Computing 11(4), 701–717 (2012)

A GPU Simulation for Evolution-Communication P Systems with Energy Having no Antiport Rules

Zylynn F. Bangalan[1], Krizia Ann N. Soriano[1], Richelle Ann B. Juayong[1], Francis George C. Cabarle[1], Henry N. Adorna[1], Miguel A. Martínez–del–Amor[2]

[1] Algorithms & Complexity Lab
Department of Computer Science
University of the Philippines Diliman
Diliman 1101 Quezon City, Philippines
E-mail: zfbangalan@up.edu.ph, knsoriano1@up.edu.ph, rbjuayong@up.edu.ph, fccabarle@up.edu.ph, hnadorna@dcs.upd.edu.ph

[2] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: mdelamor@us.es

Summary. Evolution-Communication P system with energy (ECPe systems) is a cell-like variant P system which establishes a dependence between evolution and communication through special objects, called 'energy,' produced during evolution and utilized during communication. This paper presents our initial progress and efforts on the implementation and simulation of ECPe systems using Graphics Processing Units (GPUs). Our implementation uses matrix representation and operations presented in a previous work. Specifically, an implementation of computations on ECPe systems without antiport rules is discussed.

1 Introduction

Evolution-Communication P system with energy (ECPe systems) is a modification to Evolution-Communication P system (ECP system). Objects evolve through *evolution rules* while communication of objects to other regions bounded by membranes is done through *symport/antiport rules*. The unique features of an ECPe system not present for Evolution-Communication P system are listed below.

- A communication rule must require at least one quantum of energy, referred to as e, to be triggered.
- This quantum of energy, e, is produced by evolution rules and consumed by communication rules only i.e. it can never be present at the initial configuration.

ECPe systems are actually presented to provide a measure for communication over Evolution-Communication P systems [1].

Other important characteristics of ECPe systems which are common to other variants of P systems are also worth mentioning. Objects in each membrane are considered as multisets since there can be multiple instances and type of objects present within membranes. Evolution rules and communication rules within each membrane are applied in a *nondeterministic maximally parallel manner*. Maximal parallelism ensures that all rules that can be applied must be applied given the multiset of objects in each membrane while nondeterministic application of rules arises because it is possible that more than one rule is applicable at the same time. Further definition and discussion about ECPe systems is found in section 2.

Computations in ECPe system have been represented using vectors, matrices and linear algebra in [6]. As suggested in [8], we can make these vectors and matrix representations local to regions to avoid dealing with sparse vectors and matrices and to make computations in the system suitable for parallel processing.

Many works have been made for simulating P systems. Efforts for simulation is motivated by the fact that simulations help in the analysis of P systems. Since P systems are highly parallel in nature, many of the works are focused on its parallel implementation. One example is the parallel implementation of Transition P systems on a cluster of computers presented in [5]. Another is that in [3] wherein Spiking Neural P systems are simulated in parallel using GPUs. Though there are already a number of parallel implementations for P systems, none of these is directly applicable to ECPe systems. We are interested in a parallel implementation of ECPe systems in GPUs to contribute on researches implementing cell-like variant of P systems on GPUs. We also aim to spark interest and aid in further researchers in the field of Membrane computing, particularly ECPe systems, and parallel computing paradigms in general.

Just like the work in [3], we will make use of NVIDIA GPUs for this parallel simulation and implementation. NVIDIA, a manufacturer of GPUs, introduced *Compute Unified Device Architecture* (CUDA). CUDA is a software and hardware architecture that permits programmers to have a control over NVIDIA's GPU hardware and use them for parallel computations.

Our parallel implementation of ECPe systems is a continuation on an earlier study [8]. Algorithms for the methodology presented in [8] are developed and are implemented in CUDA C. It is important to emphasize that the implementation done only involves methodology for forward computing of Evolution Communication P system without antiport rules only

2 ECPe system

2.1 Definitions and notations

ECPe system is introduced in [1] as a model where *special objects* are used to establish dependence of communication on evolution. The goal of [1] is mainly to

initiate communication complexity analysis for P systems. In order to evaluate *communication* complexity for computations in ECPe systems, a cost of using a communication rule is considered. This cost is in the form of a quantity of "energy". A single object can be transported by a communication rule with the help of one or more quantum of energy, e. This quantum of energy is a special object, $e \notin O$, which can be produced by evolution rules and consumed through communication rules only. No communication rules can be applied without consuming an amount of "energy". When objects are transported, the quanta of energy consumed are lost. They do not pass across membranes.

Following the definition in [1], an EC P System with energy is a construct of the form

$$\Pi = (O, e, \mu, \omega_1, ..., \omega_m, R_1, R'_1, ..., R_m, R'_m, i_{out})$$

where:

1. O is the alphabet of objects;
2. m is the total number of membranes;
3. μ is the membrane structure
   Membrane i is called the *parent membrane* of a membrane j, denoted $parent(j)$, if the paired-labelled square brackets corresponding to membrane j is inside the paired-labelled square brackets corresponding to membrane i, i.e., $[_i \ldots [_j \,]_j]_i$. Conversely, membrane j is called a *child membrane* of membrane i, denoted $j \in children(i)$ where $children(i)$ is referring to the set of membranes inside membrane i.
4. $\omega_1, ..., \omega_m$ are multisets of objects present in the regions bounded by membranes;
5. $R_1, ..., R_m$ are sets of evolution rules, each associated with a region delimited by a membrane;
   An evolution rule is of the form $a \to v$ where $a \in O$, $v \in (O \cup \{e\})^*$ i.e e should never be in the initial configuration and cannot be evolved.
6. $R'_1, ..., R'_m$ are sets of symport/antiport rules each associated with a membrane;
   A symport rule is of the form (ae^i, in) or (ae^i, out), where $a \in O$, $i \geq 1$. The number i is called the *energy* of the rule. An antiport rule is of the form $(ae^i, out; be^j, in)$ where $i, j \geq 1$. The number $i + j$ is called the *energy* of the rule.
7. $i_{out} \in \{0, 1, ..., m\}$ is the output region where i_0 is the environment;

As in classical cell-like variants, rules must be applied in a nondeterministic and maximally parallel manner. A configuration at any time i is denoted by C_i while a transition from C_i to C_{i+1} through nondeterministic and maximally parallel manner of rule application can be denoted as $C_i \Rightarrow C_{i+1}$. A series of transition is said to be a computation and can be denoted as $C_i \Rightarrow^* C_j$ where $i < j$. Computation succeeds when the system halts; this occurs when the system reaches a configuration wherein none of the rules can be applied. This configuration is called a halting configuration. If the system doesn't halt, this implication failure of computation because the system did not produce any output.

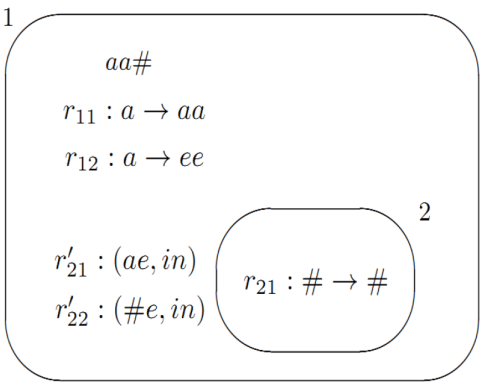


Fig. 1. An ECPe system with a construct of $\Pi = (\{a, \#, e\}, e, [_1[_2]_2]_1, \omega_1, \omega_2, R_1, R_2, R'_1, R'_2, 2)$ where $\omega_1 = \{a^2, \#\}$, $\omega_2 = \emptyset$, $R_1 = \{r_{11} : a \to aa, r_{12} : a \to ee\}$, $R_2 = \{\# \to \#\}$, $R'_1 = \emptyset$, $R'_2 = \{r'_{21}(ae, in), r'_{22} : (\#e, in)\}$

2.2 Matrix Representations

In [6], a matrix representation for ECPe system is obtained. With this representation, matrix operations can be used to model computations in ECPe system. The work is motivated by the fact that an algebraic representation helps in simulating P system. An implementation can help in easier or faster analysis of ECPe system. To obtain a matrix representation, a total order over the objects and over the rules are defined so that the elements can be identified by their positions. The following are also defined:

- Configuration vector $\mathbf{C}_i$ is a vector with length $|(O \cup \{e\}) \times \{1, ..., m\}|$ and whose elements are numbers representing the multiplicity of objects in each region at time i.
- Application vector $\mathbf{a}_i$ is a vector with length $|\bigcup_{1 \le k \le m} R_k \cup R'_k|$ and whose elements are the number of times each rule is applied during a transition $C_{i-1} \Rightarrow C_i$.
- Transition matrix M_Π is an n by r matrix ($n = |\bigcup_{1 \le k \le m} R_k \cup R'_k|$ and $r = |(O \cup \{e\}) \times \{1, ..., m\}|$) that shows the effect of the application of each rule. The matrix $M_\Pi(r, (\alpha, k))$ gives the number of consumed or produced object α in region k when the rule r is applied once. Elements with negative value represent the number of objects consumed or moved out of a given region while elements with positive value represent the number of objects that will be produced or moved in a given region. Elements with zero value represent the objects that are either not involved in the rule or involved but the total effect of their production and consumption is zero.
- Trigger matrix is an $r \times n$ matrix that represents all the needed objects in order to activate the rule. The elements are the number of objects required for a rule to be applied.

(Formal definitions of the vectors and matrices above are presented in [6]).

Forward computing is the process of finding all possible next configuration given an input configuration. Equation 1 shows that given a configuration vector for a certain time $\mathbf{C}_{k-1}$, transition matrix M_{Π} and an application vector for a transition $C_{k-1} \Rightarrow C_k$, the next configuration vector $\mathbf{C}_k$ can be computed:

$$(\mathbf{C}_k = \mathbf{C}_{k-1} + \mathbf{a}_k \cdot M_{\Pi}) \tag{1}$$

The equation above implies that in order to compute all next configurations for forward computing, there is a need to find all valid application vectors $\mathbf{a}_k$. [7] shows that this problem can be reduced into solving a system of linear equations.

2.3 Localization of Computations

In [7], the localization of rules in ECPe system are taken into consideration, dividing equation (1) into multiple equations (one for every local region) making them suitable for parallel processing. Localization also provides a hint to unreachability based on rules and initial multiset of objects. The following notations are defined over an ECPe system Π without antiport rules.

- Let $IO(r, k)$ be the set of objects in region k involved in a rule r.
- Let $TO(r, k)$ be the set of objects in region k that trigger a rule r.

The following definitions and theorems taken from [7].

Definition 1 ***Involved Rules in Region*** k

$$IR(k) = R_k \cup R'_k \cup (\bigcup_{k' \in children(k)} R'_{k'})$$

Definition 2 ***Possible Objects in Region*** k

$$PO(k) = \{\alpha | \alpha \textit{ appeared in } w_k\} \cup (\bigcup_{r \in IR(k)} IO(r, k))$$

Definition 3 ***Effect Rules in Region*** k

$$ER(k) = \{r | r \in R_k\} \cup (IR(k) - \{r' | r' \in IR(k) \textit{ and } TO(r', k) \neq \emptyset\})$$

Definition 4 ***Trigger Rules in Region*** k

$$TR(k) = \{r | TO(r, k) \neq \emptyset\}$$

Definition 5 ***Configuration Vector for each Region*** k
A configuration vector $\mathbf{C}_{i,k}$ *is a vector whose length is* $|PO(k)|$. *The vector* $\mathbf{C}_{i,k}(\alpha)$ *refers to the multiplicity of object* α *in region* k *at configuration* C_i.

Definition 6 ***Application Vector for each Region*** k
An application vector $\mathbf{a}_{i,k}$ *is a vector whose length is* $|IR(k)|$. *The vector* $\mathbf{a}_{i,k}(r)$ *refers to the number of application of rule* r *specifically in region* k *during the transition* $C_{i-1} \Rightarrow C_i$.

Definition 7 ***Transition Matrix for each Region*** k
A transition matrix $M_{\Pi,k}$ is a matrix whose dimension is $|IR(k)| \times |PO(k)|$. The matrix $M_{\Pi,k}(r, \alpha)$ returns the number of consumed or produced object α in region k upon single application of rule r. The consumed objects have negative values while the produced objects are positive. If object α in region k is not used in rule r, then its value is zero.

Theorem 1 *(from [7]) The effect of Equation (1) is the same as the effect of performing*

$$\mathbf{C}_{i,k} = \mathbf{C}_{i-1,k} + \mathbf{a}_{i,k} \cdot M_{\Pi,k} \tag{2}$$

for each region k provided that if k and $k^{'}$ are the sender and receiver regions corresponding to a communication rule $r^{'} \in IR(k) \cap IR(k')$, then $\mathbf{a}_{i,k}(r^{'}) = \mathbf{a}_{i,k'}(r^{'})$.

Corollary 1 *(from [7]) The formula for computing backward is*

$$\mathbf{C}_{i-1,k} = \mathbf{C}_{i,k} - \mathbf{a}_{i,k} \cdot M_{\Pi,k} \tag{3}$$

for each region k provided that if k and $k^{'}$ are the sender and receiver regions corresponding to a communication rule $r^{'} \in IR(k) \cap IR(k')$, then $\mathbf{a}_{i,k}(r^{'}) = \mathbf{a}_{i,k'}(r^{'})$.

The above definitions and theorems are used to make vectors and matrices local to regions to exploit independence between regions for parallel computations.

2.4 Methodology for Forward Computing

Given $\mathbf{C}_{i,k}$, we determine $\mathbf{C}_{i+1,k}$ by forward computing using the methodology presented in [8].
1. Categorize all possible objects in $PO(k)$ for all region k. First, categorize all $\alpha \in PO(k)$ for a certain region k. The categories are:

- Category 1: Evolution Trigger
  Object α is in this category if there exists $r \in R_k$ such that $TO(r, k) = \{\alpha\}$.
- Category 2: Communication Trigger Only
  Object α belongs in this category if there does not exist $r \in R_k$ such that $TO(r, k) = \{\alpha\}$ but there exists $r' \in IR(k)$ such that $\alpha \in TO(r', k)$.
- Category 3: Not a Trigger
  Object α is not in Category 1 and is not in Category 2.

2. Construct identity rules for objects in Category 2 and 3 for all region k. For each $\alpha \in PO(k)$ that belongs to one of Category 2 and Category 3, include an identity rule $\alpha \rightarrow \alpha$. Place all these rules in a set labelled $R_{add,k}$. Also, keep a list of $\alpha' \in PO(k) - \{e\}$ that fall under Category 2. Call this list as $List_{cat_2}$ and sort it in increasing order of transport energy requirement.
3. Construct Trigger Matrix $TM_{\Pi,k}$ for all region k The defined rules associated with the rows of $TM_{\Pi,k}$ must be the rules that lessen the multiplicity of objects in region k. These rules are represented in the set $TR(k)$. Again, let

the additional rules from $R_{add,k}$ be represented in the rows as well. The set of objects represented in the columns of $TM_{\Pi,k}$ remains $PO(k)$. $TM_{\Pi,k}$ has dimensions $|TR(k) \cup R_{add,k}| \times |PO(k)|$. $TM_{\Pi,k}(r, \alpha)$ gives the multiplicity of α in region k that is required to apply rule r once.

4.Set the length of the vector of unknowns (extended application vector) $\mathbf{a'}_{i,k}$ for all region k The length of $\mathbf{a'}_{i,k}$ is $|TR(k) \cup R_{add,k}|$.

5. Solve system of linear equation Find all solutions to the equation

$$\mathbf{a'}_{i,k} \cdot TM_{\Pi,k} = \mathbf{C}_{i-1,k} \tag{4}$$

Again, because the application vector's ($\mathbf{a'}_{i,k}$) elements represent the number of application of rules, it must not contain negative numbers i.e. the elements must always be natural numbers. The value $\mathbf{a'}_{i,k}(r)$ returns either the number of application of each rule $r \in TR(k)$ or how many object α is unevolved or unmoved (if $(r : \alpha \to \alpha) \in R_{add,k}$). $TR(k)$ and $R_{add,k}$ are disjoint sets.

6. Filter solutions in step 5 For each region k, if $List_{cat_2} \neq \emptyset$, scan the sorted $List_{cat_2}$ and find out the first object, labelled $\alpha_{cat_2,min}$, falling under Category 2 whose corresponding identity rule application is non-zero, i.e. $\mathbf{a'}_{i,k}(\alpha_{cat_2,min} \to \alpha_{cat_2,min}) > 0$. Since $List_{cat_2}$ is sorted in increasing order of energy requirement for transport, $\alpha_{cat_2,min}$ has the least energy required for communication. Label its corresponding energy as $energy(\alpha_{cat_2,min})$. Filter solutions in step 5 by adding, for each region k with a non-empty $List_{cat_2}$, the inequality below:

$$\mathbf{a'}_{i,k}(e \to e) < energy(\alpha_{cat_2,min}) \tag{5}$$

7. For each solution in step 6, find $\mathbf{a}_{i,k}$ When values for $\mathbf{a'}_{i,k}$ in all region k are found, disregard all identity rules $r' \in R_{add,k}$. Fill the values of an application vector $\mathbf{a}_{i,k}$ through the equation

$$\mathbf{a}_{i,k}(r) = \mathbf{a'}_{i,k}(r),\ r \in R_k. \tag{6}$$

For every communication rule $r \in IR(r, k') \cap IR(r, k'')$,

$$\mathbf{a}_{i,k'}(r) = \mathbf{a}_{i,k''}(r) = \mathbf{a'}_{i,k'}(r) \tag{7}$$

for all communication rule $r \in IR(k') \cap IR(k'')$ where region k' is the sending region.

Theorem 2 *(from [7]) All possible $\mathbf{a}_{i,k}$ generated through the above methodology leads to a valid $\mathbf{C}_{i,k}$ yielded from $\mathbf{C}_{i-1,k}$ in one computational step.*

3 Graphics Processing Unit (GPU)

3.1 On Using GPUs

Although GPUs are initially used for image processing, it is actually designed to handle computationally demanding applications. Thus, its use is extended to accommodate different applications. GPU has been widely used to work with highly

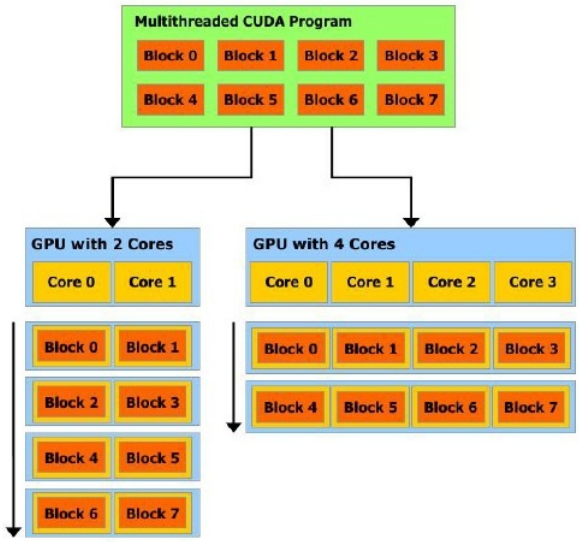


Fig. 2. NVIDIA CUDA automatic scaling(More cores, faster execution), from [3]

parallel applications due to its parallel nature as compared to setting-up multiple CPUs that will harness the same power by that of a GPU [3]. Another reason is that they provide not only the hardware but also application programming interfaces (APIs) for computation. As mentioned in [11], the GPU is designed to cater to a class of applications with the following characteristics,

- Computational requirements are large.
- Parallelism is substantial.
- Throughput is more important than latency.

3.2 Compute Unified Device Architecture (CUDA)

The Compute Unified Device Architecture (CUDA) programming model is introduced by NVIDIA, a manufacturer of GPUs. CUDA is a hardware and software architecture that runs highly parallel computations on the family of GPUs manufactured by NVIDIA [3]. With this feature and compatibility with today's leading GPU devices, CUDA became popular and progress has been made to make programming in CUDA easier. Though CUDA is an extension of the C programming language for parallel computations, programmers can also access CUDA APIs with FORTRAN, Haskell, Perl, Python, Ruby, and etc.

The parallel code written in extensions of the C programming language is executed in multiple threads within multiple blocks which are in turn parts of a grid of blocks. These blocks belong to the GPU. Each GPU consists of multiple cores having their own block of threads [3]. This feature is illustrated in Figure 2.

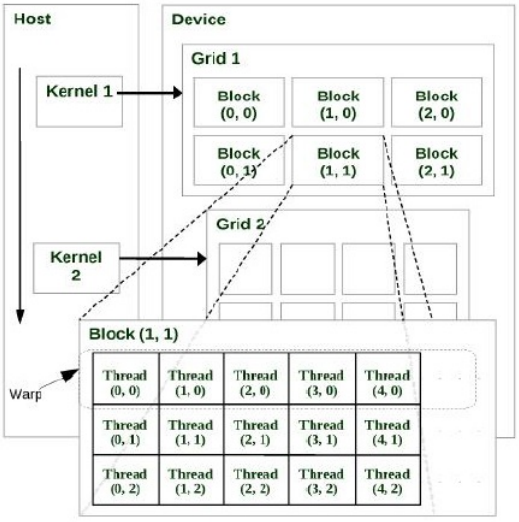


Fig. 3. NVIDIA CUDA showing execution of sequential code in the host and parallel execution of the kernel function in the device, from [3]

CUDA distinguishes CPU (host part) from the GPU (device part). Typically, initial operations and preparations are done before proceeding to the demandingly parallel computations. CUDA deals with this by performing preparations in the host and moving the prepared data to the GPU for fast parallel computation then, moving the results of the parallel computation back to the host for further interpretation of output. The entity that connects the CPU and GPU, or makes the data movement possible, is the *kernel function.* This function is called in the host but is executed in the device [13] as illustrated in Figure 3. Usually, preparations for the data are done to maximize parallelism. Operationally, the CPU controls the flow of the application program while the GPU acts as a co-processor to the host where demanding computations are held.

4 CUDA GPU Computing and ECPe systems

4.1 Generating all possible configuration vectors

The process of finding all possible configurations for all regions of an ECPe system is illustrated by Figure 4. Input files contain information regarding the ECPe system to be simulated. Some of the necessary information regarding the ECPe system are the current configuration, membrane structure, number of possible objects, number of involved rules and Transition matrix for each region k. Configuration vectors, the initial configuration and the generated configurations, are

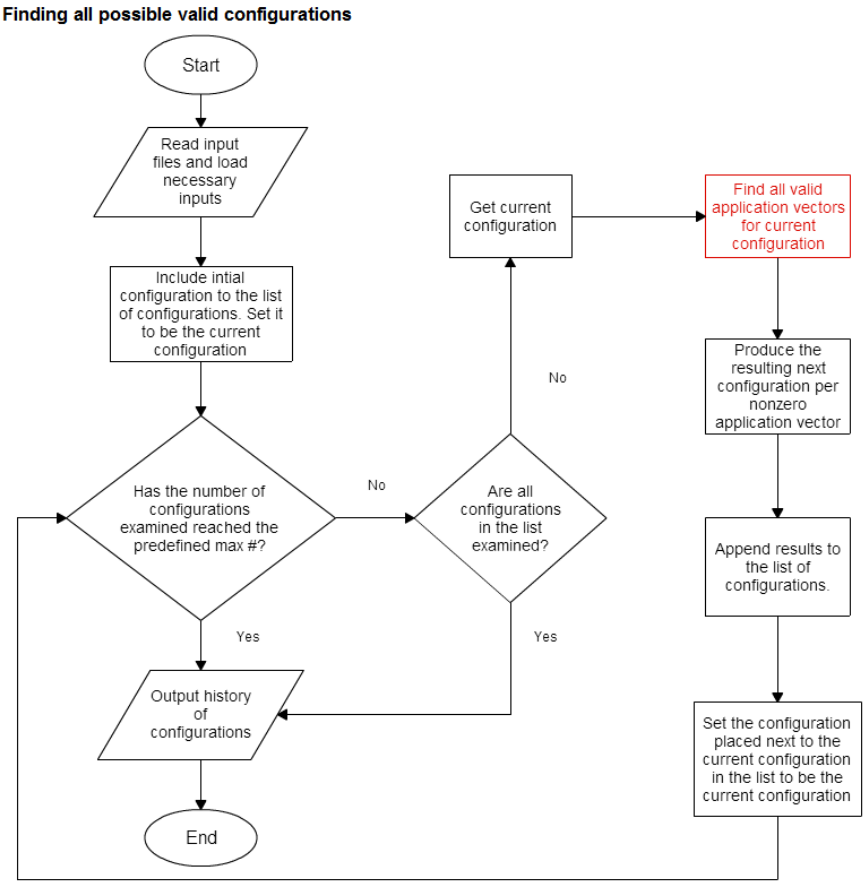


Fig. 4. Flow chart of finding all possible configurations of an ECPe system until a halting configuration is reached or until the maximum number of iterations set by the user is reached.

written to a file. Thus, to explore each configuration, each configuration vector must be read first from a file. This vector will be labelled as the current configuration and will undergo computations, as illustrated by Figure 5 to determine all valid application vectors and to produce possible next configurations. Results are appended at the bottom of this file.

The process illustrated in Figure 4 will generate all possible next configurations until a maximum number of explored configurations have been reached, or no further configurations needs to be explored (the case of halting configuration).

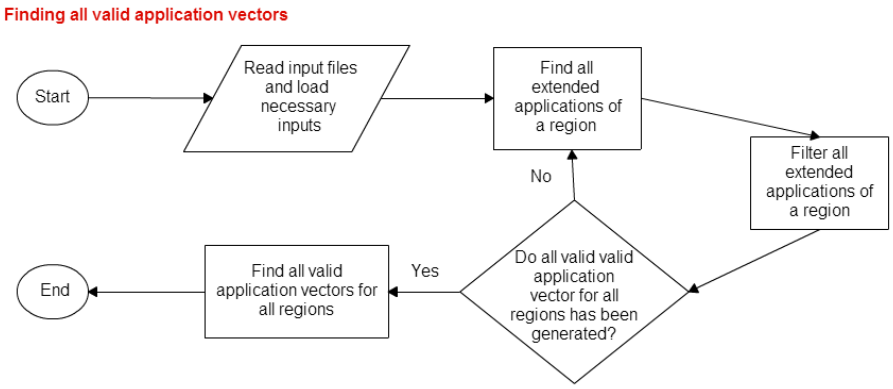


Fig. 5. Flow chart of finding all possible application vectors for all region(s) of an ECPe system. A subprocess of finding all possible configuration vectors illustrated in Figure 4, highlighted in red.

4.2 Generating all possible application vectors

The generation of all possible application vectors is guided by the methodology for forward computing as discussed in Section 2.4. It is further illustrated in Figure 5.
Steps 1-4: Preparation of input We can notice that steps 1 through 4 are done to prepare the values needed for generating application vectors. It is in Step 5 that computation starts to generate all possible application vectors. Thus, to simulate steps 1 through 4, we read from a file the necessary values needed for generating all possible application vectors.
Step 5: Finding extended application vectors through solving a system of linear equations Finding extended valid applications involves finding solutions to a system of equations. Forward computing step 5 in Section 2.4 details the system of linear equations to be solved. This can be implemented in parallel. We extend the sequential implementation of computation on ECPe systems without antiport rules and adapt ideas on parallelizing some of the processes of the implementation presented in [8]. The implementation uses localized matrix representation discussed in Section 2.4.

The following observations on the system of linear equations used in finding extended valid application vectors are helpful in solving the system in parallel. Let us take as an example the ECPe system Π illustrated in Figure 1. If we let $C_{i-1,1}$ be the initial configuration, the system of linear equations produced upon performing $\mathbf{a'}_{i,1} \cdot TM_{\Pi,1} = \mathbf{C}_{i-1,1}$ is,

$$a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2$$
$$a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1$$
$$a'_{i,1}(r'_{21}) + a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2$$

- **Each variable's value in the resulting system of linear equations can only be a natural number** since these variables correspond to the number of applications of a rule.
- **Every equation in the system corresponds to an object condition**. Since the dimension of the Trigger matrix $TM_{\Pi,k}$ is $|IR(k) \cup R_{add,k}| \times |PO(k)|$, each equation resulting upon performing $\mathbf{a'}_{i,1} \cdot TM_{\Pi,1} = \mathbf{C}_{i-1,1}$, will give us the equation for the application of each rules (i.e. the variables in the left hand side of the equation) whose available triggering objects is in the right hand side of the equation.

$$\begin{aligned} a&: \quad a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2 \\ \#&: \quad a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1 \\ e&: \ a'_{i,1}(r'_{21}) + a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2 \end{aligned}$$

- **If k is a sending region for at least one (1) communication rule, an energy condition must be in the resulting system of equations for that region**. In our example, the resulting system of linear equations of the two regions are,
  Region 1:

$$\begin{aligned} a&: \quad a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2 \\ \#&: \quad a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1 \\ e&: \ a'_{i,1}(r'_{21}) + a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2 \end{aligned}$$

  Region 2:

$$\begin{aligned} a&: \ a'_{i,2}(r'_{21}) + a'_{i,2}(Add_{21}) = 0 \\ \#&: \quad a'_{i,2}(r_{21}) + a'_{i,2}(r'_{22}) = 0 \end{aligned}$$

  Since region 2 is a receiving region only for both symport rules, it does not have an energy condition unlike region 1 which is a sending region for both symport rules.
- **Each variable (representing application of a particular communication rule) in the energy equation occurs in exactly one other object condition**. Moreover, **only variables associated with communication rules in energy equation can occur in other object conditions**, other variables occur exactly in one equation.

$$\begin{gathered} a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2 \\ a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1 \\ a'_{i,1}(r'_{21}) + a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2 \end{gathered}$$

- **Coefficients of the terms for non-energy conditions is always one** because of non-cooperative form of the evolution rules and the restriction to communication rules that only one object can be transported by a rule. **Coefficients of the terms in energy equation can be any positive integer** since communication rules must consume any amount of energy $|e| \geq 1$. The union of all rules in the non energy conditions not including the identity

rules represents the set of trigger rules. For example, if (ae, in) is (ae^2, in) and $(\#e, in)$ is $(\#e^3, in)$, the resulting system of linear equations becomes,

$$\begin{gathered} a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2 \\ a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1 \\ 2a'_{i,1}(r'_{21}) + 3a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2 \end{gathered}$$

With the above observations, all possible application vectors can be found by solving first the energy condition. We can use each solution for solving non-energy equations since each variable representing rule application of a certain communication rule in the energy equation occurs in one and only one other object condition. For example in this equation,

$$\begin{aligned} a&: \quad a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + a'_{i,1}(r'_{21}) = 2 \\ \#&: \quad a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{11}) = 1 \\ e&: \; a'_{i,1}(r'_{21}) + a'_{i,1}(r'_{22}) + a'_{i,1}(Add_{12}) = 2 \end{aligned}$$

The possible solutions to the energy condition are,

$a'_{i,1}(r'_{21})$	$a'_{i,1}(r'_{22})$	$a'_{i,1}(Add_{12})$
2	0	0
1	1	0
0	2	0
0	1	1
0	0	2
1	0	1

We substitute the rule application to its corresponding object to communicate. Let's take the first three solutions as an example. It will give us,

$$\begin{array}{|c|} \hline a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + 2 = 2 \\ 0 + a'_{i,1}(Add_{11}) = 1 \\ 2 + 0 + 0 = 2 \\ \hline a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + 1 = 2 \\ 1 + a'_{i,1}(Add_{11}) = 1 \\ 1 + 1 + 0 = 2 \\ \hline a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) + 0 = 2 \\ 2 + a'_{i,1}(Add_{11}) = 1 \\ 0 + 2 + 0 = 2 \\ \hline \end{array}$$

Then subtract it to the number in the right hand side of the equation which corresponds to the current count of the associated communicated object.

$a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 2 - 2 = 0$ $a'_{i,1}(Add_{11}) = 1 - 0 = 1$ $2 + 0 + 0 = 2$
$a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 2 - 1 = 1$ $a'_{i,1}(Add_{11}) = 1 - 1 = 0$ $2 + 0 + 0 = 2$
$a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 2 - 0 = 2$ $a'_{i,1}(Add_{11}) = 1 - 2 = -1$ $2 + 0 + 0 = 2$

If this yields a negative number, then this solution for energy equation is not applicable. If not, solutions for the objects constitute one extended application vector [8]. In the above illustrations the solution 0 2 0 is not a valid solution. The new linear systems of equation to solve are now,

$$\begin{array}{ll} a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 0 & a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 2 \\ a'_{i,1}(Add_{11}) = 1 & a'_{i,1}(Add_{11}) = 0 \end{array}$$

$$\begin{array}{ll} a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 1 & a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 2 \\ a'_{i,1}(Add_{11}) = 0 & a'_{i,1}(Add_{11}) = 2 \end{array}$$

$$\begin{array}{c} a'_{i,1}(r_{11}) + a'_{i,1}(r_{12}) = 1 \\ a'_{i,1}(Add_{11}) = 1 \end{array}$$

The new systems of linear equations are then solved independently of each other.

Solving Equations. As what has been observed, each equation in the system of linear equations are not always independent nor dependent of each other. In case of regions that has energy condition, each non-energy object condition are dependent on it. However, each non-energy equation iscomputation independent of each other, this is also true in regions that does not have energy condition. Thus, the approach is to solve each equation independently of each other, except for energy conditions. As discussed in the previous section, regions with energy condition solves the equation for the energy condition first. After reflecting the solutions for the energy condition to the equations for the non-energy condition, solutions to the equations for the non-energy solutions are computed.

For regions without energy condition, the computation goes straight to the independent computations for solutions to the equations for the non-energy equation. The computation for the solutions to each equation can be reduced to the Integer Partition Problem as proposed in [8].

Solving Integer Partition Problem. Integer partitioning means finding a way of writing r as a sum of positive integers which are called partitions. To use these partitions in solving a non-energy equation, the number of partitions must be less than or equal to the number of variables m in the left hand side of the equation.

combinations	sticks and pebbles	integer partitions
123	||| · ·	0002
124	|| · |·	0011
125	|| · ·|	0020
134	| · ||·	0101
135	| · | · |	0110
145	| · ·||	0200
234	·|||·	1001
235	·|| · |	1010
245	·| · ||	1100
345	· · |||	2000

Fig. 6. Interpreting $\binom{5}{3}$ as integer partitions through the sticks and pebbles analogy.

If the number of partitions is less than m, zeroes must be padded accordingly so that it represents a vector with m elements. Then permutations of the partitions must be obtained. The problem of partitioning r with m components and then obtaining permutation of the partition can be reduced to solving combinations of $\binom{n}{s}$ where $n = r + m - 1, s = m - 1$.

Combinations can be interpreted or converted into integer partitions through the sticks and pebbles analogy (*see Figure 6*). In a $\binom{n}{s}$ combinations, there are n positions, s sticks, and $n - s$ pebbles. Elements in the combinations represent the position of each sticks and the pebbles take the position where there is no stick. The number of pebbles between each stick are the elements of the integer partition.

However, solutions to each equation in the system of linear equations does not end on the generation of integer partitions. Since equations for the energy condition can contain a coefficient greater than or equal to 1, the generated integer partitions need to be further filtered to ensure that they are valid solutions to the equation for the energy condition.Given an energy equation of the form $c_1a_1+c_2a_2+...+c_ma_k = n$, we first obtain a lexicographic order of the partitions, add zeroes to these partitions accordingly, and get the permutations of the partitions. Equate these permutated partitions to the equation to get the energy solutions. Let $p_1, p_2, ..., p_m$ be the partitions of r with m components. $a_1 = \frac{c_1}{p_1}, a_2 = \frac{c_2}{p_2}, ..., a_m = \frac{c_m}{p_m}$ is an energy solution if every $a_i, i \in 1, 2, ..., m$ is a natural number.

No similar filtering is needed for the solutions to the equations for the non-energy condition since we have observed that the coefficient of the equations for the non-energy condition is always equal to 1.

Parallel Implementation To solve the equation for energy condition, the process of generation of combinations and its conversion to integer partitions (whose process is described in the previous section), are given to the threads in the GPU device.

If the number of permuted integer partitions C (also equal to the number of combinations of $\binom{n}{s}$) is less than or equal to the number of all available threads T then each thread will produce one permuted integer partition only and equate this to the energy equation to check if it is valid. If $C > T$, then each thread will produce and check $\lceil \frac{C}{T} \rceil$ partitions except for the last thread. The last thread will produce at most $\lceil \frac{C}{T} \rceil$ partitions. The number of blocks to be used is equal to $\lceil \frac{T}{M} \rceil$, where M is equal to maximum threads per block. Since the non-energy equations

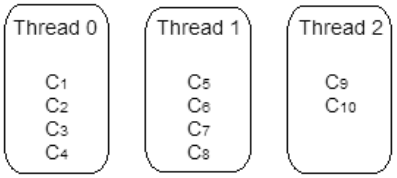


Fig. 7. When $C > T$. If there are three (3) available threads, each thread, except for the last thread, will generate four (4) integer partitions namely, $C_i, ..., C_{i+4}$and the last thread will generate three (3) integer partitions.

are independent of one another (they depend only on the energy equation) their solutions can be obtained in parallel. If there is an energy equation, the solutions to non-energy equations can be obtained in parallel after the solutions of energy equations are substituted in the system. For the implementation in GPU, a block is responsible for generating solutions of a non-energy equation. At most M threads will produce the partitions for the assigned non-energy equation to the block where they belong. If the number of non-energy equations NC is less than or equal to the number of blocks B then each block will produce solutions for one non-energy equation only. If $NC > B$, then each block will produce solutions for $\lceil \frac{NC}{B} \rceil$ number of non-energy equations except for the last block. The last block will produce solutions for at most $\lceil \frac{C}{T} \rceil$ number of non-energy equations.

Step 6: Filter extended application vectors. If the system has an energy equation, the extended application vectors must be filtered. Filtering is done in parallel. If the number of extended application vectors E is less than or equal to the number of all available threads T then each thread will filter one extended application vector only. If $E > T$, then each thread will filter $\lceil \frac{E}{T} \rceil$ extended application vectors except for the last thread. The last thread will produce at most $\lceil \frac{E}{T} \rceil$ extended application vectors. The number of blocks to be used is equal to $\lceil \frac{E}{M} \rceil$. The analogy of job allocation to threads is similar to Figure 7.

Step 7: Finding all valid application vectors In this step other necessary values are loaded from a file as well as the extended application vectors generated from the previous steps (namely, Step 5 and 6) Identity rules at each region is removed. Thereafter, the extended application vectors are merged with each other to reflect the application of symport rules from sending regions to receiving regions

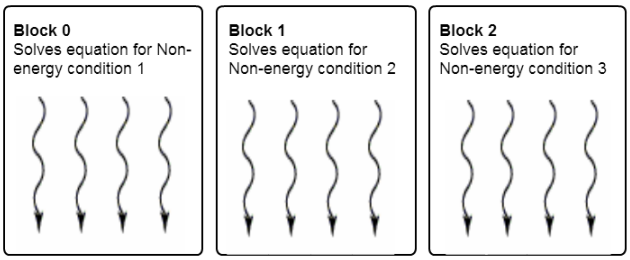


Fig. 8. When $NC \leq B$. If there are three (3) available blocks and equation for non-energy conditions, each block will generate solutions to each equations utilizing the threads allocated for each block to generate integer partitions similar to

since the computation done are local to each region only. This will give us all the valid application vectors of region k at time i. This step is done in the host only, using computing and memory resources of the host only.

5 Space and Communication Requirements

STEP	HOST ALLOCATION	HOST TO DEVICE	DEVICE ALLOCATION	DEVICE TO HOST
Finding solutions for the energy equation in the system of linear equations	1 solcount ×left		(C ×left) 1 solcount ×left	1 solcount ×left
Finding solutions for non-energy equation in the system of linear equations	TM_rows ×TM_cols NEeqCount × TM_rows ×2	NEeqCount ×2	NEeqCount × TM_rows × 2 NEeqCount × validCount NEeqCount NEIPsize	
Finding extended application vectors	TM_rows × extAppVecCount 1		TM_rows × extAppVecCount 1	TM_rows × extAppVecCount 1

Table 1. Memory allocations in Host and Device together with the communication between them if there is an energy equation.

Tables 1 and 2 summarizes the space requirements and size of message communicated between host and device. Note that in case of allocation, the value one

STEP	HOST ALLOCATION	HOST TO DEVICE	DEVICE ALLOCATION	DEVICE TO HOST
Finding solutions for non-energy equation in the system of linear equations	TM_rows× TM_cols	NEeqCount×2	NEeqCount× NEIPsize	
Finding extended application vectors	TM_rows			

Table 2. Memory allocations in Host and Device together with the communication between them if no energy equation.

(1) on the table implies that an extra variable is allocated to hold values for use as counter or flag. A value 1 is also used as signal for communication between host and device. In Tables 1 and 2, cells without values indicate no allocation or communication. The variables used are defined as follows:

- $membraneCount$ is the number of membranes in the system
- IR is the number of involved rules in the region
- TM_rows is the number of rows of the trigger matrix
- TM_cols is the number of columns in the trigger matrix
- $NEeqCount$ is the number of non-energy equations. For sender regions, this is equal to $TM\_rows - 1$.
- $left$ is the number of terms(partitions) in the left-hand side of the energy equation
- $right$ is the multiplicity of an object at the right-hand side of the energy equation.
- $C = \frac{(left+right-1)!}{(right)!(left-1)!)}$ is the number of integer partitions of right hand side of the energy equation (multiplicity of energy in the system) with components equal to the number of partitions in the left hand side.
- $solcount = O(C)$ is the number of valid solutions for the energy equation
- $validCount$ is the number of energy solutions for the system of linear equations.
- $extAppVecCount = O(C^{NEeqCount})$ is the number of extended application vectors of the region
- $NEIPsize$ is the summation of combinations of $\binom{i}{j}$ of each non-energy equation, where $i = m2+r2-1$, $j = m2$, $m2$ being the new number of partitions in the left-hand side of the equation, and $r2$ being the new right-hand side after the energy solutions are substituted.
- $validExtAppCount = O(C^{NEeqCount})$

In the variables given, the first six items are variables that is only dependent on rules and membrane structures for a given P system, while the rest are variables that are also dependent on configuration. Consequently, this means that the rest of the variables excluding the first six potentially changes every time a new configuration is explored.

From these tables, it can be observed that the required variables for regions without energy equations are significantly less than regions with energy equations.

This is expected since regions without energy equations only act as receivers and need not be concerned with the number of application of communication rules. Thus, we only analyze the requirement for sending regions where energy equations needs to be addressed.

Focusing on sender regions, Table 1 shows that the most expensive communication step occurs upon accomplishment of finding extended valid application vectors where the set of all valid application vectors will be communicated from host to device as a matrix with size at most $C^{NEeqCount} \times TM\_cols$.

The required memory space in host in finding all valid extended application vectors (i.e. Steps 1-6 of the methodology for forward computing described section 2.4) when the region has energy is,

$$E_h = C \times left + TM\_rows \times TM\_cols + (NEeqCount \times TM\_rows \times 2)$$

$$+(TM\_rows \times C^{NEeqCount} \times 2) + 3$$

The needed memory space in host when finding all valid application vectors(i.e. Step 7 of the methodology for forward computing described section 2.4) is,

$$M = membraneCount \times (membraneCount + IR + IR \times validAppVecCount)$$

From this, we can conclude that the upperbound of the memory needed in host is $O((E_h + M))$.

On the other hand, the required memory space in device in finding all valid extended application vectors (i.e. Steps 1-6 of the methodology for forward computing described in Section 2.4) when the region has energy is,

$$E_d = C \times left \times 2 + (NEeqCount \times TM\_rows \times 2) + NEeqCount \times C + NEeqCount$$

$$+NEIPsize + (TM\_rows \times C^{NEeqCount} \times 2) + 3$$

Note that, when finding all valid application vectors (i.e. Step 7 of the methodology for forward computing described in Section 2.4), there is no need to allocate memory in device.

6 Conclusion and Future Works

In this paper, we were able to present a hybrid implementation of computation for ECPe systems without antiport by employing GPUs. Our implementation makes use of matrix representation and operations discussed in [6]. To improve our implementation, the following are recommended for further study:

- **Optimize memory usage**
  Some processes in the parallel simulation makes use of arrays that may become large (depending on the number of rules and objects in a region, number of regions, and etc). An example is the padding of energy and non-energy solutions

with 0s so that all generated solutions will have the same dimension and is faithful to the size of extended application vector determined in Step 4 of the methodology for forward computing described in Section 2.4. It is primarily done to make merging easier when it comes to finding valid application vectors. However, doing so might require a significant amount of memory.

- **Accept input from P Lingua**
  It is a good characteristic of the program if it is P Lingua compatible. That is, from the definition of an ECPe system, say Π. Its corresponding P Lingua format is accepted as an input of the program. In this way, running any ECPe system in the program would be easier. The number of input files used by the program will also lessen.
- **Concurrent processes for solving energy solutions of a region and non-energy solutions of regions without energy equation**
  Our approach in solving system of linear equations is to solve first for the energy solutions if there is an energy equation in the said system of linear equations. Thus, in this way we cannot generate solutions for the non-energy equations yet. However, it can be done that while energy solutions are being generated, non-energy solutions for other regions are also being generated.
- **Extend parallel simulation of ECPe system to include antiport rules**
  Since the current implementation is done without antiport rules and there is still no existing parallel simulation of ECPe systems which include antiport rules as of the writing of this paper, it is attractive to extend the work to include antiport rules.

Acknowledgements

F.G.C. Cabarle and R.A. B. Juayong are supported by the Engineering Research and Development (ERDT) Scholarship Program. H.N. Adorna is funded by a DOST-ERDT research grant and the Semirara Mining Corporation professorial chair of the UP Diliman, College of Engineering. M.A. Martínez-del-Amor is supported by "Proyecto de Excelencia con Investigador de Reconocida Valía P08-TIC-04200" from Junta de Andalucía, and project TIN2012-37434 from "Ministerio de Ciencia e Innovación" of Spain, both co-financed by FEDER funds.

References

1. H. Adorna, Gh. Păun, M.J. Pérez-Jiménez: On Communication Complexity in Evolution-Communication P Systems, *Romanian Journal of Information Science and Technology, Vol. 13 No. 2 pp. 113-130, 2010*
2. S. G. Akl: Adaptive and optimal parallel algorithms for enumerating permutations and combinations, *The Computer Journal,30, 5 (1987), 433436*
3. F. Cabarle, H. Adorna, M. A. Martínez-del-Amor.: Simulating Spiking Neural P Systems Without Delays Using GPUs, *International Journal of Natural Computing Research (IJNCR), Vol. 2 No. 2 pp. 19-31, 2011*

4. F. Cabarle, H. Adorna, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez: Improving GPU Simulations of Spiking Neural P Systems, *Romanian Journal of Information Science and Technology Volume 15, Number 1, pp. 520, 2012.*
5. G. Ciobanu, G. Wenyuan: P Systems Running on a Cluster of Computers, *In Workshop on Membrane Computing pp. 123-139, 2003.*
6. R.A. Juayong, H. Adorna: A Matrix Representation for Computations in Evolution-Communication P Systems with Energy, *Proc. of Philippine Computing Science Congress, Naga, Camarines Sur, Philippines, March 3-4, 2011*
7. R.A. Juayong, H. Adorna: Computing on Evolution-Communication P Systems with Energy Using Symport Only, *Workshop on Computation: Theory and Practice 2011 (WCTP 2011), UP Diliman NISMED auditorium*
8. R.A. Juayong, F.G.C. Cabarle, H. Adorna, M. Martínez-del-Amor: On the Simulations of Evolution-Communication P Systems with energy without Antiport Rules for GPUs, *Technical report of the 10th Brainstorming Week in Membrane Computing, Seville, Spain, Feb 2012.*
9. G. D. Knott: A Numbering System for Combinations. *Comm. ACM, Vol. 17, No. 1, pp. 45-46, January 1974.*
10. C. Mifsud, Algorithm 154: combination in lexicographical order,*Communications of the ACM, p. 103. Volume 6 Issue 3, March 1963.*
11. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips: GPU Computing, *Proceedings of the IEEE, Vol. 96 No. 5 pp. 879-899, 2008*
12. Gh. Păun: Introduction to Membrane Computing. In: Gabriel Ciobanu, Mario J. Pérez-Jiménez and Gheorghe Păun, eds: Applications of Membrane Computing, Natural Computing Series. Springer, pp.142. (2006)
13. Gh. Păun, M.J. Pérez-Jiménez: Spiking Neural P Systems. Recent Results, Research Topics. Algorithmic Bioprocesses, Natural Computing Series. Springer. pp. 273-291. (2009)
14. Gh. Păun, M.J. Pérez-Jiménez: Solving Problems in Distributed Way in Membrane Computing: dP System, *Int. J. of Computers, Communication and Control, Vol. 5 No. 2 pp. 238-250, 2010*
15. A.E. Porreca, A. Leporati, G. Mauri, C. Zandron: Introducing a space complexity measure for P systems, *Intern. J. Computers, Communications and Control, Vol. 4 No. 3 pp. 301310, 2009.*

Appendix

Assumptions on input files

Figure 9 and 11 shows the file format for all the necessary input files. Given below are the assumptions for the input files needed in our implementation.

- A total order for objects is assumed for all of the vectors and matrices used. For example, if the correspondence of possible objects in the Transition matrix is $< a, \#, e >$, then the correspondence of possible objects in the Trigger matrix must also be $< a, \#, e >$.
- A total order for rules is assumed for all of the vectors and matrices used. For example, if the correspondence of involved rules in the Transition matrix is $< r_{11}, r_{12}, r'_{21} >$, then the correspondence of involved rules in the Trigger matrix must also be $< r_{11}, r_{12}, r'_{21} >$.
- If a region involves symport rule(s), the correspondence of the rule(s) is assumed to be at the latter part of the vectors and matrices defined for the region. For example, if r_{11}, r_{12}, r'_{21}, r'_{22} are the rules involved in region k, and r'_{21}, r'_{22} are symport rules, The total order should be $< r_{11}, r_{12}, r'_{21}, r'_{22} >$ or $< r_{11}, r_{12}, r'_{22}, r'_{21} >$.
- If a region contains symport rule(s) wherein it is a receiving region, the correspondence of the said symport rules(s) is assumed to be at the last part of the vector and matrices defined for the region. For example, if r_{11}, r_{12}, r'_{21}, r'_{22}, r'_{23} are the involved rules in region k, and r'_{21} is the symport rules where in region k is a sending region and r'_{22}, r'_{23} are symport rules where in region k is a receiving region. The total order should be, $< r_{11}, r_{12}, r'_{21}, r'_{22}, r'_{23} >$ or $< r_{11}, r_{12}, r'_{21}, r'_{23}, r'_{22} >$.

```
2

[1[2]2]1

4
3
2 1 2
1 0 0
-1 0 2
-1 0 -1
0 -1 -1
e e s s

3
2
1 0
0 0
1 0
0 1
e r r
```

```
number or regions

membrane structure

number of Involved Rules in a region
number of Possible Objects in a region
initial configuration
Transition Matrix
.
.
.
rule types

number of Involved Rules in a region
number of Possible Objects in a region
initial configuration
Transition Matrix
.
.
.
rule types
```

Fig. 9. Input file for generating all possible next configuration (*trans_file.txt*)

```
2 1 2 $ 1 0 $
0 0 5 $ 1 1 $
2 0 3 $ 1 1 $
4 0 1 $ 1 1 $
0 0 2 $ 2 1 $
2 0 0 $ 2 1 $
0 1 0 $ 3 0 $
0 0 5 $ 1 1 $
0 0 7 $ 1 1 $
2 0 5 $ 1 1 $
4 0 3 $ 1 1 $
0 0 4 $ 2 1 $
2 0 2 $ 2 1 $
0 0 1 $ 3 1 $
0 0 9 $ 1 1 $
2 0 7 $ 1 1 $
4 0 5 $ 1 1 $
```

```
1
1_1
1_2
1_3
1_4
1_5
1_6
1_1_1
1_2_1
1_2_2
1_2_3
1_2_4
1_2_5
1_2_6
1_3_1
1_3_2
1_3_3
```

Fig. 10. Output file of all possible next configurations Left: list of all possible next configurations(conf.txt), Right: label of all possible next configurations(*conf_index.txt*)

```
1

6
3
1 0 0
1 0 0
1 0 1
0 1 1
0 1 0
0 0 1

1
3
1 1 1
5
1

2
3
2

2

2
2
0 1
1 0

0

2
1
1
```

```
Region label

Number of trigger rows
Number of trigger cols
Trigger matrix

//if there is energy eq. this will follow
1
Number of partitions in the left hand side of energy equation
coefficient array
cat2min row position in trigger matrix
cat2min energy required

No. of Non energy equations (m)
No. of partitions in the lefthandside of non-energy equation 1
No. of partitions in the lefthandside of non-energy equation 2
...
No. of partitions in the lefthandside of non-energy equation m

//if no energy equation
0

No. of Non energy equations (m)
No. of partitions in the lefthandside of non-energy equation 1
No. of partitions in the lefthandside of non-energy equation 2
...
No. of partitions in the lefthandside of non-energy equation m
```

Fig. 11. Input file for generating extended application vectors (*find_extended_app.txt*)

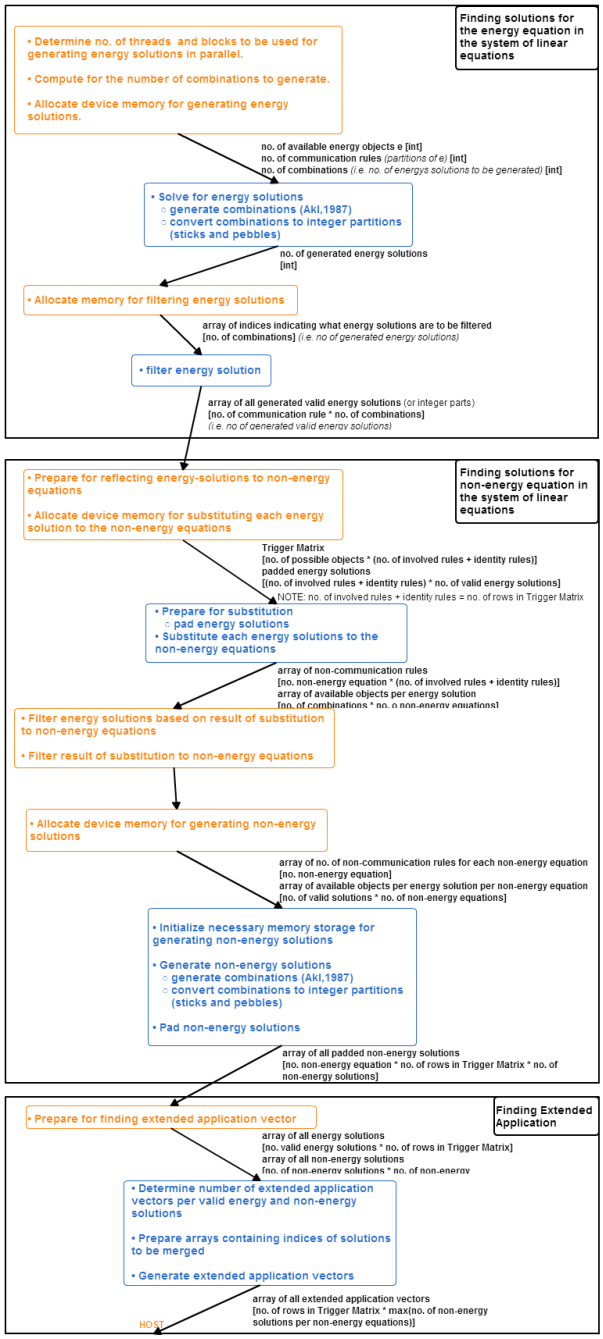


Fig. 12. Flow and communication between host and device for Step 5: Finding extended valid application vectors. (If region has an energy condition)

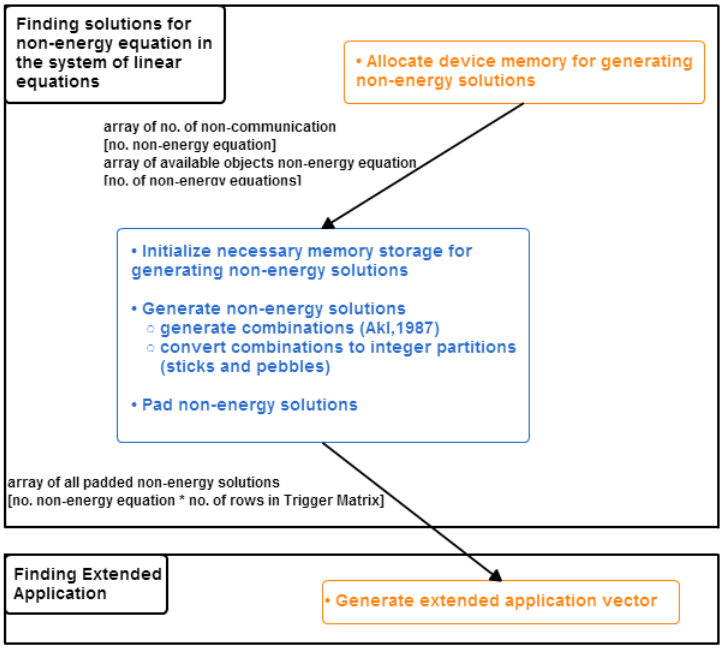


Fig. 13. Flow and communication between host and device for Step 5: Finding extended valid application vectors. (If region does not have an energy condition)

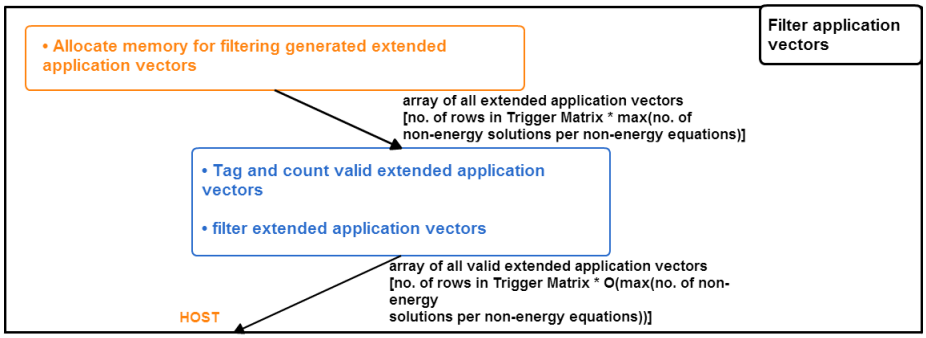


Fig. 14. Flow and communication between host and device for Step 6: Filtering extended valid application vectors.

2D P Colonies and Modelling of Liquid Flow Over the Earth's Surface

Luděk Cienciala, Lucie Ciencialová, and Miroslav Langer

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

Summary. We continue the investigation of 2D P colonies introduced in [1], a class of abstract computing devices composed of independent agents, acting and evolving in a shared 2D environment where the agents are located. Agents have limited information about the contents of the environment where they can move in four directions.

1 Introduction

P colonies were introduced in the paper [5] as formal models of computing devices belonging to membrane systems and similar to formal grammars called colonies. This model is inspired by the structure and the behaviour of communities of living organisms in a shared environment. The independent organisms living in a P colony are called agents. Each agent is represented by several objects embedded in a membrane. The number of objects inside each agent is the same and constant during computation. The environment is agents' communication channel and storage place for objects. At any moment all agents "know" about all the objects in the environment and they can access any object immediately. More information about P colonies the reader can find in [4, 2]. P colonies are one of the types of P systems. They were introduced in 2000 in [6] by Gheorghe Păun as a formal model inspired by the structure and the behaviour of cells.

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

For more information about P systems see [8, 7] or [11].

In addition 2D P colony has the environment in a form of a 2D grid of square cells. The agents are located in this grid and their view is limited to the cells that immediately surround them. Based on the contents of these cells, the agents decide their future locations.

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

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

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

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

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

2 Definitions

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

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

A multiset of objects M is a pair $M = (V, f)$, where V is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : V \to N$; f assigns to each object in V its multiplicity in M. The set of all multisets with the set of objects V is denoted by V°. The set V' is called the support of M and is denoted by $supp(M)$ if for all $x \in V'$ $f(x) \neq 0$ holds. The cardinality of M, denoted by

$|M|$, is defined by $|M| = \sum_{a \in V} f(a)$. Each multiset of objects M with the set of objects $V' = \{a_1, \ldots a_n\}$ can be represented as a string w over alphabet V', where $|w|_{a_i} = f(a_i);\ 1 \leq i \leq n$. Obviously, all words obtained from w by permuting the letters represent the same multiset M. The ε represents the empty multiset.

3 2D P colonies

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

The first rule type, called the evolution rule, is of the form $a \ \rightarrow b$. It means that the object a inside the agent is rewritten (evolved) to the object b. The second rule type, called the communication rule, is of the form $c \leftrightarrow d$. When the communication rule is performed, the object c inside the agent and the object d outside the agent swap their places. Thus, after the execution of the rule, the object d appears inside the agent and the object c is placed outside the agent.

The third rule type, called the motion rule, is of the form matrix $3 \times 3 \rightarrow$ move direction. Based on the contents of the neighbouring cells, an agent can move one step to the left, right, up or down.

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

Definition 1. *The 2D P colony is a construct*
$$\Pi = (A, e, Env, B_1, \ldots, B_k, f), k \geq 1, \textit{ where}$$

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

The configuration of the 2D P colony is given by the state of the environment - matrix of type $m \times n$ with multisets of objects over $A - \{e\}$ as its elements, and

by the state of all agents - pairs of objects from alphabet A and the coordinates of the agents. An initial configuration is given by the definition of the 2D P colony.

The computational step consists of three parts. The first part lies in determining the applicable set of programs according to the actual configuration of the P colony. There are programs belonging to all agents in this set of programs. In the second part we have to choose one program corresponding to each agent from the set of applicable programs. There is no collision between the communication rules belonging to different programs. The third part is the execution of the chosen programs.

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

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

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

Another way to determine the result of the computation is to take into account not only the number of objects but also their location. The result could then be a grayscale image, a character string or a number that is dependent on both the number and pposition of the objects (for example, $g = \sum_{j=0}^{n-1} \left(\sum_{i=0}^{m-1} f(i,j) \right) \cdot n^i$, where $f(i,j)$ is the number of copies of object f in the $[i,j]$-cell).

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

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

These measures can be observed both for the individual steps of the computation and the computation as a whole.

4 The issue of the flow of liquid over the surface

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

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

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

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

5 Problem solution

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

5.1 Preparation of maps

Map data is obtained from the geographic information system (GIS) and processing system ArcGIS. We use the map data for the Czech Republic called the digital model of the terrain in scale 1: 25 000 (DMÚ25).

Raster graphics images are probably the most appropriate format for modelling real-world phenomena in the field of GIS. To process this format, many tools were created and can be used for performing various analyses. A raster image is composed of a regular network of cells, usually in a square shape, to which values of displayed properties can be assigned independently. More information about GIS and image processing the reader can find in [3] and about geosimulation in [10].

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

Basic classification algorithms to calculate the runoff:

- Single flow direction (SFD) - each pixel of the liquid flows in one direction only (toward neighbour in the direction of the largest gradient). Each pixel belongs to only one basin.

- Multiple flow direction (MFD) - fluid can flow out of each pixel in multiple directions, maximum of eight. In the case of MFD a unit volume flow is fairly distributed among all lower neighbours. The MFD may include the pixel to multiple basins.

There is implemented a tool for calculating the flow direction in ArcGIS software, called simply Flow direction. Flow Direction tool works as a simple flow direction (SFD). After its execution integer raster file is created that specifies the flow direction for each cell. Every cell can reach value ranging from 1 to 255.

Eight basic directions of the flow are represented by the numbers $1, 2, 4, 8, 16$, 32, 64 and 128 (see Table 1). Other directions are generated as sums of values of the basic directions.

32	64	128
16	↖ ↑ ↗ ← →	1
	↙ ↓ ↘	
8	4	2

Table 1. The numbers of eight basic directions

When creating the model, we used the test data to propose group of programs. The final visualization is based on data from DMÚ 25.

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

direction	→	←	↑	↓	↘	↙	↗	↖
symbol	a	E	i	m	q	u	y	2

Table 2. The coding table

The first processed map is map without drainless area and its size is 20×12 and it is shown on the Table 3. Transcoded symbols are shown on the Table 4.

5.2 Definition of the agent

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

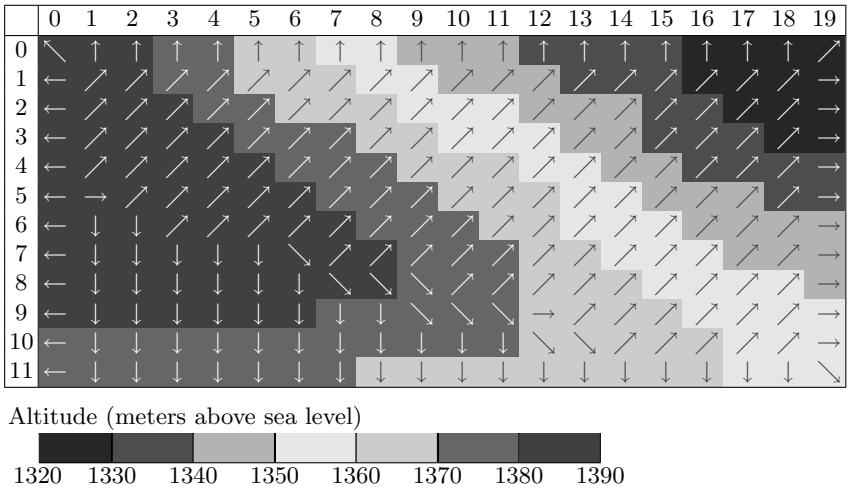


Table 3. Processed map

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	2	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	i	y
1	E	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
2	E	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
3	E	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
4	E	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
5	E	a	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
6	E	m	m	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	y	a
7	E	m	m	m	m	m	q	y	y	y	y	y	y	y	y	y	y	y	y	a
8	E	m	m	m	m	m	m	q	q	q	y	y	y	y	y	y	y	y	y	a
9	E	m	m	m	m	m	m	m	m	q	q	q	a	y	y	y	y	y	y	a
10	E	m	m	m	m	m	m	m	m	m	m	m	q	q	y	y	y	y	y	a
11	E	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	t

Table 4. Transcoded symbols

Each of the objects carries the information about the state of the agent. The first object has information about the activity of the agent. At this stage of the simulation it is the information that the agent "flows" down the terrain or it is still inactive (belonging to the rainfall that have not fall). The second object stores information about the previous direction of flow. This information can further modify the way of the agent as inertia.

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

direction	$\rightarrow$	$\leftarrow$	$\uparrow$	$\downarrow$	$\searrow$	$\swarrow$	$\nearrow$	$\nwarrow$
symbol	9	8	6	7	D	D	U	U
symbol	L	K	H	I	I	I	H	H

The inertia of crossing directions is modified because of longer distance between the centres of the cells.

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

$$(1)\left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ e \rightarrow 9 \right\rangle;\ (2)\left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ e \rightarrow 8 \right\rangle;$$

$$(3)\left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow;\ e \rightarrow 6 \right\rangle;\ (4)\left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow;\ e \rightarrow 7 \right\rangle;$$

$$(5)\left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ e \rightarrow D \right\rangle;\ (6)\left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ e \rightarrow D \right\rangle;$$

$$(7)\left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ e \rightarrow U \right\rangle;\ (8)\left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ e \rightarrow U \right\rangle;$$

In the case of programs (5) and (6) (resp. (7) and (8)) it is neccessary to take one step down (resp. up).

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

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

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

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

$$(15)\left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ H \rightarrow 9 \right\rangle;\ (16)\left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ H \rightarrow 8 \right\rangle;$$

$$(17)\left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow;\ H \rightarrow \mathbf{U} \right\rangle;\ (18)\left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow;\ H \rightarrow 7 \right\rangle;$$

$$(19)\left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ H \rightarrow D \right\rangle;\ (20)\left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ H \rightarrow D \right\rangle;$$

$$(21)\left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ H \rightarrow U \right\rangle;\ (22)\left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ H \rightarrow U \right\rangle;$$

$$(23)\left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ I \rightarrow 9 \right\rangle;\ (24)\left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ I \rightarrow 8 \right\rangle;$$

$$(25)\left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow;\ I \rightarrow 6 \right\rangle;\ (26)\left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow;\ I \rightarrow 7 \right\rangle;$$

$$(27)\left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ I \rightarrow D \right\rangle;\ (28)\left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ I \rightarrow D \right\rangle;$$

$$(29)\left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ I \rightarrow U \right\rangle;\ (30)\left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ I \rightarrow U \right\rangle;$$

$$(31)\left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ J \rightarrow 9 \right\rangle;\ (32)\left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ J \rightarrow L \right\rangle;$$

$$(33)\left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow;\ J \rightarrow 6 \right\rangle;\ (34)\left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow;\ J \rightarrow 7 \right\rangle;$$

$$(35)\left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ J \rightarrow 7 \right\rangle;\ (36)\left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ J \rightarrow D \right\rangle;$$

$$(37)\left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ J \rightarrow 6 \right\rangle;\ (38)\left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ J \rightarrow U \right\rangle;$$

$$(39)\left\langle \begin{bmatrix} * & * & * \\ * & a & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ K \rightarrow N \right\rangle;\ (40)\left\langle \begin{bmatrix} * & * & * \\ * & E & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ K \rightarrow 8 \right\rangle;$$

$$(41)\left\langle \begin{bmatrix} * & * & * \\ * & i & * \\ * & * & * \end{bmatrix} \rightarrow \Uparrow;\ K \rightarrow 6 \right\rangle;\ (42)\left\langle \begin{bmatrix} * & * & * \\ * & m & * \\ * & * & * \end{bmatrix} \rightarrow \Downarrow;\ K \rightarrow 7 \right\rangle;$$

$$(43)\left\langle \begin{bmatrix} * & * & * \\ * & q & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ K \rightarrow D \right\rangle;\ (44)\left\langle \begin{bmatrix} * & * & * \\ * & u & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ K \rightarrow 7 \right\rangle;$$

$$(45)\left\langle \begin{bmatrix} * & * & * \\ * & y & * \\ * & * & * \end{bmatrix} \rightarrow \Rightarrow;\ K \rightarrow U \right\rangle;\ (46)\left\langle \begin{bmatrix} * & * & * \\ * & 2 & * \\ * & * & * \end{bmatrix} \rightarrow \Leftarrow;\ K \rightarrow 6 \right\rangle;$$

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

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

A source of water is placed into cells $[5,6],[5,7],[6,6],[6,7]$. In every source cell there are four agents. To simulate rain all agents are not active in the initial configuration. Only one agent has the configuration of Xe in each cell. The next three become active always in two computational steps. The numbers of active agents in the environment are shown in the Tables 5 - 13.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 5. The numbers of active agents in the initial configuration

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 6. The numbers of active agents after 2 step of computation

The obtained results were compared with data that are listed in the master thesis [9]. This work is devoted to the simulation flow of liquid over the Earth's surface using cellular automata.

In work [9], the author devotes a great deal of time preparing data for cellular automaton, calculates not only the direction of flow but also the number and direction of outflows and inflows into the cell. After then cellular automaton starts

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 7. The numbers of active agents after 4 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 8. The numbers of active agents after 6 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Table 9. The numbers of active agents after 8 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Table 10. The numbers of active agents after 10 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Table 11. The numbers of active agents after 12 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0

Table 12. The numbers of active agents after 14 step of computation

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 13. The numbers of active agents after 16 step of computation

working. The results are shown in Figure 1. Cells shown white are cells that contain water.

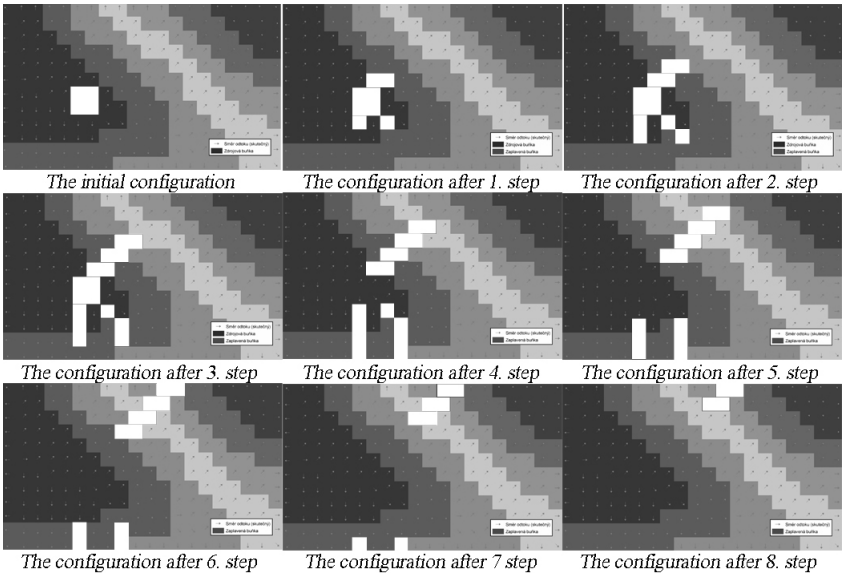


Fig. 1. Cellular automaton

If we compare the simulation process using 2D P colonies and using cellular automaton we obtain the following:

- Cells are flooded with water in the same time sequence.

- 2D P colony needs twice as many steps for the computation as cellular automaton.
- In 2D P colonies we do not need pre-treatment of data, output from the tool Flow direction is sufficient.

6 Conclusion and future work

The aim of this paper was to analyse the situation and to create a 2D model P colonies that would simulate the flow of liquid over the Earth's surface, a phenomenon called Surface runoff. This process is very common in nature and accumulation of water leads to flash flooding or floods in general. Flow down of water on the surface is influenced by many factors: the surface slope, soil saturation, temperature, humidity, size of source and lots of others. The first condition was partially met. Water flows down the surface in the right direction. In the case of places which are depressions, the possibility of overflow of the "tank" and the subsequent redistribution have not been implemented. The way of solving the problem is obvious and it is similar to absorption of water into the ground - certain amount of objects, which will represent the amount of water to be absorbed, needs to be added. Agents which have stopped, consumed the object and sets the direction of the flow using flow direction in neighbouring cells if it is possible.

Remark 1.
This work was partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by SGS/7/2011 and by project OPVK no. CZ.1.07/2.2.00/28.0014.

References

1. Cienciala, L., Ciencialová, L., Perdek, M.: 2D P colonies. In Csuhaj-Varjú et al. (eds.). CMC 2012, Springer, LNCS 7762, 2013, pp. 161–172.
2. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, G.: *Cells in environment: P colonies*, Multiple-valued Logic and Soft Computing, 12, 3-4, 2006, pp. 201–215.
3. Eastman, R. J.: *IDRISI Andes Guide to GIS and Image Processing*, Clerk Lab. Clerk University, Worcester, MA, USA, 2006.
4. Kelemen, J., Kelemenová, A.: *On P colonies, a biochemically inspired model of computation.* Proc. of the 6^{th} International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH, Hungary, 2005, pp. 40–56.
5. Kelemen, J., Kelemenová, A., Păun, Gh.: *Preview of P colonies: A biochemically inspired computing model.* Workshop and Tutorial Proceedings, Ninth International Conference on the Simulation and Synthesis of Living Systems, ALIFE IX (M. Bedau at al., eds.) Boston, Mass., 2004, pp. 82–86.
6. Păun, Gh.: *Computing with membranes.* Journal of Computer and System Sciences 61, 2000, pp. 108–143.

7. Păun, Gh.: *Membrane computing: An introduction.* Springer-Verlag, Berlin, 2002.
8. Păun, Gh., Rozenberg, G., Salomaa, A.: *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2009.
9. Pustějovská, L.: Implementation of a cellular automaton capable of simulating flow of liquid over terrestrial surface, master thesis (in czech), VŠB-Technical University of Ostrava, Czech Republic, 2008.
10. Torrents, B.: *Geosimulation.* John Wiley & Sons, 2004.
11. P systems web page: http://psystems.disco.unimib.it

Scenario Based P Systems

Gabriel Ciobanu[1] and Dragoş Sburlan[2]

[1] Romanian Academy, Institute of Computer Science
Blvd. Carol I no. 8, 700505 Iasi, Romania
E-mail: gabriel@info.uaic.ro
[2] Faculty of Mathematics and Informatics
Ovidius University of Constanta, Romania
E-mail: dsburlan@univ-ovidius.ro

Summary. In this paper we define and study *Scenario Based P Systems*, a model of computation inspired by the metabolic pathways and networks. Starting from the classical definition of P systems with symbol objects and multiset rewriting rules, we define regular expressions able to capture the causal dependencies among different executions of the rules. The results show the computational power of this model.

1 Motivation

Metabolic pathways are sequences of biochemical reactions occurring inside the living cell which are involved in cell's energy management and in the synthesis of structural components. Because in such sequences participate many biochemicals (the metabolites), metabolic pathways are usually very complex. Moreover, many distinct pathways co-exist inside the cell and they form what is called the metabolic network. A metabolic pathway illustrate all the changes in time by which an initial molecule is transformed into another product. Usually, the products of one biochemical reaction constitute the substrate for the next biochemical reaction. The resulting product can be used by the cell to start another metabolic pathway, or it can be stored for a later use. Depending on the needs of the cell and on the availability of the substrate, these metabolic pathways are started.

In a broader perspective, the principle of causality plays the main role in finding/expressing metabolic pathways which connect parts of a metabolic network (our understanding of phenomena happening inside the cells is based on the causal relations existing among cell's "observable" events). In this context, one can consider the biochemical reactions as causal consequences where the input metabolites can cause the output metabolites. Moreover, there might be a certain temporal order by which any later event is determined by the earlier one, and which is not necessarily related with the involved metabolites.

This paper explores the concept of causality in the P system framework having as inspiration the biochemical dynamics expressed by the metabolic pathways. Its

goal is to capture the causal dependencies existing among the executions of rules, while abstracting away other aspects. In the membrane computing literature there were several attempts to formalize causal semantics [3], [4], [2], and [8], most of them proposing a notion of causality based on the temporal order of single rule application. Our new approach introduces regular expressions to define the causal relation between the executions of rules; the time between the moments when these rules compete for objects can be also specified in the definition of regular expressions. Therefore, we define scenarios as a method to model different possible evolutions in the metabolic networks, and their causal relationships.

2 Preliminaries

We recall some notions and results from the classical theory of formal languages [5].

An *ET0L system* is a construct $H = (V, T, \omega, \Delta)$, where V is an alphabet, $T = \{T_1, \dots T_m\}$, $m \geq 1$, such that T_i, $1 \leq i \leq m$, are finite complete sets of rules (tables) of non-cooperative rules over V, $\omega \in V^*$ is the axiom, and Δ is the terminal alphabet. In a derivation step, all the symbols present in the current sentential form are rewritten using one (nondeterministically chosen) table. The language generated by H consists of all the strings over Δ which can be generated in this way by starting from ω.

Lemma 1. *For each $L \in ET0L$ there is an extended tabled Lindemayer system $H = (V, T, \omega, \Delta)$ with two tables ($T = \{T_1, T_2\}$) generating L, such that for each $a \in \Delta$ if $a \to \alpha \in T_1 \cup T_2$ then $\alpha = a$.*

A *register machine* is a formal construct $M = (n, \mathcal{P}, l_0, l_h)$ where $n \geq 1$ is the number of registers, $\mathcal{P}$ is a finite set ($card(\mathcal{P} = k)$ of instructions bijectively labeled by elements from the set $B = \{l_0, \dots, l_{k-1}\}$, $l_0 \in B$ is the initial label, and $l_h \in B$ is the final label. The instructions of M are of the following types:

- $l_1 : (add(r), l_2, l_3$ where $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $1 \leq r \leq n$, increments the value stored by the register r and non-deterministically proceeds to the instruction labeled by l_2 or l_3;
- $l_1 : (sub(r), l_2, l_3)$ where $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq r \leq n$, if the value stored by register r is 0 then proceeds to the instruction labeled by l_3, otherwise decrements the value stored by register r and proceeds to the instruction labeled by l_2;
- $l_h : halt$ stops the machine.

A register machine is *deterministic* if $l_2 = l_3$ in all its add instructions.

A non-deterministic register machine M starts with all registers being empty and runs the program $\mathcal{P}$, starting from the instruction with the label l_0. Considering the content of register 1 for all possible computations of M which are ended

by the execution of the instruction labeled l_h, one gets the set $N(M) \subseteq I\!N$ – the set generated by M.

A deterministic register machine M accepts a natural number by starting with the number as input in register 1, with all other registers being empty. M runs the program $\mathcal{P}$, starting from the instruction with the label l_0, and if it reaches the instruction l_h then it halts, accepting the number.

It is known the following result ([6]).

Theorem 1. *For any recursively enumerable set $Q \subseteq I\!N$ there exists a non-deterministic register machine with 3-registers generating Q such that when starting with all registers being empty, M non-deterministically computes and halts with n in register 1, and registers 2 and 3 being empty iff $n \in Q$.*

If FL is a family of languages, then by NFL we denote the family of length sets of languages in FL. We denote by REG, CF, $ET0L$, and RE the family of regular, context-free, extended tabled interactionless Lindemayer, and recursive enumerable languages, respectively. It is know that

$$NREG = NCF \subset NET0L \subset NRE.$$

The non-semilinear set $\{2^n \mid n \geq 0\} \in NETOL \setminus NCF$.

3 Scenario Based P Systems

The principle of causality implies a certain temporal order between some events and by which any later event is determined by the earlier one. However, the actual time elapsed between the occurrence of consecutive events that are in a given causality relation is not important. Based on these considerations we introduce a new model of P systems that use regular expressions to express a certain causal dependence relation between the execution of the rules.

The reader is assumed to be familiar with the basic notions, notations, and functioning of P Systems.

A *Scenario Based P System* (a SBP system, for short) of degree $m \geq 1$ is a construct $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, E_1, \ldots E_m, i_0)$, where

- O is an alphabet of *objects*;
- $C \subseteq O$ is the set of *catalysts*;
- μ is a tree structure of $m \geq 1$ uniquely labelled *membranes* (which delimit the regions of Π); usually, the set of labels is $\{1, \ldots, m\}$;
- $w_i \in O^*$, for $1 \leq i \leq m$, are multisets of objects which are initially present in the regions of μ (as indicated by the index);
- R_i, $1 \leq i \leq m$, is a finite set of labelled multiset rewriting rules. The set of labels is denoted by $\mathcal{L}_i$ and each label in $\mathcal{L}_i$ uniquely identifies a rule from R_i; in addition, $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$ for all $i \neq j$, $1 \leq i, j \leq m$. A rule from R_i is written as $l : \alpha \to \beta$ where $l \in \mathcal{L}_i$ and $\alpha, \beta \in O^*$. In particular, a rule can be *non-cooperative* $l : a \to v$ or *catalytic* $l : ca \to cv$, where $l \in \mathcal{L}_i$, $a \in O \setminus C$, $v \in ((O \setminus C) \times \{here, out, in\})^*$, and $c \in C$;

• E_i, $1 \leq i \leq m$, is a finite set of regular expressions over $\mathcal{L}_i \cup \{d\}$, where d is a special symbol (the "delay" symbol), $d \notin \bigcup_{i=0}^{m} \mathcal{L}_i$; moreover, if $e \in E_i$ then $L(e) \subseteq (\mathcal{L}_i \cup \{d\})^* \mathcal{L}_i (\mathcal{L}_i \cup \{d\})^*$ (that is, any word in $L(e)$ contains at least one symbol from $\mathcal{L}_i$);

• $i_0 \in \{1, \ldots, m\}$ is the label of the *output region* of Π.

A *configuration* of Π is a vector $(\alpha_1, \ldots, \alpha_m)$, where $\alpha_i \in O^*$, $1 \leq i \leq m$, is the multiset of objects present in the region i of Π. The *initial configuration* of Π is the vector $C_0 = (w_1, \ldots, w_m)$.

Let $E_i = \{e_{(i,1)}, \ldots, e_{(i,s_i)}\}$, where $1 \leq i \leq m$ and such that $s_i \geq 1$; in addition, let $L_{(i,1)}, \ldots, L_{(i,s_i)}$ be the corresponding regular languages. A word $l_0 \ldots l_t \in L_{(i,j)}$, $1 \leq i \leq m$, $1 \leq j \leq s_i$ (a finite sequence of symbols from $\mathcal{L}_i \cup \{d\}$) is called a *scenario* and illustrates the fact that the corresponding rules (if there exists such corresponding rules; recall that d is not associated with any rule) will be applied (if possible) in the implicit order of symbols. Given a multiset of objects w, a scenario $l_0 \ldots l_t$ is *applicable* to w if the rule having the label l_0 is applicable to w or $l_0 = d$; similarly, a scenario is *started* if the rule labeled with l_0 is applied to w or $l_0 = d$.

As usually in the P system framework, a computation of Π is a sequence of configurations (possibly infinite) $C_0, C_1, \ldots, C_k, C_{k+1}, \ldots$. Given a configuration $C_k = (w_{(k,1)}, \ldots, w_{(k,m)})$, then one gets the next configuration $C_{k+1} = (w_{(k+1,1)}, \ldots, w_{(k+1,m)})$ by applying on each multiset $w_{(k,i)}$, $1 \leq i \leq m$, some rules from R_i in a nondeterministic, maximal parallel manner and with competition on objects; these rules are selected according with the conditions described below.

For a scenario $l_0 \ldots l_t$ that is started in configuration C_k, the rule labeled l_i, $1 \leq i \leq t$, $l_i \neq d$, compete for objects in configuration C_{k+i} iff the rules labeled l_{i-j}, $1 \leq j \leq i$, $l_{i-j} \neq d$ were applied (won the competitions) in the corresponding configurations C_{k+i-j}. A started scenario is said to be *entirely applied* if the rules corresponding to all labels were applied in the given order, in consecutive configurations; in case there exists a rule labeled l_i, $1 \leq i \leq t$, $l_i \neq d$, that lost the competition on objects or if the rule cannot be applied then the started scenario is said to be *interrupted*; the executions of the remaining rules (in case they exist, that is, not all the remaining symbols in the scenario are d) in the subsequent configurations are dropped.

In any configuration, new scenarios from each $L_{(i,j)}$, $1 \leq i \leq m$, $1 \leq j \leq s_i$, are nondeterministically selected for applications.Given such a scenario and a configuration C_k, if the first label of rule appears in the scenario on position $l \geq 0$ (the first symbols being all d) then the corresponding rule will compete for objects with other rules (from the scenarios in progress) after l computational steps. For each multiset $w_{(k,i)}$ from C_k, $1 \leq i \leq m$, there might exist new scenarios, scenarios in progress, and interrupted scenarios, which determine the rules to be applied in order to obtain the next configuration C_{k+1}.

A computation of Π is a halting one if no rule can be applied (all the started scenarios are interrupted and no matter how a new scenario is selected for application it becomes interrupted at the first symbol corresponding to a rule) in the last configuration (the *halting configuration*). The result of a halting computation is the number of objects from O contained in the output region i_0, in the halting configuration. A non-halting computation yields no result. By collecting the results of all possible halting computations of a given P system Π, one gets $N(\Pi)$ – the set of all natural numbers generated by Π.

The family of all sets of numbers computed by SBP systems with at most m membranes and with a list of features f is denoted by $NOSBP_m(f)$. The features considered in this paper are *ncoo* (P systems using only non-cooperative rules) and cat_k (P systems using non-cooperative rules and catalytic rules with at most k catalysts).

The above definition can be relaxed such that in a halting configuration one counts only the symbols from a given alphabet $\Sigma \subseteq O$.

Given a scenario based P system Π with $m > 1$ membranes and using the features f, it is easy to construct an equivalent scenario based P system with the same features but having only one region. This can be accomplished by a simple encoding of the region labels into the objects and expressing the rules accordingly [1].

Theorem 2. $NOSBP_m(cat_1) = NRE$, $k \geq 1$.

Proof. The inclusion $NOSBP_m(cat) \subseteq NRE$ is supposed to be true by invoking the Church-Turing thesis. The opposite inclusion can be shown to be true by simulating the computation of an arbitrary register machine $M = (n, \mathcal{P}, l_0, l_h)$ with a scenario based P system $\Pi = (O, C, \mu, w_1, R_1, E_1)$ where

$$\begin{aligned} O &= \{a_i \mid 1 \leq i \leq n\} \\ &\cup \{l_1, l_2 \mid l_1 : (add(r), l_2) \in \mathcal{P}\} \\ &\cup \{l_1, l_2, l_3, \overline{l_1}, \overline{l_2}, S, \overline{S}, \overline{\overline{S}}, X \mid l_1 : (sub(r), l_2, l_3)\} \\ C &= \{c\}, \quad \mu = [\,]_1, \quad w_1 = l_0. \end{aligned}$$

The set of rules R_1 and the set of regular expressions E_1 are defined as follows:

- for each register machine instruction $l_1 : (add(r), l_2)$, the rule $r_{l_1} : l_1 \to a_r l_2$ is added to R_1 and the regular expression r_{l_1} is added to E_1.
- for each register machine instruction $l_1 : (sub(r), l_2, l_3)$, the next rules are added to R_1:

$$\begin{aligned} r_{(l_1,1)} : l_1 \to \overline{l_1}S \ &, \ r_{(l_1,2)} : ca_r \to cX \\ r_{(l_1,3)} : X \to \lambda \ &, \ r_{(l_1,4)} : \overline{l_1} \to \overline{l_2} \\ r_{(l_1,5)} : S \to \overline{S} \ &, \ r_{(l_1,6)} : \overline{S} \to \overline{\overline{S}} \\ r_{(l_1,7)} : \overline{\overline{S}} \to \lambda \ &, \ r_{(l_1,8)} : \overline{l_1} \to l_3 \\ r_{(l_1,9)} : \overline{l_2} \to l_2 \ &. \end{aligned}$$

The regular expressions $r_{(l_1,1)}r_{(l_1,2)}$, $r_{(l_1,3)}r_{(l_1,4)}$, $r_{(l_1,5)}r_{(l_1,6)}$, $r_{(l_1,7)}r_{(l_1,8)}$, $r_{(l_1,9)}$ are added to E_1.

- for the register machine instruction $l_1 : halt$, the rule $r_{l_1} : l_1 \to \lambda$ is added to R_1 and the regular expression r_{l_1} is added to E_1.

The simulation of the register machine M by the scenario based P system Π proceeds as follows. At a certain moment during the computation of M the values stored by the registers are $t_1, \ldots, t_r, \ldots t_n$, and the label of the instruction that has to be executed is l_1. Correspondingly, the multiset contained in the region of Π is $a_1^{t_1} \ldots a_r^{t_r} \ldots a_n^{t_n} l_1 c$ (that is, the value t_r stored by the register r of M is modeled in this simulation as the multiplicity of the object a_r in a configuration of Π).

If l_1 is the label of an addition instruction $l_1 : (add(r), l_2)$, then Π is executing the scenario described by r_{l_1}, that is the rule $l_1 \to a_r l_2$ is applied. As a consequence the next configuration of Π will be $a_1^{t_1} \ldots a_r^{t_r+1} \ldots a_n^{t_n} l_2 c$ (which indicates that the addition instruction was simulated correctly).

If l_1 is the label of a subtraction instruction $l_1 : (sub(r), l_2, l_3)$, then Π is executing the scenario described by $r_{(l_1,1)}r_{(l_1,2)}$. Consequently, because in this scenario the rule $r_{(l_1,1)} : l_1 \to \overline{l_1} S$ is executed firstly, the next configuration of Π is described by the multiset $a_1^{t_1} \ldots a_r^{t_r} \ldots a_n^{t_n} \overline{l_1} S c$. Because the object S appeared in the multiset, then the scenario $r_{(l_1,5)}r_{(l_1,6)}$ will be started. Next, two cases might happen:

- if $t_r > 0$ then the rule $r_{(l_1,2)} : ca_r \to cX$ is executed (the second rule from the already started scenario $r_{(l_1,1)}r_{(l_1,2)}$) in the same moment with the rule $r_{(l_1,5)} : S \to \overline{S}$ (from scenario $r_{(l_1,5)}r_{(l_1,6)}$). The configuration of Π becomes $a_1^{t_1} \ldots a_r^{t_r-1} \ldots a_n^{t_n} \overline{l_1} X \overline{S} c$. Next, the scenario $r_{(l_1,3)}r_{(l_1,4)}$ is started. It follows that the rules $r_{(l_1,6)} : \overline{S} \to \overline{\overline{S}}$ (from scenario $r_{(l_1,5)}r_{(l_1,6)}$) and $r_{(l_1,3)} : X \to \lambda$ (from scenario $r_{(l_1,3)}r_{(l_1,4)}$) are simultaneously executed; the configuration of Π becomes $a_1^{t_1} \ldots a_r^{t_r-1} \ldots a_n^{t_n} \overline{l_1}\, \overline{\overline{S}} c$. Finally, the scenario $r_{(l_1,7)}r_{(l_1,8)}$ is started. Accordingly, the rules $r_{(l_1,4)} : \overline{l_1} \to \overline{l_2}$ (from scenario $r_{(l_1,3)}r_{(l_1,4)}$) and $r_{(l_1,7)} : \overline{\overline{S}} \to \lambda$ (from scenario $r_{(l_1,7)}r_{(l_1,8)}$) are executed in the same time; the configuration of Π becomes $a_1^{t_1} \ldots a_r^{t_r-1} \ldots a_n^{t_n} \overline{l_2} c$. Next, the scenario $r_{(l_1,7)}r_{(l_1,8)}$ interrupts its execution (the object $\overline{l_1}$ is not anymore present in the current configuration of Π, hence the rule $r_{(l_1,8)}$ cannot be executed); the scenario $r_{(l_1,9)}$ starts its execution and this yields to the configuration $a_1^{t_1} \ldots a_r^{t_r-1} \ldots a_n^{t_n} l_2 c$ (which indicates a correct simulation of the register machine subtraction instruction in the case when register r is not empty);
- if $t_r = 0$ then the rule $r_{(l_1,2)} : ca_r \to cX$ cannot be executed and consequently the object X (which triggered the execution of the scenario $r_{(l_1,3)}r_{(l_1,4)}$) is not produced anymore. However, in this case the scenario $r_{(l_1,5)}r_{(l_1,6)}$ is started and the rules $r_{(l_1,5)} : S \to \overline{S}$ and $r_{(l_1,6)} : \overline{S} \to \overline{\overline{S}}$ are executed in consecutive configurations of Π. The resulting configuration becomes $a_1^{t_1} \ldots a_r^{t_r} \ldots a_n^{t_n} \overline{l_1}\, \overline{\overline{S}} c$. Next, the scenario $r_{(l_1,7)}r_{(l_1,8)}$ starts its execution and after two computational

steps the resulting multiset becomes $a_1^{t_1} \dots a_r^{t_r} \dots a_n^{t_n} l_3 c$ (which indicates a correct simulation of the register machine subtraction instruction in the case when register r is empty).

In case the configuration of Π is $a_1^{t_1} \dots a_n^{t_n} l_1 c$ where the object l_1 corresponds to the label of the register machine halting instruction, then the scenario r_{l_1} is started (the rule $r_{l_1} : l_1 \to \lambda$ is executed). The next configuration of Π becomes $a_1^{t_1} \dots a_n^{t_n} c$ and the computation stops.

Consequently, since the computation of M was correctly simulated by Π and the register machines are computational universal, we have $NOSBP_m(cat) \supseteq NRE$.

4 A More Realistic Scenario

A particular case, interesting from a biological point of view, is when all possible scenarios used by a SBP system Π in any region i are of type $d^{l_1} w_1 d^{l_2} w_2 d^{l_3} \dots d^{l_k} w_k d^{l_{k+1}}$, where $w_i \in \mathcal{L}_i^+$, $l_i \in I\!N$, $1 \leq i \leq k+1$. We will consider that the regular expressions from each E_i, $1 \leq i \leq m$, are of type $d^* \alpha_1 d^* \alpha_2 d^* \dots d^* \alpha_k d^*$ where each α_j, $1 \leq j \leq k$, are regular expressions over $\mathcal{L}_i$ which use only the grouping and the Boolean OR operations in their definitions (consequently, each α_i indicates a finite language). Such regular expressions and their corresponding scenarios suggest that one knows the application order of the rules but does not know when their executions will actually happen.

Let $E_i = \{e_{(i,1)}, \dots, e_{(i,s_i)}\}$, where $1 \leq i \leq m$, $s_i \geq 1$, and consider the corresponding regular languages $L_{(i,j)}$, $1 \leq i \leq m$, $1 \leq j \leq s_i$. In the above conditions, for a scenario $x = d^{l_1} w_1 d^{l_2} w_2 d^{l_3} \dots d^{l_k} w_k d^{l_{k+1}} \in L_{(i,j)}$ we define $deg(x) = \max\limits_{1 \leq i \leq k} \{|w_i|\}$.

For a given SBP system Π we define the *degree of synchronization*

$$deg(\Pi) = max\{deg(s) \mid (\exists) 1 \leq i \leq m, 1 \leq j \leq s_i \text{ such that } s \in L_{(i,j)}\}.$$

In this case, the family of all sets of numbers computed by such SBP systems with the feature $f \in \{ncoo, cat\}$ and of synchronization degree at most n will be denoted by $NOSBP_m^{d_n}(f)$.

The following example shows how to generate a non-semilinear set of numbers with a SBP systems with non-cooperative rules and with a synchronization degree 1.

Example 1. Let $\Pi_1 = (O, C, \mu, w_1, R_1, E_1, i_0)$ such that

$$\begin{aligned} &O = \{a, b\}; \quad C = \emptyset; \quad \mu = [\,]_1; \quad w_1 = ab; \\ &R_1 = \{r_1 : b \to b,\ r_2 : a \to aa,\ r_3 : b \to \lambda\}; \\ &E_1 = \{d^* r_1 d^* r_2 d^* r_3 d^*\};\ i_0 = 1. \end{aligned}$$

The system Π_1 computes the set $\{a^{2^n} \mid n \geq 1\}$, the well know non–semilinear set from $NET0L \setminus NCF$. The regular expression used in the definition of Π can

be simplified such that $E_1 = \{r_1 d^* r_2 d^* r_3\}$.Using this simplification, the system performs the computation as follows. In the first configuration $C_0 = (ab)$, a scenario $r_1 d^{n_1} r_2 d^{n_2} r_3$, $n_1, n_2 \geq 0$, is selected for application. This means that the rule labeled r_1 is applied in the configuration C_0 because there exists an object b; the rule labeled r_2 will compete for objects after n_1 computational steps (where n_1 can be any natural number) and if it is applied, it will double the objects a. Finally, after the next n_2 computational steps the rule r_3 compete for the object b, and if it is applied then it will delete the objects b (and consequently the selection of a scenario for an application is blocked). I all these computational steps (between the starting of the first scenario and the application of its last rule labeled r_3) new scenarios are selected for applications. Each of them will double the number of symbols a. Consequently, the system computes the set $\{a^{2^n} \mid n \geq 1\}$.

Theorem 3. *For any $n \geq 2$,*

$$NOSBP_m(f) \supseteq NOSBP_m^{d_n}(f) \supseteq NOSBP_m^{d_n-1}(f), f \in \{ncoo, cat_k\}, k \geq 1.$$

The following result shows the relation between the family of all sets of numbers computed by SBP systems with at most m membranes and using only non-cooperative rules and the family of length sets of context-free languages.

Proposition 1. $NOSBP_m^{d_1}(ncoo) \supset NCF = NREG$.

Proof. From the above observation one knows that $NOSBP_m(ncoo) = NOSBP_1(ncoo)$, hence in our proof we will use a scenario based P system with one region. Let $G = (N, T, P, S)$ be a context-free grammar and let $P = \{r_1, \ldots, r_k\}$ be the set of labeled productions. Then one can construct an equivalent scenario based P system $\Pi = (O, C, [\,]_1, R_1, E_1, i_0 = 1)$ defined by:

$$\begin{aligned} &O = N \cup T, \quad C = \emptyset, \\ &R_1 = P \cup \{r_A : A \to A \mid A \in N\}, \\ &E_1 = \{d^* r d^* \mid r \in P\} \cup \{d^* r_X d^* \mid X \in N\}. \end{aligned}$$

At any moment during the computation of Π scenarios from the languages indicated by the regular expressions from E_1 can be started. A scenario of type $d^k r d^p$, $r = A \to \alpha \in R_1$, $k, p \geq 0$, simulates the application of the context-free production $A \to \alpha \in P$. In order to prevent the maximal parallel rewriting of the object A in a given configuration of Π, scenario of type $d^k r_A d^p$, $r_A = A \to A \in R_1$ are employed. It follows that there exist a computation of Π where for any configuration exactly one object $A \in N$ is rewritten.

Thus, Π correctly simulates G, and so we conclude that $NOSBP_m^{d_1}(ncoo) \supseteq NCF = NREG$. The strict inclusion follows easily from Example 1.

The length set of any language generated by an ET0L system can be generated by a SBP systems with non-cooperative rules and synchronization degree 2.

Theorem 4. $NOSBP_m^{d_2}(ncoo) \supseteq NET0L$.

Proof. To prove this result, we simulate the computation of a arbitrary ET0L system using a SBP system with non-cooperative rules and having the synchronization degree 2. Without loss of generality, let $H = (V, T, \omega, \Delta)$ be an ET0L system, such that $V = \{a_1, \ldots, a_k\}$, $\Delta = \{a_1, \ldots, a_p\}$, $p \leq k$, and $T = \{T_1, T_2\}$, where

$$T_1 = \{a_i \to \alpha_{1,j} \mid 1 \leq i \leq k, 1 \leq j \leq l_{1,i}\},$$
$$T_2 = \{a_i \to \alpha_{2,j} \mid 1 \leq i \leq k, 1 \leq j \leq l_{2,i}\}.$$

Then we construct the SBP system $\Pi = (O, C, \mu, w_1, R_1, E_1, i_0 = 1)$ that simulates the computation of H as follows:

$$\begin{aligned} O &= V \cup \{\overline{a} \mid a \in V\} \\ &\quad \cup \{t_{i,j} \mid i \in \{1,2\}, 1 \leq j \leq 2k+1\} \\ &\quad \cup \{t, f, \#\}; \\ C &= \emptyset; \quad \mu = [\,]_1; \quad w_1 = t\omega. \end{aligned}$$

In order to simplify the notation and construction, we will present the regular expressions from E_1 by using directly the rules in their descriptions (and not the labels of the rules). The set of rules R_1 is composed by all the rules appearing in these regular expressions. In addition, the regular expressions will be grouped according to their usage in the simulation.

1. regular expressions/rules used to select a table to be simulated:
   $d^* \;\; t \to t_{1,1}^{max\{l_{1,i} \mid 1 \leq i \leq k\}} X \;\; d^*$
   $d^* \;\; t \to t_{2,1}^{max\{l_{2,i} \mid 1 \leq i \leq k\}} X \;\; d^*$
2. regular expressions/rules used to simulate an application of the table T_1:
   $d^* \;\; t_{1,1} \to t_{1,1} \;\; a_1 \to \overline{\alpha_{1,j}} \;\; d^*$ where $1 \leq j \leq l_{1,1}$
   $d^* \;\; t_{1,1} \to t_{1,2} \;\; a_1 \to \# \;\; d^*$
   $d^* \;\; t_{1,2} \to t_{1,2} \;\; a_2 \to \overline{\alpha_{1,j}} \;\; d^*$ where $1 \leq j \leq l_{1,2}$
   $d^* \;\; t_{1,2} \to t_{1,3} \;\; a_2 \to \# \;\; d^*$
   ...
   $d^* \;\; t_{1,k} \to t_{1,k} \;\; a_k \to \overline{\alpha_{1,j}} \;\; d^*$ where $1 \leq j \leq l_{1,k}$
   $d^* \;\; t_{1,k} \to t_{1,k+1} \;\; a_k \to \# \;\; d^*$

   $d^* \;\; t_{1,k+1} \to t_{1,k+2} \;\; \overline{a_1} \to a_1 \;\; d^*$
   $d^* \;\; t_{1,k+2} \to t_{1,k+3} \;\; \overline{a_2} \to a_2 \;\; d^*$
   ...
   $d^* \;\; t_{1,2k} \to t_{1,2k+1} \;\; \overline{a_k} \to a_k \;\; d^*$
3. regular expressions/rules used to simulate an application of the table T_2:
   $d^* \;\; t_{2,1} \to t_{2,1} \;\; a_1 \to \overline{\alpha_{2,j}} \;\; d^*$ where $1 \leq j \leq l_{2,1}$
   $d^* \;\; t_{2,1} \to t_{2,2} \;\; a_1 \to \# \;\; d^*$
   $d^* \;\; t_{2,2} \to t_{2,2} \;\; a_2 \to \overline{\alpha_{2,j}} \;\; d^*$ where $1 \leq j \leq l_{2,2}$
   $d^* \;\; t_{2,2} \to t_{2,3} \;\; a_2 \to \# \;\; d^*$
   ...
   $d^* \;\; t_{2,k} \to t_{2,k} \;\; a_k \to \overline{\alpha_{2,j}} \;\; d^*$ where $1 \leq j \leq l_{2,k}$

$d^* \quad t_{2,k} \to t_{2,k+1} \quad a_k \to \# \quad d^*$

$d^* \quad t_{2,k+1} \to t_{2,k+2} \quad \overline{a_1} \to a_1 \quad d^*$
$d^* \quad t_{2,k+2} \to t_{2,k+3} \quad \overline{a_2} \to a_2 \quad d^*$
$\ldots$
$d^* \quad t_{2,2k} \to t_{2,2k+1} \quad \overline{a_k} \to a_k \quad d^*$

4. starting over the simulation or ending the simulation:
   $d^* \quad t_{1,2k+1} \to \lambda \quad X \to t \quad d^*$
   $d^* \quad t_{1,2k+1} \to \lambda \quad X \to f \quad d^*$
5. checking if there are "nonterminals" in the last configuration:
   $d^* \quad f \to f_1 \quad a_1 \to \# \quad d^*$
   $d^* \quad f_1 \to f_2 \quad a_2 \to \# \quad d^*$
   $\ldots$
   $d^* \quad f_p \to \lambda \quad a_p \to \# \quad d^*$
   $d^* \quad \# \to \# \quad d^*.$

The SBP system constructed above simulates the computation of an ET0L system as follows. At the beginning of simulation, scenarios from all the languages indicated by the regular expressions from E_1 are started. However, because there is an object t in the initial configuration, only the rules that appear in scenarios from the group 1 can be applied (that is it will be applied either $t \to t_{1,1}^{max\{l_{1,i}|1\leq i\leq k\}}X$ or $t \to t_{2,1}^{max\{l_{2,i}|1\leq i\leq k\}}X$). Let us assume that the rule $t \to t_{1,1}^{max\{l_{1,i}|1\leq i\leq k\}}X$ was executed, hence the table to be simulated is T_1. The number $max\{l_{1,i} \mid 1 \leq i \leq k\}$ of objects $t_{1,1}$ guarantees that any combination of the rules from T_1 which have the same symbol on the left and which are executed at certain moment by H, can be simulated by Π. Consequently, scenarios indicated by the regular expressions

$$d^* \quad t_{1,1} \to t_{1,1} \quad a_1 \to \overline{\alpha_{1,j}} \quad d^* \text{ where } 1 \leq j \leq l_{1,1}$$

are started. In these scenarios the rules of type $a_1 \to \overline{\alpha_{1,j}}$ (which correspond to the rules $a_1 \to \alpha_{1,j} \in T_1$) are applied at a certain moment. However, also the scenarios indicated by the regular expression

$$d^* \quad t_{1,1} \to t_{1,2} \quad a_1 \to \# \quad d^*$$

start their execution; in case the rule $t_{1,1} \to t_{1,2}$ is executed and there are objects a_1 in the region, then the symbol # will be produced and the system Π will never stop (no output). This scenario is used to check if all objects a_1 were rewritten.

The computation continues in the same manner for all the objects from V. After all objects from V were rewritten (i.e., in the current configuration there are only objects from the set $\{\overline{a} \mid a \in V\}$ and object $t_{1,k+1}$), the system Π rewrites back all the objects from the set $\{\overline{a} \mid a \in V\}$ into their corresponding version from V. Scenarios indicated by the following regular expressions are used to complete the task:

$$\begin{array}{c} d^* \quad t_{1,k+1} \to t_{1,k+2} \quad \overline{a_1} \to a_1 \quad d^* \\ d^* \quad t_{1,k+2} \to t_{1,k+3} \quad \overline{a_2} \to a_2 \quad d^* \\ \ldots \\ d^* \quad t_{1,2k} \to t_{1,2k+1} \quad \overline{a_k} \to a_k \quad d^* \end{array}$$

Finally, scenarios from group 4 start, and object X is rewritten either into object t (and Π restarts the computation by simulating the application of another table of H) or into object f (which will be used to check whether the current configuration of Π corresponds to a string computed by H which is formed only by symbols from Δ). In case object f is generated, then scenarios indicated by regular expressions

$$\begin{array}{c} d^* \quad f \to f_1 \quad a_{p+1} \to \# \quad d^* \\ d^* \quad f_1 \to f_2 \quad a_{p+2} \to \# \quad d^* \\ \cdots \\ d^* \quad f_p \to \lambda \quad a_k \to \# \quad d^* \\ d^* \quad \# \to \# \quad d^*. \end{array}$$

are applied and if in the current configuration there exists a symbol from $V \setminus \Delta$, then the symbol # is generated and the computation never stop. Otherwise, the system stops generating a multiset that correspond to a string from $L(H)$.
Consequently, it was proved that $NOSBP_m^{d_2}(ncoo) \supseteq NET0L$.

The following result shows the computation power of SBP systems using non-cooperative and catalytic rules, and the degree of synchronization 3.

Theorem 5. $NOSBP_m^{d_3}(cat_1) = NRE$.

Proof. For the inclusion $NOSBP_m^{d_3}(cat_1) \supseteq NRE$ we will simulate with a SBP system with one region $\Pi = (O, C, \mu, w_1, R_1, E_1, i_0 = 1)$ a register machine $M = (n, \mathcal{P}, l_0, l_h)$. The system Π is defined as follows:

$$\begin{aligned} O &= \{a_i \mid 1 \leq i \leq n\} \cup \{l_1, l_2 \mid l_1 : (add(r), l_2) \in \mathcal{P}\} \\ &\quad \cup \{l_1, l_2, l_3, \overline{l_1}, \overline{l_2}, X, Y \mid l_1 : (sub(r), l_2, l_3)\}; \\ C &= \{c\}, \quad \mu = [\,]_1, \quad w_1 = l_0. \end{aligned}$$

The sets R_1 of rules and E_1 of regular expressions are constructed as follows:

- for any instruction $l_1 : (add(r), l_2)$ we add the rule $r_{(l_1,1)} : l_1 \to a_r l_2$ to R_1 and the regular expression $d^* r_{(l_1,1)} d^*$ to E_1;
- for any instruction $l_1 : (sub(r), l_2, l_3)$ we add the following rules to R_1 and the regular expressions to E_1

The rules added to R_1	The regular expressions added to E_1
$r_{(l_1,1)} : l_1 \to \overline{l_1} X$ $r_{(l_1,2)} : c a_r \to c$ $r_{(l_1,3)} : \overline{l_1} \to \overline{l_2}$	$d^* r_{(l_1,1)} r_{(l_1,2)} r_{(l_1,3)} d^*$
$r_{(l_1,4)} : X \to Y$ $r_{(l_1,5)} : Y \to \lambda$ $r_{(l_1,6)} : \overline{l_1} \to l_3$	$d^* r_{(l_1,4)} r_{(l_1,5)} r_{(l_1,6)} d^*$
$r_{(l_1,7)} : \overline{l_2} \to l_2$	$d^* r_{(l_1,7)} d^*$

- for the instruction $l_1 : halt$, the rule $r_{l_1} : l_1 \to \lambda$ is added to R_1 and the regular expression $d^* r_{l_1} d^*$ is added to E_1.

Similarly as in the proof of Theorem 2, we model the value stored in the register r of M as the multiplicity of the object a_r in a configuration of Π.

Since the scenarios are nondeterministically selected from the languages indicated by the regular expressions, and these scenarios may contain as a prefix a string of an arbitrary length and which is composed only by symbols d, then we don't know when the first rules of the scenarios will be executed.

Let Π be in a configuration $C_1 = a_1^{t_1} \dots a_r^{t_r} \dots a_n^{t_n} l_1 c$ and let us assume that in the configuration C_1 a rule from R_1 will be executed. In this configuration there might exist scenarios already in execution and/or scenarios that can be started. No matter which is the case, the single rule that can be applied in configuration C_1 is $l_1 \to \overline{l_1} X$ which belongs to a scenario s_1 from $L(d^* \ \ l_1 \to \overline{l_1} X \ \ ca_r \to c \ \ \overline{l_1} \to \overline{\overline{l_2}} \ \ d^*)$ (a scenario started in a previous configuration). This rule will be applied once and the resulting configuration will be $C_2 = a_1^{t_1} \dots a_r^{t_r} \dots a_n^{t_n} \overline{l_1} X c$. Next, we distinguish two cases:

- if $t_r > 0$ (that is, in C_2 there exists objects a_r) then the rule $ca_r \to c$ from scenario s will be executed. Moreover in this configuration will start new scenarios (apart from those already in execution). In particular, a scenario s_2 from $L(d^* \ \ X \to Y \ \ Y \to \lambda \ \ \overline{l_1} \to l_3 \ \ d^*)$ will be executed (which means that the rule $X \to Y$ will compete for objects, at a certain moment, in one subsequent configuration). However, there might be the case that a scenario of the same kind, started in a previous step, attempts to execute the rule $X \to Y$ in configuration C_2. Consequently we have two possible cases: in configuration C_2 will be only executed the rule $ca_r \to c$ (hence the next configuration will become $C_{(3,1)} = a_1^{t_1} \dots a_r^{t_r - 1} \dots a_n^{t_n} \overline{l_1} X c$) or the pair of rules $ca_r \to c$ and $X \to Y$ (hence the next configuration will become $C_{(3,2)} = a_1^{t_1} \dots a_r^{t_r - 1} \dots a_n^{t_n} \overline{l_1} Y c$). In the first case (i.e., in configuration $C_{(3,1)}$) we have again a branch in the computation: either will be executed the rule $\overline{l_1} \to \overline{l_2}$ (which means that the next configuration will be $C_{(3,1,1)} = a_1^{t_1} \dots a_r^{t_r - 1} \dots a_n^{t_n} \overline{l_2} X c$) or the pair of rules $\overline{l_1} \to \overline{l_2}$ and $X \to Y$ (which means that the next configuration will be $C_{(3,1,2)} = a_1^{t_1} \dots a_r^{t_r - 1} \dots a_n^{t_n} \overline{l_2} Y c$).
  It follows that for the configuration $C_{(3,1,1)}$ will be executed a scenario that, at a certain moment, will rewrite firstly the object X into Y (by an application of the rule $X \to Y$) and then will delete the object Y (by an application of the rule $Y \to \lambda$). In the same time, a scenario that applies the rule $\overline{\overline{l_2}} \to l_2$ will be executed and the configuration reached will be $a_1^{t_1} \dots a_r^{t_r - 1} \dots a_n^{t_n} l_2 c$ which corresponds to a correct simulation of the register machine subtraction instruction.
- if $t_r = 0$ then the rule $ca_r \to c$ from scenario s cannot be executed anymore and so, the execution of the scenario s will be interrupted (hence the rule $r_{(l_1,3)} : \overline{l_1} \to \overline{l_2}$ is not executed anymore in this simulation of the subtraction instruction). However the rule $r_{(l_1,4)} : X \to Y$ in a scenario from the set $L(d^* r_{(l_1,4)} r_{(l_1,5)} r_{(l_1,6)} d^*)$ will be executed at a certain moment. Next, the object Y will trigger the execution of the rule $r_{(l_1,5)} : Y \to \lambda$. Finally the rule $r_{(l_1,6)}$:

$\overline{l_1} \rightarrow l_3$ is applied and the resulting multiset will become $a_1^{t_1} \dots a_n^{t_n} l_3 c$ which again corresponds to a correct simulation of the register machine subtraction instruction.

It follows that Π correctly simulates the computation of M, and so, taking into account the Turing-Church thesis, $NOSBP_m^{d_3}(cat_1) = NRE$.

5 Conclusion

Metabolic pathways are usually composed of chains of enzymatically catalyzed chemical reactions. They are interconnected in a complex way in the framework of a metabolic network. Inspired by this biological phenomenon, we have defined and studied the scenario based P systems. In this computational model, regular expressions are used to express the causal dependence relations existing between various executions of the rules. In this way we intend to identify certain causalities in the chains of reactions connecting different parts of the metabolic network.

References

1. Agrigoroaiei, O., Ciobanu, G., Flattening the Transition P Systems with Dissolution, *Lecture Notes in Computer Science* vol.6501, pp.53–64, 2011.
2. Agrigoroaiei, O., Ciobanu, G., Quantitative Causality in Membrane Systems, *Lecture Notes in Computer Science* vol.7184, pp.62–72, 2012.
3. Busi, N., Causality in Membrane Systems, *Lecture Notes in Computer Science* vol.4860, pp.160–171, 2007.
4. Ciobanu, G., Lucanu, D., Events, Causality and Concurency in Membrane Systems, *Lecture Notes in Computer Science* vol.4860, pp.209–227, 2007.
5. Rozenberg, G., Salomaa, A. (Eds.): *Handbook of Formal Languages*, Springer Verlag, Berlin, 2004.
6. Minsky, M., *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, 1967.
7. Păun, G.: *Membrane Computing. An Introduction*, Springer, 2002.
8. Sburlan, D., P Systems with Chained Rules, *Lecture Notes in Computer Science* vol.7184, pp.359–370, 2012.

Universal P Systems: One Catalyst Can Be Sufficient

Rudolf Freund[1] and Gheorghe Păun[2]

[1] Technische Universität Wien, Institut für Computersprachen
Favoritenstr. 9, A-1040 Wien, Austria
rudi@emcc.at

[2] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania

and

Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, ghpaun@gmail.com

Summary. Whether P systems with only one catalyst can already be universal, is still an open problem. Here we establish universality (computational completeness) by using specific variants of additional control mechanisms. At each step using only multiset rules from one set of a finite number of sets of rules allows for obtaining computational completeness with one catalyst and only one membrane. If the targets are used for choosing the multiset of rules to be applied, for getting computational completeness with only one catalyst more than one membrane is needed. If the available sets of rules change periodically with time, computational completeness can be obtained with one catalyst in one membrane. Moreover, we also improve existing computational completeness results for P systems with mobile catalysts and for P systems with membrane creation.

1 Introduction

P systems with catalytic rules were already considered in the originating papers for membrane systems, see [9]. In [3] two catalysts were shown to be sufficient for getting universality/computational completeness (throughout this paper, with these notions we will indicate that all recursively enumerable sets of (vectors of) non-negative integers can be generated). Since then, it has become one of the most challenging open problems in the area of P systems, whether or not one catalyst might already be enough to obtain computational completeness.

Using additional control mechanisms as, for example, priorities or promoters/inhibitors, P systems with only one catalyst can be shown to be computationally completene, e.g., see Chapter 4 of [11]. On the other hand, additional features

for the catalyst may be taken into account; for example, we may use bi-stable catalysts (catalysts switching between two different states) or mobile catalysts (catalysts able to cross membranes). Moreover, additional membrane features may be used, for example, membrane creation or controlling the membrane permeability by means of the operations δ and τ.

P systems with membrane creation were introduced in [8], showing both their universality and efficiency (the Hamiltonian path problem is solved in linear time in a semi-uniform way; this result was improved in [4], where a polynomial solution to the Subset Sum problem in a uniform way is provided). For proving universality, in [8] (Theorem 2) P systems starting with one membrane, having four membranes at some time during the computation, using one catalyst, and also controlling the membrane permeability by means of the operations δ (deleting the surrounding membrane) and τ (increasing the thicknes of the surrounding membrane, i.e., making it impermeable for objects to pass through) are needed. However, as already shown in [10], P systems with one catalyst and using the operations δ and τ are universal, i.e., the membrane creation facility is not necessary for getting universality in this framework. Here we improve the result shown in [8] from two points of view: (i) the control of membrane permeability is not used, and (ii) the maximal number of membranes used during a computation is two.

P systems with mobile catalysts were introduced in [5], and their universality was proved with using three membranes and target indications of the forms *here*, *out*, and in_j. We here improve this result by replacing the target indications in_j with the weaker one *in*.

Recently, several variants of P systems using only one catalyst together with control mechanisms for choosing the rules applicable in a computation step have been considered: for example, in [6] the rules are labeled with elements from an alphabet H and in each step a maximal multiset of rules having the same label from H is applied. In this paper, we will give a short proof for the universality of these *P systems with label selection* with only one catalyst in a single membrane. As a specific variant, for each membrane we can choose the rules according to the target indications, and we will prove universality for these *P systems with target selection* with only one catalyst, but needing more than one membrane (such systems with only one membrane lead to the still open problem of catalytic P systems with one catalyst).

Regular control languages were considered already in [6] for the maximally parallel derivation mode, whereas in [1] universality was proved for the sequential mode: there even only non-cooperative rules were needed in one membrane for time-varying P systems to obtain universality (in time-varying systems, the set of available rules varies periodically with time, i.e., the regular control language is of the very specific form $W = (U_1 \dots U_p)^*$, allowing to apply rules from a set U_i in the computation step $pn + i$, $n \geq 0$; p is called the *period*), but a bounded number of steps without applying any rule had to be allowed. We here prove that *time-varying P systems* using the maximally parallel derivation mode in one membrane

with only one catalyst are computationally complete with a period of six and the usual halting when no rule can be applied.

2 Prerequisites

The set of integers is denoted by $\mathbb{Z}$, the set of non-negative integers by $\mathbb{N}$. An *alphabet* V is a finite non-empty set of abstract *symbols*. Given V, the free monoid generated by V under the operation of concatenation is denoted by V^*; the elements of V^* are called strings, and the *empty string* is denoted by λ; $V^* \setminus \{\lambda\}$ is denoted by V^+. Let $\{a_1, \cdots, a_n\}$ be an arbitrary alphabet; the number of occurrences of a symbol a_i in a string x is denoted by $|x|_{a_i}$; the *Parikh vector* associated with x with respect to $a_1, \cdots, a_n$ is $\left(|x|_{a_1}, \cdots, |x|_{a_n}\right)$. The *Parikh image* of a language L over $\{a_1, \cdots, a_n\}$ is the set of all Parikh vectors of strings in L, and we denote it by $Ps(L)$. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$; for families of languages of a one-letter alphabet, the corresponding sets of non-negative integers are denoted by NFL.

A (finite) multiset over the (finite) alphabet V, $V = \{a_1, \cdots, a_n\}$, is a mapping $f : V \longrightarrow \mathbb{N}$ and represented by $\langle f(a_1), a_1\rangle \cdots \langle f(a_n), a_n\rangle$ or by any string x the Parikh vector of which with respect to $a_1, \cdots, a_n$ is $(f(a_1), \cdots, f(a_n))$. In the following we will not distinguish between a vector $(m_1, \cdots, m_n)$, its representation by a multiset $\langle m_1, a_1\rangle \cdots \langle m_n, a_n\rangle$ or its representation by a string x having the Parikh vector $\left(|x|_{a_1}, \cdots, |x|_{a_n}\right) = (m_1, \cdots, m_n)$. Fixing the sequence of symbols $a_1, \cdots, a_n$ in the alphabet V in advance, the representation of the multiset $\langle m_1, a_1\rangle \cdots \langle m_n, a_n\rangle$ by the string $a_1^{m_1} \cdots a_n^{m_n}$ is unique. The set of all finite multisets over an alphabet V is denoted by V°.

The family of regular and recursively enumerable string languages is denoted by REG and RE, respectively. For more details of formal language theory the reader is referred to the monographs and handbooks in this area as [2] and [12].

A *register machine* is a tuple $M = (m, B, l_0, l_h, P)$, where m is the number of registers, P is the set of instructions bijectively labeled by elements of B, $l_0 \in B$ is the initial label, and $l_h \in B$ is the final label. The instructions of M can be of the following forms:

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$
  Increase the value of register j by one, and non-deterministically jump to instruction l_2 or l_3. This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$
  If the value of register j is zero then jump to instruction l_3, otherwise decrease the value of register j by one and jump to instruction l_2. The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$. Stop the execution of the register machine.

A *configuration* of a register machine is described by the contents of each register and by the value of the current label, which indicates the next instruction

to be executed. Computations start by executing the first instruction of P (labeled with l_0), and terminate with reaching the $HALT$-instruction.

Register machines provide a simple universal computational model [7]. In the generative case as we need it later, we start with empty registers, use the first two registers for the necessary computations and take as results the contents of the k registers 3 to $k+2$ in all possible halting computations; during a computation of M, only the registers 1 and 2 can be decremented. In the following, we shall call a specific model of P systems *computationally complete* or *universal* if and only if for any (generating) register machine M we can effectively construct an equivalent P system Π of that type simulating each step of M in a bounded number of steps and yielding the same results.

2.1 P Systems

The basic ingredients of a (cell-like) P system are the membrane structure, the objects placed in the membrane regions, and the evolution rules. The *membrane structure* is a hierarchical arrangement of membranes. Each membrane defines a *region/compartment*, the space between the membrane and the immediately inner membranes; the outermost membrane is called the *skin membrane*, the region outside is the *environment*, also indicated by (the label) 0. Each membrane can be labeled, and the label (from a set Lab) will identify both the membrane and its region. The membrane structure can be represented by a rooted tree (with the label of a membrane in each node and the skin in the root), but also by an expression of correctly nested labeled parentheses. The *objects* (multisets) are placed in the compartments of the membrane structure and usually represented by strings, with the multiplicity of a symbol corresponding to the number of occurrences of that symbol in the string. *The evolution rules* are multiset rewriting rules of the form $u \to v$, where u is a multiset of objects from a given set O and $v = (b_1, tar_1) \dots (b_k, tar_k)$ with $b_i \in O$ and $tar_i \in \{here, out, in\}$ or $tar_i \in \{here, out\} \cup \{in_j \mid j \in Lab\}$, $1 \leq i \leq k$. Using such a rule means "consuming" the objects of u and "producing" the objects $b_1, \dots, b_k$ of v; the *target indications* $here$, out, and in mean that an object with the target $here$ remains in the same region where the rule is applied, an object with the target out is sent out of the respective membrane (in this way, objects can also be sent to the environment, when the rule is applied in the skin region), while an object with the target in is sent to one of the immediately inner membranes, non-deterministically chosen, wheras with in_j this inner membrane can be specified directly. In general, we omit the target indication $here$.

Formally, a (cell-like) P system is a construct

$$\Pi = (O, \mu, w_1, \dots, w_m, R_1, \dots, R_m, f)$$

where O is the alphabet of objects, μ is the membrane structure (with m membranes), $w_1, \dots, w_m$ are multisets of objects present in the m regions of μ at the beginning of a computation , $R_1, \dots, R_m$ are finite sets of evolution rules, associated with the regions of μ, and f is the label of the membrane region from

which the outputs are taken ($f = 0$ indicates that the output is taken from the environment).

If a rule $u \to v$ has at least two objects in u, then it is called *cooperative*, otherwise it is called *non-cooperative*. In *catalytic P systems* we use non-cooperative as well as *catalytic rules* which are of the form $ca \to cv$, where c is a special object which never evolves and never passes through a membrane (both these restrictions can be relaxed), but it just assists object a to evolve to the multiset v. In a *purely catalytic P system* we only allow catalytic rules. In both catalytic and purely catalytic P systems, we replace O by O, C in order to specify those objects from O which are the catalysts in the set C.

The evolution rules are used in the *non-deterministic maximally parallel* way, i.e., in any computation step of Π we choose a multiset of rules from 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 membrane regions $1, \ldots, m$.

The membranes and the objects present in the compartments of a system at a given time form a *configuration*; starting from a given *initial configuration* and using the rules as explained above, we get *transitions* among configurations; a sequence of transitions forms a *computation*. A computation is *halting* if it reaches a configuration where no rule can be applied. With a halting computation we associate a *result*, in the form of the number of objects present in membrane f in the halting configuration. The set of vectors of non-negative integers and the set of (Parikh) vectors of non-negative integers obtained as results of halting computations in Π are denoted by $N(\Pi)$ and $Ps(\Pi)$, respectively.

The family of sets $Y(\Pi)$, $Y \in \{N, Ps\}$, computed by P systems with at most m membranes and cooperative rules and with non-cooperative rules is denoted by $YOP_m(coop)$ and $YOP_m(ncoo)$, respectively. It is well known that for any $m \geq 1$, $YREG = YOP_m(ncoo) \subset NOP_m(coop) = YRE$, see [9].

The family of sets $Y(\Pi)$, $Y \in \{N, Ps\}$, computed by (purely) catalytic P systems with at most m membranes and at most k catalysts is denoted by $YOP_m(cat_k)$ ($YOP_m(pcat_k)$); from [3] we know that, with the results being sent to the environment in order to avoid the discussion how to count the catalysts in the skin membrane, we have $YOP_1(cat_2) = YOP_1(pcat_3) = YRE$.

If we allow catalysts to move from one membrane region to another one, then we speak of *P systems with mobile catalysts*. The families of sets $N(\Pi)$ and $Ps(\Pi)$ computed by P systems with at most m membranes and k mobile catalysts is denoted by $NOP_m(mcat_k)$ and $PsOP_m(mcat_k)$, respectively.

For all the variants of P systems using rules of some type X as defined above, we may consider systems containing only rules of the form $u \to v$ where $u \in O$ and $v = (b_1, tar) \ldots (b_k, tar)$ with $b_i \in O$ and $tar \in \{here, out, in\}$ or $tar \in \{here, out\} \cup \{in_j \mid j \in H\}$, $1 \leq i \leq k$, i.e., in each rule there is only one target for all objects b_i; if *catalytic rules* are considered, then we request the rules to be of the form $ca \to c(b_1, tar) \ldots (b_k, tar)$. *P systems with target selection* contain only these forms of rules; moreover, in each computation step, for each membrane

region i we choose a maximal non-empty (if it exists) multiset of rules from R_i having the same target indication tar (for different membranes these targets may be different). The families of sets $N(\Pi)$ and $Ps(\Pi)$ computed by P systems with target selection with at most m membranes and rules of type X are denoted by $NOP_m(X, ts)$ and $PsOP_m(X, ts)$, respectively.

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. Then a *P system with label selection* is a construct

$$\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, H, W, f)$$

where $\Pi' = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, f)$ is a P system as defined above, H is a set of labels for the rules in the sets $R_1, \ldots, R_m$, and $W \subseteq 2^H$. In any transition step in 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 and the set R cannot be extended by any further rule with a label from U so that the obtained multiset of rules would still be applicable to the existing objects in the membrane regions $1, \ldots, m$. The family of sets $N(\Pi)$ and $Ps(\Pi)$ computed by P systems with label selection with at most m membranes and rules of type X is denoted by $NOP_m(X, ls)$ and $PsOP_m(X, ls)$, respectively.

Another method to control the application of the labeled rules is to use control languages (see [6] and [1]). A *controlled P system* is a construct

$$\Pi = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, H, W, f)$$

where $\Pi' = (O, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, f)$ is a P system as defined above, H is a set of labels for the rules in the sets $R_1, \ldots, R_m$, and W is a string language over 2^H from a family FL. Every successful computation in Π has to follow a control word $U_1 \ldots U_n \in W$: in transition step i, only rules with labels in U_i are allowed to be applied, and after the n-th transition, the computation halts; we may relax this end condition, and then we speak of *weakly controlled P systems*. If $W = (U_1 \ldots U_p)^*$, 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 $Y(\Pi)$, $Y \in \{N, Ps\}$, computed by (weakly) controlled P systems and (weakly) time-varying P systems with period p, with at most m membranes and rules of type X as well as control languages in FL is denoted by $YOP_m(X, C(FL))$ $(YOP_m(X, wC(FL)))$ and $YOP_m(X, TV_p)$ $(YOP_m(X, wTV_p))$, respectively.

In the *P systems with membrane creation* considered in this paper, besides the catalytic rules $ca \to c(u, tar)$ and the non-cooperative rules $a \to (u, tar)$ we also use catalytic membrane creation rules of the form $ca \to c[\ u\ ]_i$ (in the context of c, from the object a a new membrane with label i containing the multiset u is generated) and membrane dissolution rules $a \to u\delta$ (we assume that no objects can be sent into a membrane which is going to be dissolved; with dissolving the membrane i by applying δ, all objects contained inside this membrane are collected in the region surrounding the dissolved membrane); in all cases, c is a

catalyst, a is an object, u is a multiset, and tar is a target indication of the form $here$, out, and in_j. The family of sets $Y(\Pi)$, $Y \in \{N, Ps\}$, computed by such P systems with membrane creation and using at most k catalysts, with m initial membranes and having at most h membranes during its computations is denoted by $YP_{m,h}(cat_k, mcre)$.

3 Computational Completeness of P Systems with Label Selection

Theorem 1. $YOP_1(cat_1, ls) = YRE$, $Y \in \{N, Ps\}$.

Proof. We only prove the inclusion $PsRE \subseteq PsOP_1(cat_1, ls)$. Let us consider a register machine $M = (n+2, B, l_0, l_h, I)$ with only the first and the second register ever being decremented, and let $A = \{a_1, \ldots, a_{n+2}\}$ be the set of objects for representing the contents of the registers 1 to $n+2$ of M. We construct the following P system:

$$\begin{aligned}
&\Pi = (O, \{c\}, [\ ]_1, cdl_0, R_1, H, W, 0),\\
&O = A \cup B \cup \{c, d, \#\},\\
&H = \{l, l' \mid l \in B\} \cup \{l_x \mid x \in \{1, 2, d, \#\}\},
\end{aligned}$$

and the sets of labels in W and the rules for R_1 are defined as follows:

A. Let $l_i : (\texttt{ADD}(r), l_j, l_k)$ be an ADD instruction in I. If $r > 2$, then the (labeled) rules

$$l_i : l_i \to l_j(a_r, out), \quad l_i' : l_i \to l_k(a_r, out),$$

are introduced, and for $r \in \{1, 2\}$, we introduce the rules

$$l_i : l_i \to l_j a_r, \quad l_i' : l_i \to l_k a_r.$$

In both cases, we define $\{l_i, l_i'\}$ to be the corresponding set of labels in W. The contents of each register r, $r \in \{1, 2\}$, is represented by the number of objects a_r present in the skin membrane; any object a_r wit $r > 2$ is immediately sent out into the environment.

B. The simulation of a SUB instruction $l_i : (\texttt{SUB}(r), l_j, l_k)$, for $r \in \{1, 2\}$, is carried out by the following rules and the corresponding sets of labels in W: For the case that the register r, $r \in \{1, 2\}$, is empty we take the (labeled) rules

$$l_i : l_i \to l_k, \quad l_r : ca_r \to c, \quad l_d : cd \to c\#,$$

(if no symbol a_r is present, i.e., if the register r is empty, then the trap symbol $\#$ is introduced) and for the case that the register r is not empty, we introduce the rules

$$l_i' : l_i \to l_j, \quad l_r' : ca_r \to c\#$$

(if at least one symbol a_r is present, i.e., if the register r is not empty, then the trap symbol # is introduced); the corresponding sets of labels to be taken into W are $\{l_i, l_r, l_d\}$ and $\{l'_i, l'_r\}$, respectively. In both cases, the simulation of the SUB instruction works correctly, if we have made the right choice.

C. We also add the labeled rule $l_\# : \# \to \#$ to R_1 and $\{\#\}$ to W, hence, the computation cannot halt once the trap symbol # has been generated.

In sum, we have the equality $Ps\,(M) = Ps\,(\Pi)$, which completes the proof. □

4 Computational Completeness of P Systems with Target Selection

Theorem 2. *$YOP_7\,(cat_1, ts) = YRE$, $Y \in \{N, Ps\}$.*

Proof. We only prove the inclusion $PsRE \subseteq PsOP_7\,(cat_1, ts)$. Let us consider a register machine $M = (n+2, B, l_0, l_h, I)$ with only the first and the second register ever being decremented, and let $A = \{a_1, \ldots, a_{n+2}\}$ be the set of objects for representing the contents of the registers 1 to $n+2$ of M. The set of labels $B \setminus \{l_h\}$ is divided into three disjoint subsets:

$$B_+ = \{l \mid l_i : (\texttt{ADD}\,(r), l_j, l_k) \in I\},$$
$$B_{-r} = \{l \mid l_i : (\texttt{SUB}\,(r), l_j, l_k) \in I\}, r \in \{1, 2\};$$

moreover, we define $B_- = B_{-1} \cup B_{-2}$, $B'_- = \{l' \mid l \in B_-\}$, $B''_- = \{l'' \mid l \in B_-\}$, and $B' = B_+ \cup B_- \cup B'_- \cup B''_-$ as well as $A = \{a_1, \ldots, a_{n+2}\}$. We construct the following P system:

$$\Pi = (O, \{c\}, [\,[\quad]_2 \ldots [\quad]_7\,]_1, w_1, \ldots, w_7, R_1, \ldots, R_7, 0),$$
$$O = A \cup \{a'_1, a'_2\} \cup B' \cup \{c, d, \#\},$$

with $w_1 = l_0$, $w_2 = c$, and $w_i = \lambda$ for $3 \leq i \leq 7$. In order to make argumentation easier, in the following we refer to the membrane labels 1 to 7 according to the following table:

1	2	3	4	5	6	7
skin	$-$	0_1	0_2	$-_1$	$-_2$	$+$

The sets of rules now are constructed as follows:

A. The simulation of any instruction from I starts in the skin membrane with moving all objects except the output symbols a_r for $r > 2$ into an inner membrane; according to the definition, taking the target *in* means non-deterministically choosing one of the inner membranes, but the same membrane for all objects to be moved in. The output symbols a_r for $r > 2$ are sent out into the environment by $a_r \to (a_r, out)$, thus yielding the result of a halting computation as the number of symbols a_r sent out into the environment during this computation. Hence, in sum we get

$$R_1 = \{x \to (x, in) \mid x \in B_+ \cup B_- \cup \{a_1, a_2, a_1', a_2', \#\}\} \cup \{x \to (xd, in) \mid x \in B_-'\} \cup \{a_r \to (a_r, out) \mid 3 \le r \le n+2\}.$$

B. For the simulation of an ADD instruction $l_i : (\mathtt{ADD}(r), l_j, l_k) \in I$ all non-terminal symbols (all symbols except a_r for $r > 2$) are expected to have been sent to membrane $+$:

$$\begin{aligned} R_+ = {} & \{l_i \to (l_j a_r, out), l_i \to (l_k a_r, out) \mid l_i : (\mathtt{ADD}(r), l_j, l_k) \in I\} \\ & \cup \{l \to (\#, out) \mid l \in B' \setminus B_+\} \\ & \cup \{x \to (x, out) \mid x \in \{a_1, a_2, \#\}\}. \end{aligned}$$

If the symbols arrive in membrane $+$ with a label $l \in B' \setminus B_+$, then the trap symbol $\#$ is generated and the computation will never halt. Sending out all terminal symbols a_r for $r > 2$ from the skin membrane can be done as a last step of a successful computation, but we may also choose to send out all those present there at a specific moment instead of immediately continuing the simulation of an instruction of the register machine. Hence, the simulation of an ADD instruction by Π takes at most three steps.

C. The simulation of a SUB instruction $l_i : (\mathtt{SUB}(r), l_j, l_k)$ is carried out in two steps for the zero test, i.e., when the register r is empty, using (the rules in) membrane 0_r, and in five steps for decrementing the number of symbols a_r, first using membrane $-_r$ to mark the corresponding symbols a_r into a_r' and then using the catalyst c in membrane $-$ to erase one of these primed objects; the marking procedure is necessary to guarantee that the catalyst erases the correct object. For $r \in \{1, 2\}$, we define the following sets of rules:

$$\begin{aligned} R_{0_r} = {} & \{l_i \to (l_k, out), a_r \to (\#, out) \mid l_i : (\mathtt{SUB}(r), l_j, l_k) \in I\} \\ & \cup \{l \to (\#, out) \mid l \in B' \setminus B_{-r}\} \\ & \cup \{x \to (x, out) \mid x \in \{a_{3-r}, \#\}\}. \end{aligned}$$

If the number of objects a_r is not zero, i.e., if the register r is not empty, the introduction of the trap symbol $\#$ causes the computation to never halt. On the other hand, if we want to decrement the register, we have to guarantee that exactly one symbol a_r is erased:

$$\begin{aligned} R_{-_r} = {} & \{l_i \to (l_i', out) \mid l_i \in B_{-r}\} \cup \{a_r \to (a_r', out)\} \\ & \cup \{l \to (\#, out) \mid l \in B' \setminus B_{-r}\} \\ & \cup \{x \to (x, out) \mid x \in \{a_{3-r}, \#\}\}. \end{aligned}$$

The whole multiset of objects, with the primed versions of l_i and the a_r, via the skin membrane now has to enter membrane $-$; here the dummy symbol d guarantees that the catalyst cannot do nothing if no primed symbol a_r' has arrived; again the generation of $\#$ causes the computation to not halt anymore:

$$\begin{aligned} R_- = {} & \{l_i' \to l_j'', ca_r' \to c, l_i'' \to \#, l_i'' \to (l_j, out) \mid l_i : (\mathtt{SUB}(r), l_j, l_k) \in I\} \\ & \cup \{cd \to c\#, d \to (\lambda, out)\} \cup \{a_r' \to (a_r, out) \mid r \in \{1, 2\}\}, \\ & \cup \{l \to (\#, out) \mid l \in B' \setminus B_-''\} \\ & \cup \{x \to (x, out) \mid x \in \{a_{3-r}, \#\}\}. \end{aligned}$$

In R_-, for correctly continuing the simulation of a SUB instruction, exactly two steps have to be carried out:

In the first step, the target indication *here* has to be used with applying the two rules $l_i' \to l_j''$ and $ca_r' \to c$ (eliminating exactly one copy of a_r', i.e., decrementing register r) and leaving all other objects unchanged; if instead the target indication *out* were chosen, the forced application of the rule $l_i' \to (\#, out)$ would yield the trap symbol #. In the second step, the target indication *out* has to be chosen and the rules $l_i'' \to (l_j, out)$, $d \to (\lambda, out)$, and $a_r' \to (a_r, out)$ are to be applied; if instead the target indication *here* were chosen again, the forced application of the rule $l_i'' \to \#$ would yield the trap symbol #.

Whenever a trap symbol is generated in one of the inner membranes, we get an infinite computation, as in R_1 we have the rule $\# \to (\#, in)$ and in every inner membrane we have the rule $\# \to (\#, out)$.

We finally observe that a computation in Π halts if and only if the final label l_h appears (and then stays in the skin membrane) and no trap symbol # is present, hence, we conclude $Ps\,(M) = Ps\,(\Pi)$. □

To eventually reduce the number of inner membranes remains as a challenging task for future research.

5 Computational Completeness of Time-Varying P Systems

Theorem 3. *$NOP_1\,(cat_1, \alpha TV_6) = NRE$, $Y \in \{N, Ps\}$, $\alpha \in \{\lambda, w\}$.*

Proof. We only prove the inclusion $PsRE \subseteq PsOP_1\,(cat_1, TV_6)$. Let us consider a register machine $M = (n+2, B, l_0, l_h, I)$ with only the first and the second register ever being decremented. Again, we define $A = \{a_1, \ldots, a_{n+2}\}$ and divide the set of labels $B \setminus \{l_h\}$ into three disjoint subsets:

$$B_+ = \{l \mid l_i : (\texttt{ADD}\,(r), l_j, l_k) \in I\},$$
$$B_{-r} = \{l \mid l_i : (\texttt{SUB}\,(r), l_j, l_k) \in I\},\ r \in \{1, 2\};$$

moreover, we define $B_- = B_{-1} \cup B_{-2}$ as well as

$$B' = \left\{l, \tilde{l}, \hat{l} \mid l \in B \setminus \{l_h\}\right\} \cup \left\{l^-, l^0, \bar{l}^-, \bar{l}^0, \mid l \in B_-\right\}.$$

The main challenge in the construction for the time-varying P system Π is that the catalyst has to fulfill its task to erase an object a_r, $r \in \{1, 2\}$, for both objects in the same membrane where all other computations are carried out, too; hence, at a specific moment in the cycle of period six, parts of simulations of different instructions have to be coordinated in parallel. The basic components of the time-varying P system Π are defined as follows (we here do not distinguish between a rule and its label):

$$\Pi = (O, \{c\}, [\ ]_1, l_0, R_1 \cup \cdots \cup R_6, R_1 \cup \cdots \cup R_6, (R_1 \ldots R_6)^*, 0),$$
$$O = A \cup \{a'_1, a'_2\} \cup B' \cup \{c, h, \#\}.$$

We now list the rules in the sets of rules R_i to be applied in computation steps $6n + i$, $n \geq 0$, $1 \leq i \leq 6$:

$\mathbf{R}_1$: in this step, the ADD instructions are simulated, i.e., for each $l_i : (\texttt{ADD}(r), l_j, l_k) \in I$ we take

$cl_i \to ca_r\tilde{l}_j$, $cl_i \to ca_r\tilde{l}_k$ (only in the sixth step of the cycle, from $\tilde{l}_j$ and $\tilde{l}_k$ the corresponding unmarked labels l_j and l_k will be generated); in order to obtain the output in the environment, for $r \geq 3$, a_r has to be replaced by (a_r, out);

$cl \to cl^-$, $cl \to cl^0$ initiate the simulation of a SUB instruction for register 1 labeled by $l \in B_{-1}$;

$cl \to c\hat{l}$ marks a label $l \in B_{-2}$ (the simulation of such a SUB instruction for register 2 will start in step 4 of the cycle);

$\# \to \#$ keeps the trap symbol $\#$ alive guaranteeing an infinite loop once $\#$ has been generated;

$h \to \lambda$ eliminates the auxiliary object h needed for simulating SUB instructions and eventually generated two steps before.

$\mathbf{R}_2$: in the second and the third step, the SUB instructions on register 1 are simulated, i.e., for all $l \in B_{-1}$ we start with

$ca_1 \to ca'_1$ (if present, exactly one copy of a_1 can be primed) and

$l^- \to \bar{l}^- h$, $l^- \to \bar{l}^0 h$ for all $l \in B_{-1}$;

$\# \to \#$;

$c\tilde{l} \to c\tilde{l}$, $\tilde{l} \to \#$ for all $l \in B_+$,

$c\hat{l} \to c\hat{l}$, $\hat{l} \to \#$ for all $l \in B_{-2}$.

$\mathbf{R}_3$: for all $l_i : (\texttt{SUB}(1), l_j, l_k) \in I$ we take

$c\bar{l}_i^0 \to c\tilde{l}_k$, $a'_1 \to \#$, $\bar{l}_i^0 \to \#$ (zero test; if a primed copy of a_1 is present, then the trap symbol $\#$ is generated);

$\bar{l}_i^- \to \tilde{l}_j$, $ca'_1 \to c$, $ch \to c\#$ (decrement; the auxiliary symbol h is needed to keep the catalyst c busy with generating the trap symbol $\#$ if we have taken the wrong guess when assuming the register 1 to be non-empty);

$\# \to \#$;

$c\tilde{l} \to c\tilde{l}$, $\tilde{l} \to \#$ for all $l \in B_+$;

$c\hat{l} \to c\hat{l}$, $\hat{l} \to \#$ for all $l \in B_{-2}$.

$\mathbf{R}_4$: in the fourth step, the simulation of SUB instructions on register 2 is initiated, i.e., we take

$c\hat{l} \to cl^-$, $c\hat{l} \to cl^0$ for all $l \in B_{-2}$;

$c\tilde{l} \to c\tilde{l}$, $\tilde{l} \to \#$ for all $l \in B_+ \cup B_{-1}$;

$\# \to \#$,

$h \to \lambda$.

$\mathbf{R}_5$: in the fifth and the sixth step, the SUB instructions on register 2 are simulated, i.e., for all $l \in B_{-2}$ we start with

$ca_2 \to ca_2'$ (if present, exactly one copy of a_2 can be primed) and
$l^- \to \bar{l}^- h$, $l^- \to \bar{l}^0 h$ for all $l \in B_{-2}$;
$c\tilde{l} \to c\bar{l}$, $\tilde{l} \to \#$ for all $l \in B_+ \cup B_{-1}$;
$\# \to \#$.

$\mathbf{R}_6$: the simulation of SUB instructions $l_i : (\mathtt{SUB}\,(2), l_j, l_k) \in I$ on register 2 is finished by

$c\bar{l}_i^0 \to cl_k$, $a_2' \to \#$, $\bar{l}_i^0 \to \#$ (zero test; if a primed copy of a_2 is present, then the trap symbol # is generated);

$\bar{l}_i^- \to l_j$, $ca_2' \to c$, $ch \to c\#$ (decrement; the auxiliary symbol h is needed to keep the catalyst c busy with generating the trap symbol # if we have taken the wrong guess when assuming the register 2 to be non-empty);

$c\tilde{l} \to cl$, $\tilde{l} \to \#$ for all $l \in B_+ \cup B_{-1}$.

Without loss of generality, we may assume that the final label l_h in M is only reached by using a zero test on register 2; then, at the beginning of a new cycle, after a correct simulation of a computation from M in the time-varying P system Π no rule will be applicable in R_1 (another possibilty would be to take $c\bar{l}_i^0 \to c$ instead of $c\bar{l}_i^0 \to cl_h$ in R_6).

At the end of the cycle, in case all guesses have been correct, the requested instruction of M has been simulated and the label of the next instruction to be simulated is present in the skin membrane. Only in the case that M has reached the final label l_h, the computation in Π halts, too, but only if during the simulation of the computation of M in Π no trap symbol # has been generated; hence, we conclude $Ps\,(M) = Ps\,(\Pi)$. □

6 Computational Completeness of P Systems with Membrane Creation

Theorem 4. $YOP_{1,2}\,(cat_1, mcre) = YRE$, $Y \in \{N, Ps\}$.

Proof. We only prove the inclusion $PsRE \subseteq PsOP_{1,2}\,(cat_1, mcre)$. Let us consider a register machine $M = (n+2, B, l_0, l_h, I)$ with only the first and the second register ever being decremented, and let $A = \{a_1, \ldots, a_{n+2}\}$. We construct the following P system:

$$\Pi = \left(O, \{c\}, [\quad]_1, cdl_0, R_1, R_2, R_3, 0\right),$$
$$O = A \cup \{l, l', l'' \mid l \in B\} \cup \{c, d, d', d'', \#\},$$

and the sets of rules are constructed as follows.

A. For each ADD instruction $l_i : (\mathtt{ADD}\,(r), l_j, l_k)$ in I, the rules

$$\text{step 1: } l_i \to l_i',\ d \to d',$$
$$\text{step 2: } l_i' \to a_r l_j,\ l_i' \to a_r l_k,\ d' \to d.$$

are introduced in R_1 and obviously simulate an ADD instruction in two steps.

B. For each SUB instruction $l_i : (\texttt{SUB}\,(r), l_j, l_k)$ in I, the following rules are introduced in R_1 and R_{r+1}, $r \in \{1, 2\}$:

Step	R_1	R_{r+1}
1	$cl_i \to c[\ l_i\ ]_{r+1},\ d \to d'$	–
2	$ca_r \to c\,(a_r, in_{r+1}),\ d' \to (d', in_{r+1})$	$l_i \to l_i'$
3	–	$a_r \to \lambda\delta,\ l_i' \to l_i'',\ d' \to d''$
4	$cl_i'' \to cl_j,\ d'' \to d$	$l_i'' \to l_k,\ d'' \to d\delta$

A SUB instruction $l_i : (\texttt{SUB}\,(r), l_j, l_k)$ (with $r \in \{1, 2\}$) is simulated according to the four steps suggested in the table given above:

In the first step, we create a membrane with the label $r+1$, where l_i is sent to, and simultaneously d becomes d'. In the next step, if any a_r exists, i.e., if register r is not empty, then one copy of a_r should enter the membrane $r+1$ just having been created in the preceding step. Note that the selection of the membrane (the use of in_{r+1} instead of in) is important: a_r has to go to the membrane created in the previous step, when $r+1$ has been specified by the label l_i. At the same time, d' enters the membrane $r+1$, and l_i becomes l_i' in this membrane. If the register r is empty, then the catalyst is doing nothing in this second step.

In the third step, in membrane $r+1$, l_i' becomes l_i'' and d' becomes d''. If a_r is not present in membrane $r+1$, nothing else happens there in this step; if a_r is present, it dissolves the membrane and disappears. Observe that in both cases $ca_r \to c\,(a_r, in_{r+1})$ will not be applicable (anymore) in R_1. Thus, we either have $cl_i''d''$ in the skin membrane (when the register has been non-empty), or we have only c in the skin membrane and $l_i''d''$ in the inner membrane $r+1$. In the first case, in the fourth step we use the rules $cl_i'' \to cl_j$ and $d'' \to d$ from R_1, which is the correct continuation of the simulation of the SUB instruction; in the latter case, we use $l_i'' \to l_k$ and $d'' \to d\delta$ in R_{r+1}. The inner membrane is dissolved, and in the skin membrane we get the objects cl_kd. In both cases, the simulation of the SUB instruction is correct and we return to a configuration as that we started with, hence the simulation of another instruction can start.

C. We also add the rules $a_r \to (a_r, out)$ for $3 \le r \le n+2$ and $\# \to \#$ to R_1.

In any moment, any copy of a terminal symbol a_r in the skin membrane is sent out to the environment. Once the trap symbol $\#$ has been introduced, the computation continues forever.

There is one interference between the rules of Π simulating the ADD and the SUB instructions of M. If in the second step of simulating a SUB instruction, instead of $d' \to (d', in_{r+1})$ we use $d' \to d$, then the case when register r is non-empty continues correctly, as the simulation lasts four steps, and in the end d is present in the skin membrane (the dissolution of membrane r is done by a_r). If the register r has been empty, l_i'' will become l_k in membrane $r+1$ and it will remain there until d' enters the membrane, changes to d'', and then dissolves it (as long as d, d' switch to each other in the skin membrane, the computation cannot

halt). Thus, also in this case we return to the correct submultiset $cl_k d$ in the skin membrane.

Consequently, exactly the halting computations of M are simulated by the halting computations in Π; hence, $Ps(M) = Ps(\Pi)$. The observation that the maximal number of membranes in any computation of Π is two completes the proof. □

It remains as an open problem whether it is possible to use the target indication in only instead of the in_j.

7 Computational Completeness of P Systems with Mobile Catalysts

If the membrane creation rules are of the form $ca \to [\ cb\ ]_i$, then this implicitly means that the catalyst is moving from one region to another one. However, for mobile catalysts, the universality of such systems with only one catalyst has already been proved in [5], using three membranes and target indications of the forms $here$, out, and in_j. In this paper, we improve this result from the last point of view, making only use of the target indications $here$, out, and in. In fact, if in the proof of Theorem 2 we let the catalyst c move with all the other objects, then we immediately obtain a proof for $NOP_7(mcat_1) = NRE$ where even only the target indications out and in are used (but instead of three we need seven membranes).

Theorem 5. $YOP_3(mcat_1) = YRE$, $Y \in \{N, Ps\}$.

Proof. We only prove the inclusion $PsRE \subseteq PsOP_1(cat_1, ls)$. Let us consider a register machine $M = (n+2, B, l_0, l_h, I)$ with only the first and the second register ever being decremented, and let $A = \{a_1, \ldots, a_{n+2}\}$ be the set of objects for representing the contents of the registers 1 to $n+2$ of M. We construct the following P system:

$$\Pi = (O, \{c\}, [\ [\ \ ]_2 [\ \ ]_3\ ]_1, cl_0, R_1, R_2, R_3, 0),$$
$$O = A \cup \{l, l', l'', l''' \mid l \in B\} \cup \{c, \#\},$$

and the sets of rules are constructed as follows:

A. Let $l_i : (\texttt{ADD}(r), l_j, l_k)$ be an ADD instruction in I. If $r = 3$, then the rules $l_i \to l_j(a_3, out)$, $l_i \to l_k(a_3, out)$ are introduced in R_1; if $r \in \{1, 2\}$, in R_1 we introduce the rules $l_i \to l_j(a_r, in)$, $l_i \to l_k(a_r, in)$, as well as the rules $a_{4-j} \to \#$ and $\# \to \#$ in R_{j+1}, $j \in \{1, 2\}$. The contents of each register r, $r \in \{1, 2\}$, is represented by the number of objects a_r present in membrane $r+1$; any object a_r, $3 \le r \le n+2$, is immediately sent out into the environment. If a_{4-j} is introduced in membrane j, $j \in \{1, 2\}$, then the trap object $\#$ is produced and the computation never halts.

B. The simulation of a SUB instruction $l_i : (\texttt{SUB}\,(r)\,, l_j, l_k)$ is carried out by the following rules (the simulation again has four steps, as in the proof of Theorem 4):

For the first step, we introduce the rule $cl_i \to (c, in)\,(l_i, in)$ in R_1 and the rule $l_i \to \#$ in both R_2 and R_3 (if c and l_i are not moved together into an inner membrane, then the trap object # is produced and the computation never halts).

In the second step, R_{r+1} has to use the rule $cl_i \to cl_i'$. This checks whether c and l_i have been moved together into the right membrane $r+1$; if this is not the case, then the rule $cl_i \to cl_i'$ is not available and the rule $l_i \to \#$ must be used, which causes the computation to never halt.

Thus, after the second step, we know whether both c and l_i (l_i') are in the correct membrane $r+1$. The rules $ca_r \to (c, out)$ and $l_i' \to l_i''$are introduced in R_{r+1} in order to perform the third step of the simulation. If there is any copy of a_r in membrane $r+1$ (i.e., if register r is not empty), then the catalyst exits, while also removing a copy of a_r. Simultaneously, l_i' becomes l_i''. Hence, if the register r has been non-empty, we now have c in the skin membrane and l_i'' in membrane $r+1$; if register r has been empty, we have both c and l_i'' in membrane $r+1$. We introduce the rules $cl_i'' \to (c, out)\,(l_k, out)\,,\quad l_i'' \to (l_i''', out)\,,$ in R_{r+1} and the rules $cl_i''' \to cl_j,\ \ l_i''' \to \#,\ \ \# \to \#$ in R_1. If c is inside membrane $r+1$, we get cl_k in the skin membrane, which is the correct continuation for the case when the register is empty. If c is not in membrane $r+1$, then l_i'' exits alone thereby becoming l_i''', and, together with c, which waits in the skin membrane, introduces l_j, which is a correct continuation, too. If the rule $l_i'' \to (l_i''', out)$ is used although c is inside membrane $r+1$, then in the skin membrane we have to use the rule $l_i''' \to \#$ and the computation never halts.

In all cases, the simulation of the SUB instruction works correctly, and we return to a configuration with the catalyst and a label from H in the skin region.

In sum, we have the equality $Ps\,(M) = Ps\,(\Pi)$, which completes the proof. □

8 Final Remarks

Although we have exhibited several new universality results for P systems using only one catalyst together with some additional control mechanism, the orignal problem of characterizing the sets of non-negative integers generated by P systems with only one catalyst still remains open. A similar challenging problem is to consider *purely catalytic* P systems with only two catalysts: with only one catalyst, we obtain the regular sets; as shown in [3], three catalysts are enough to obtain universality. With two catalysts and some additional control mechanism, universality can be obtained, too; for example, the proof of Theorem 1 for P systems with label selection for the rules can easily be adapted for purely catalytic P systems, i.e., $NOP_1\,(pcat_2, ls) = NRE$. For the other variants of additional control mechanisms, the case of purely catalytic P systems with two catalysts remains for future research.

Acknowledgements. The work of Gheorghe Păun has been supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200, co-financed by FEDER funds.

References

1. A. Alhazov, R. Freund, H. Heikenwälder, M. Oswald, Yu. Rogozhin, S. Verlan, Sequential P systems with regular control. In: E. Csuhaj-Varjú, M. Gheorghe, G. Rozenberg, A. Salomaa, G. Vaszil (Eds.): *Membrane Computing - 13th International Conference, CMC 2012*, Budapest, Hungary, August 28-31, 2012, Revised Selected Papers, LNCS 7762, Springer, 2013, 112–127.
2. Jürgen Dassow, Gheorghe Păun: *Regulated Rewriting in Formal Language Theory.* Springer, 1989.
3. Rudolf Freund, Lila Kari, Marion Oswald, Petr Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Science* **330**, 2005, 251–266.
4. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez: P systems with membrane creation and rule input, to appear.
5. S.N. Krishna, A. Păun: Results on catalytic and ecolution-communication P systems. *New Generation Computing*, 22 (2004), 377–394.
6. K. Krithivasan, Gh. Păun, A. Ramanujan: On controlled P systems. *Fundamenta Informaticae*, to appear.
7. Marvin L. Minsky: *Computation: Finite and Infinite Machines.* Prentice Hall, Englewood Cliffs, New Jersey, USA, 1967.
8. M. Mutyam, K. Krithivasan: P systems with membrane creation: universality and efficiency. In: M. Margenstern, Y. Rogozhin (Eds.): *Proc. MCU 2001*, LNCS 2055, Springer, 2001, 276–287.
9. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.*, 61 (2000), 108–143 (see also TUCS Report 208, November 1998, `www.tucs.fi`).
10. Gh. Păun: Computing with membranes - a variant. *Intern. J. Found. Computer Sci.*, 11, 1 (2000), 167–182.
11. Gh. Păun, G. Rozenberg, A. Salomaa (Eds.): *The Oxford Handbook of Membrane Computing.* Oxford University Press, 2010.
12. Grzegorz Rozenberg, Arto Salomaa (Eds.): *Handbook of Formal Languages,* 3 volumes. Springer, 1997.
13. The P Systems Website: `http://ppage.psystems.eu`.

Kernel P Systems - Version 1

Marian Gheorghe[1,2], Florentin Ipate[2], Ciprian Dragomir[1], Laurenţiu Mierlă[2], Luis Valencia-Cabrera[3], Manuel García-Quismondo[3], and Mario J. Pérez-Jiménez[3]

[1] Department of Computer Science
University of Sheffield
Portobello Street, Regent Court, Sheffield, S1 4DP, UK
{m.gheorghe, c.dragomir}@sheffield.ac.uk
[2] Department of Computer Science
University of Bucarest
Str Academiei, 14, Bucarest, Romania
florentin.ipate@ifsoft.ro, laurentiu.mierla@gmail.com
[3] Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Seville, Spain
{lvalencia, mgarciaquismondo, marper}@us.es

Summary. A basic P system, called kernel P system[4] (kP system for short), combining features of different P systems introduced and studied so far is defined and discussed. The structure of such systems is defined as a dynamic graph, similar to tissue-like P systems, the objects are organised as multisets, and the rules in each compartment, rewriting and communication together with system structure changing rules, are applied in accordance with a specific execution strategy. The definition of kP systems is introduced and some examples illustrate this concept. Two classes of P systems, namely neural-like and generalised communicating P systems are simulated by kP systems. Some case studies prove the expressive power of these systems.

1 Introduction

Different classes of P systems have been introduced and studied for their computational power or for specifying or modelling various problems, like solving simple algorithms [4, 2], NP-complete problems [8] and other applications [5]. More recently various distributed algorithms and problems [13] have been studied with a new variant of P systems. In many cases the specification of the system investigated requires features, constraints or types of behaviour which are not always provided

[4] This concept was introduced initially in [10]; in this paper it is presented a revised version of it.

by the model in its initial definition. It helps in many cases to have some flexibility with modelling approaches, especially in the early stages of modelling, as it might simplify the model, shorten associated processes and clarify more complex or unknown aspects of the system. The downside of this is the lack of a coherent and well-defined framework that allows us to analyse, verify and test this behaviour and simulate the system. In this respect in [10] the concept of *kernel P system* (*kP system*) has ben introduced in order to include the most used concepts from P systems. It is intended to formally define these systems in an operational style and finally implement it within a model checker (SPIN [3], Maude [6]) and integrate it into the P–Lingua platform.

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

The paper consists of five chapters. Chapter 2 introduces basic definitions, Chapter 3 compares the newly introduce kP systems with some other classes of P systems, Chapter 4 presents a case study based on a static sorting, studying how this problem is solved with various variants of P systems. Finally Chapter 5 briefly describes two specification languages for kP systems with their implementations and some examples.

2 kP Systems

A kP system is a formal model that uses some well-known features of existing P systems and includes some new elements and, more importantly, it offers a coherent view on integrating them into the same formalism. The key elements of a kP system will be formally defined in this section, namely objects, types of rules, internal structure of the system and strategies for running such systems. Some preliminary formal concepts describing the syntax of kP systems and an informal description of the way these systems are executed will be introduced.

We consider that standard concepts like strings, multisets, rewriting rules, and computation are well-known concepts in P systems and indicate [15] as a comprehensive source of information in this respect. First we introduce the key concept of a compartment.

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 used by the definition of the kP systems will be instantiated from the compartment types defined above. The types of rules and the execution strategy 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* membrane structure*, which is a graph, (V, E), where V are vertices indicating components, and E edges; $C_i = (t_i, w_i)$, $1 \leq i \leq n$, is a* compartment *of the system consisting of a compartment type from T and an* initial multiset*, w_i over A; i_o is the* output compartment *where the result is obtained.*

Remark 2. The inner part of each compartment is called *region*, which is delimited by a *membrane*.

2.1 kP System Rules

The discussion below assumes that the rules introduced belong to the same compartment, C_i.

Each rule r may have a **guard** g, its generic form is $r\ \{g\}$. The rule r is applicable to a multiset w when its left hand side is contained into w and g is true for w. In the sequel we will analyse how the guards are specified and evaluated.

The guards are constructed using multisets over A and relational and Boolean operators – like Boolean expressions. Before presenting the definition we introduce some notations.

For a multiset w over A and an element $a \in A$, we denote by $\#_a(w)$ the number of a's occurring in w. Let $Rel = \{<, \leq, =, \neq, \geq, >\}$ be the set of relational operators, $\gamma \in Rel$, a relational operator, a^n a multiset and $r\ \{g\}$ a rule with guard g.

Definition 3. *If g is the* abstract relational expression γa^n *and the current multiset is w, then the guard denotes the* relational expression $\#_a(w)\gamma n$*. The guard g is true for the multiset w if $\#_a(w)\gamma n$ is true.*

Let us consider the Boolean operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction), listed wrt decreasing precedence order. Abstract relational expressions can be connected by Boolean operators generating *abstract Boolean expressions.*

Definition 4. *If g is the* abstract Boolean expression *and the current multiset is w, then the guard denotes the* Boolean expression *for w, obtained by replacing abstract relational expressions with relational expressions for w. The guard g is true for the multiset w when the Boolean expression for w is true.*

Definition 5. *A guard is: (i) one of the Boolean constants $true$ or $false$; (ii) an abstract relational expression; or (iii) an abstract Boolean expression.*

Example 1. If the rule is $r : ab \to c\ \{\geq a^5 \wedge \geq b^5 \vee \neg > c\}$, then this can be applied iff the current multiset, w, includes the left hand side of r, i.e., ab and the guard is true for w - it has at least 5 a's and 5 b's or no more than a c.

Definition 6. *A rule from a compartment $C_{l_i} = (t_{l_i}, w_{l_i})$ can 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 indicate the type of the current compartment, i.e., t_{l_i} – in this case it is ignored; if a link does not exist (the two compartments are not in E) then the rule is not applied; if a target, t_j, refers to a compartment type that has more than one instance connected to l_i, then one of them will be non-deterministically chosen;
- *(b)* **structure changing rules***; the following types are considered:*
  - *(b1)* **membrane division** *rule: $[x]_{t_{l_i}} \to [y_1]_{t_{i_1}} \ldots [y_p]_{t_{i_p}}\ \{g\}$,*
    where $x \in A^+$ and y_j has the form $y_j = (a_{j,1}, t_{j,1}) \ldots (a_{j,h_j}, t_{j,h_j})$ like in rewriting and communication rules; the compartment l_i will be replaced by p compartments; the j-th compartment, instantiated from the compartment type t_{i_j} contains the same objects as l_i, but x, which will be replaced by y_j; all the links of l_i are inherited by each of the newly created compartments;
  - *(b2)* **membrane dissolution** *rule: $[]_{t_{l_i}} \to \lambda\ \{g\}$;*
    the compartment 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} exists then one of them will be non-deterministically picked up; g is a guard that refers to the compartment instantiated from the compartment type t_{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.

Input-output rules considered in [10] will be expressed as rewriting and communication rules.

2.2 kP System Execution Strategy

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 7. *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;*
- $\sigma = \{r_1, \ldots, r_s\}$ – *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 [15];*
- $\sigma = \sigma_1 \& \ldots \& \sigma_s$, *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.*

Remark 3. Let us suppose that a certain order relationship exists, e.g., $r_1, r_2 > r_3, r_4$, which means that when weak priority is applied, the first two rules are executed first, if possible, then the next two. If both are executed with maximal parallelism, this is described by $\{r_1, r_2\}^\top \{r_3, r_4\}^\top$.

Remark 4. The result of a computation will be the number of objects collected in the output compartment. For a kP systems $k\Pi$, the set of all these numbers will be denoted by $M(k\Pi)$.

2.3 kP System Examples

In this section we illustrate the newly introduced P system model with some examples.

Example 2. Let us consider the set of component types
$T = \{t_1, t_2, t_3\}$, where $t_1 = (R_1, \sigma_1)$, $t_2 = (R_2, \sigma_2)$, $t_3 = (R_3, \sigma_3)$, with
$R_1 = \{r_1 : a \to a(b,2)(c,3)\ \{\geq p\}; r_2 : p \to p; r_3 : p \to \lambda\}$, and $\sigma_1 = Lab(R_1)^\top$,
$R_2 = \{r_1 : b \to (b,0)c\ \{\geq p\}; r_2 : p \to p; r_3 : p \to \lambda\}$, and $\sigma_2 = Lab(R_2)^\top$,
$R_3 = \emptyset$ and $\sigma_3 = Lab(R_3)^\top$.
A kP system with $n = 4$ compartments is $k\Pi_1 = (A, \mu, C_1, \ldots, C_4, 1)$, where
$A = \{a, b, c, p\}$, $C_1 = (t_1, w_{1,0}), C_2 = (t_2, w_{2,0}), C_3 = (t_2, w_{3,0}), C_4 = (t_3, w_{4,0})$;
with $w_{1,0} = a^3 p$, $w_{2,0} = w_{3,0} = p$, $w_{4,0} = \lambda$;
μ is given by the graph with nodes $\{C_1, C_2, C_3, C_4\}$ and edges $\{C_1, C_2\}, \{C_1, C_3\}$, $\{C_1, C_4\}$.

One can note that we do not use targets for objects meant to stay in the current compartment (i.e., we have $r_1 : a \to a(b,2)(c,3)\ \{\geq p\}$ instead of $r_1 : a \to (a,1)(b,2)(c,3)\ \{\geq p\}$). The rule r_1 in R_2 simulates an input/output rule [10] which is meant to bring a c from the environment (0) and to send out a b instead.

In this example there are only rewriting and communication rules; some rules have a guard, $\geq p$ (p is a promoter), others do not have any and in each compartment the rules are applied in maximal parallel way in every step, as indicated by

σ_j, $1 \leq j \leq 3$. As two instances of the compartment type t_2, C_2, C_3, appear in the system, when the rule r_1 from the compartment C_1 is applied, the object b goes non-deterministically to one of the two compartments labelled 2 (from t_2) as long as p remains in compartment C_1; object c goes always to the compartment C_4, of type t_3.

The initial configuration of $k\Pi_1$ is $M_0 = (a^3p, p, p, \lambda)$. The only applicable rules are r_1, r_2 and r_3 from C_1 and r_2, r_3 from C_2, C_3. If r_1, r_2 are chosen in C_1 and r_2 in C_2, C_3, then a^3p is rewritten by r_1, r_2 in C_1 and p in C_2, C_3 by r_2; then three a's stay in C_1, three b's go non-deterministically to C_2, C_3, three c's go to compartment C_4, and each p in C_2, C_3 stays in its compartment. Let us assume that two of them go to C_2 and one to C_3. Hence, the next configuration is $M_1 = (a^3p, b^2p, bp, c^3)$. If in the next step the same rules are applied identically in the first compartment, C_1, and rules r_1, r_2 are used in C_2 and r_1, r_3 in C_3, then the next configuration is $M_2 = (a^3p, b^2c^2p, bc, c^6)$. If now r_1, r_3 are used in C_1, with r_1 used in the same way and r_1, r_3 in C_2 (no rule is available in C_3) then $M_3 = (a^3, b^2c^4, b^2c, c^9)$; this is a final configuration as there is no p to trigger a further step.

Example 3. Let us reconsider the example above enriched with rules dealing with the system's structure. First the set T will be replaced by $T' = \{t_1, t_2', t_3\}$, where $t_2' = (R_2', \sigma_2')$, with $R_2' = R_2 \cup R_2^{str}$ and $\sigma_2' = Lab(R_2)^\top \& Lab(R_2^{str})^\top$. We can notice that σ_2' tells us that first the rewriting and communication rules are applied in a maximal parallel manner and then one of system's structure rules is chosen to be executed. The set R_2^{str} denotes the set of membrane division rules for t_2', i.e., $R_2^{str} = \{r_4 : []_2 \to []_2[]_2 \ \{\geq b^2 \wedge \geq p\}\}$. The new kP system, denoted $k\Pi_2$, will have the following four compartments:
$C_1 = (t_1, w_{1,0}), C_2' = (t_2', w_{2,0}), C_3' = (t_2', w_{3,0}), C_4 = (t_3, w_{4,0})$.

If the system follows the same pathway as $k\Pi_1$ then M_2 shows a different configuration given that in C_2' after applying R_2 in a maximal parallel manner, R_2^{str} is applied as indicated by σ_2', when the guard of r_4 is true. The compartment C_2' is divided into two compartments, $C_{2,1}, C_{2,2}$, instantiated from the same compartment type t_2, with the content of C_2' and appearing on positions 2 and 3 in the new configuration, $M_2' = (a^3p, b^2c^2p, b^2c^2p, bc, c^6)$; the new compartments, $C_{2,1}, C_{2,2}$, are linked to compartment C_1. Compartment C_3' is not divided as the guard of r_4 is not true for its current multiset. In the next step both $C_{2,1}, C_{2,2}$ are divided as they contain the guard triggering the membrane division rule r_4. The process will stop when either p will be rewritten to λ or b^2 stops coming to these compartments.

Remark 5. If we aim to dissolve one of the compartments instantiated from t_2 or to disconnect it from compartment C_1, once a certain condition is true, for instance $\{\geq b^2 \wedge \geq c^2 \wedge \geq p\}$, then two more rules will be added to R_2^{str}, namely
$r_5 : []_2 \to \lambda \ \{\geq b^2 \wedge \geq c^2 \wedge \geq p\}$, $r_6 : []_2 - []_1 \to []_2; []_1 \ \{\geq b^2 \wedge \geq c^2 \wedge \geq p\}$. The expression σ_2' remains the same, but in this case R_2^{str} contains three elements and at most one is applied at each step, in every compartment with label 2. For this reason σ_2' can also be written as $Lab(R_2)^\top \& Lab(R_2^{str})$.

2.4 Final Remarks

We will recap and summarise how various rules are applied in a compartment instantiated from a compartment type t_{l_i}.

The rules presented in Section 2.1 are applied as follows:

- **A rewrite - communication** rule, $x \to (a_1, t_1) \dots (a_h, t_h)$ $\{g\}$, is executable if and only if
  1. its left hand side multiset, x, is contained within the current multiset;
  2. its associated guard, g, holds (i.e., it is true for the current multiset);
  3. for each right hand side element, (a_j, t_j), $t_j \in T$ (see Definition 1), there is at least one connected compartment of type t_j to receive a_j, otherwise the rule is not applicable.
- **A membrane division** rule, $[x]_{t_{l_i}} \to [y_1]_{t_{i_1}} \dots [y_p]_{t_{i_p}}$ $\{g\}$, is executable if and only if
  1. the multiset on the left hand side, x, is included in the current multiset;
  2. its associated guard, g, is true for the current multiset;
  3. no other structure changing rule (see Definitions 6 and 7) has been applied in the same step;
  4. the compartment instantiated from t_{l_i} is replaced by p compartments instantiated from the compartment types $t_{i_1}, \dots, t_{i_p}$; their contents will be the same as the content of the compartment on the left hand side, but x will be replaced by $y_1, \dots, y_p$, respectively;
  5. all the links of the compartment instantiated from t_{l_i} will be inherited by each of those instantiated from the compartment types $t_{i_1}, \dots, t_{i_p}$.
- **A membrane dissolution** rule, $[]_{t_{l_i}} \to \lambda$ $\{g\}$, will destroy the compartment obtained from l_i and its links, given the guard g is true for the current multiset; no other structure changing rule (see Definitions 6 and 7) has been applied in the same step;.
- **A link creation** rule, $[x]_{t_{l_i}}; []_{t_{l_j}} \to [y]_{t_{l_i}} - []_{t_{l_j}}$ $\{g\}$, is executable if and only if
  1. its left hand side multiset, x, is contained in the current multiset in the compartment;
  2. its associated guard, g, holds;
  3. there exists at least one membrane instance of type t_{l_j} which is not connected to the instance of type t_{l_i} (i.e there is at least one instance to link to); if there are more instances then one will be non-deterministically chosen;
  4. no other structure changing rule (see Definitions 6 and 7) has been applied in the same step.
- **A link destruction** rule is executed when conditions similar to those mentioned for link creation hold.

3 Neural-like P Systems and P Systems with Active Membranes versus kP Systems

In order to prove how powerful and expressive kP systems are, we will show how two of the most used variants of P systems are simulated by kP systems. More precisely, we will show how neural-like P systems and P systems with active membranes are simulated by some reduced versions of kP systems.

Definition 8. *A* neural-like P system *(tissue P system with states) of degree n is a construct $\Pi = (O, \sigma_1, ...\sigma_n, syn, i_0)$ ([14], p. 249), where:*

- *O is a finite, non-empty set of objects, the* alphabet*;*
- *$\sigma_i = (Q_i, s_{i,0}, w_{i,0}, R_i)$, $1 \leq i \leq n$, represents a* cell *and*
  - *Q_i is the finite set of* states *of cell σ_i;*
  - *$s_{i,0} \in Q_i$ is the* initial state*;*
  - *$w_{i,0} \in O^*$ is the* initial multiset *of objects contained in cell σ_i;*
  - *R_i is a finite set of* rewriting and communication rules*, of the form $sw \rightarrow s'xy_{go}z_{out}$; when such a rule is applied, x will replace w in cell σ_i, the objects from y will be sent to neighbouring cells, according to the transmission mode (see Remark 6) and the objects from z will be sent out into the environment; cell σ_i will move from state s to s';*
- *$syn \subseteq \{1, ..., n\} \times \{1, ..., n\}$, the connections between cells,* synapses*;*
- *i_0 is the output cell.*

Remark 6. We discuss here a special class of P systems introduced in Definition 8 that will help us to prove a first result.

1. For neural-like P systems, three processing modes are considered, called "max", "min", "par", and three transmission modes, namely "one", "repl", "spread". For formal definitions and other details we refer to [14].
2. We denote by *simple neural-like P systems* the class of P systems given by Definition 8, where the rewriting and communication rules have the form $sw \rightarrow s'x(a_1, t_1) \cdots (a_p, t_p)$, where t_h, $1 \leq h \leq p$, denotes the target cell (σ_h), and processing mode "max", transmission mode defined by the target indications mentioned in each rule.

Notation. For a given P system, Π, the set of numbers computed by Π will be denoted by $M(\Pi)$.

Theorem 1. *If Π is neural-like P system of degree n, then there is a kP system, Π', of degree n and using only rules of type (a), rewriting and communication rules, simulating Π and such that $M(\Pi') \subseteq M(\Pi) \cup \{2\}$.*

Proof. Let Π be a simple neural-like P systems of degree n, as defined by Remark 6.2. We construct the kP system, Π', as follows: the set of compartment types $T = \{t_1, \ldots, t_n\}$, where $t_i = (R'_i, \sigma_i)$, $1 \leq i \leq n$, (see below the definitions of t_i components) and $\Pi' = (A, \mu, C_1, \ldots, C_n, i_0)$ where:

- $A = O \cup (\bigcup_{1 \leq i \leq n} Q_i) \cup \{\gamma\}$; γ is a new symbol neither in O nor in $\bigcup_{1 \leq i \leq n} Q_i$;
- $\mu = syn$;
- $C_i = (t_i, w'_{i,0}), 1 \leq i \leq n$; and
  - $w'_{i,0} = \gamma, 1 \leq i \leq n$;
  - R'_i contains the following rules:
    1. $\gamma \to s_{i,0} t w_{i,0}$, where $s_{i,0}, w_{i,0}$ are the initial state and initial multiset, respectively, associated with cell σ_i, and $t \in Q^{(i)}_{s_{i,0}}$. For $s \in Q_i$, denote by $Q^{(i)}_s = \{t | t \in Q_i, sx \to ty \in R_i\}$; i.e., $Q^{(i)}_s$ gives, when the cell σ_i is in state s, all the states where σ_i can move to. In the first step, in compartment C_i, a rule $\gamma \to s_{i,0} t w_{i,0}$ is applied and the current multiset becomes $w'_i = s_{i,0} w_{i,0}$.
    2. For each pair $(s,t), t \in Q^{(i)}_s$, there are rules
       $sx_j \to ty_j \in R_i, 1 \leq j \leq p \quad (*)$.
       If there are no rules in R_i from s to t then another pair of states is considered. For the above rules from R_i, $(*)$, the following rules are considered in R'_i:
       $x_j \to y_j$ $\{= s \wedge = t\}$, $1 \leq j \leq p$, and $st \to tq$ $\{\geq x_1 \vee \ldots \vee \geq x_p\}$, $q \in Q^{(i)}_t \quad (**)$.
       In the above guards the notation $\geq x_j$, if $x_j = a_{j,1} \ldots a_{j,l_j}$, denotes $\geq a_{j,1} \wedge \ldots \wedge \geq a_{j,l_j}$. The rules $(**)$ make use of guards; the first p rules are applied iff the current multiset contains one s and one t, whereas the last one is applicable iff at least one or more of the occurrences of one of the multisets x_j, $1 \leq j \leq p$, is included in the current multiset. Clearly, in state s only the rules $(*)$ of Π are applicable for this P system, depending on the availability of the multisets occurring on the left hand side of them; the next state Π is moving to t. Similarly, in Π' only the rules denoted by $(**)$ are applicable; the rule $st \to tq$ $\{\geq x_1 \vee \ldots \vee \geq x_p\}$ is applied once whereas the first p rules are applied as many times as their corresponding $(*)$ rules are applied.

If the set $Q^{(i)}_t$ used in $st \to tq$ $\{\geq x_1 \vee \ldots \vee \geq x_p\}$ of $(**)$ is empty, i.e., there are no rules from state t, then the rule is replaced by $st \to \lambda$. When $Q^{(i)}_{s_{i,0}} = \emptyset$ then the rule $\gamma \to w_{i,0}$ is introduced in R'_i.

At any moment the component C_i of the kP system Π' contains a multiset which is the multiset of σ_i augmented by the current state of σ_i, s, and one of the next states, t, if it exists.

The process will stop in component C_i of Π' when no pair of rules of type $(**)$ is applicable, which means no $sx_i \to ty_i$ rule is applicable in state s.

The multiset $M(\Pi')$ contains $M(\Pi)$ and maybe two states s, t occurring in the last step of the computation. Hence $M(\Pi') \subseteq M(\Pi) \cup \{2\}$. □

Remark 7. A few comments regarding Theorem 1.

1. The above simulation can be assessed with respect to number of compartments, objects and rules as well as the computation steps.

2. When rules $sw \to txy_{go} \in R_i$ are used in the "spread" mode, this means that any $a \in O$ occurring in y may go to any of the neighbours. In this case if $y = y_1 a y_2$ then for each such $a \in O$, in the set R'_i the rule $w \to xy \ \{= s \wedge = t\}$, defined in the proof of Theorem 1, will be replaced by $w \to xy_1(a,j)y_2 \ \{= s \wedge = t\}$, where j the label of one of the neighbours of the current compartment. For "one" mode all a's in y will point to the same target, j, for all neighbours of the compartment i.
3. The transmission replicative mode - when a symbol is sent to all the neighbours, can also be simulated. Indeed if $j_1, \ldots, j_h$ are the neighbours of i, then $w \to xy_1(a,j)y_2$ is transformed into $w \to xy_1(a,j_1)\ldots(a,j_h)y_2$ for each a.
4. If a rewriting rule contains z_{out} on its right side, i.e., $sw \to txy_{go}x_{out}$ then in the set of rules transcribing it, $w \to \alpha$, we will have $\alpha = xy_1(a,j)y_2z'$, where if $z = a_1 \ldots a_k$, then $z' = a'_1 \cdots a'_k$; also rules $a' \to \lambda$ will be added to R'_i, for any $a \in O$. In this way in the next step all the prime elements are removed from the compartment.
5. If we want to simulate the "min" processing mode then this can be obtained by specifying the sequential behaviour of the component i - by changing the definition of σ_i corresponding to the component.

We study now how P systems with active membranes are simulated by kP systems. In this case we are dealing with a cell-like system, so the underlying structure is a tree and a set of labels (types) for the compartments of the system. The system will start with a number of compartment and its structure will evolve. In the study below it will be assumed that the number of compartments simultaneously present in the system is bounded.

Definition 9. *A P system with active membranes of initial degree n is a tuple (see [15], Chapter 11)* $\Pi = (O, H, \mu, w_{1,0}, \ldots, w_{n,0}, R, i_0)$ *where:*

- O, $w_{1,0}, \ldots, w_{n,0}$ *and* i_0 *are as in Definition 8;*
- H *is the set of labels for compartments;*
- μ *defines the tree structure associated with the system;*
- R *consists of rules of the following types*
  - *(a)* rewriting rules*:* $[u \to v]^e_h$*, for* $h \in H$*,* $e \in \{+,-,0\}$ *(set of electrical charges),* $u \in O^+$*,* $v \in O^*$*;*
  - *(b)* in communication rules*:* $u[]^{e_1}_h \to [v]^{e_2}_h$*, for* $h \in H$*,* $e_1, e_2 \in \{+,-,0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(c)* out communication rules*:* $[u]^{e_1}_h \to []^{e_2}_h v$*, for* $h \in H$*,* $e_1, e_2 \in \{+,-,0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(d)* dissolution rules*:* $[u]^e_h \to v$*, for* $h \in H \backslash \{s\}$*, s denotes the skin membrane (the outmost one),* $e \in \{+,-,0\}$*,* $u \in O^+$*,* $v \in O^*$*;*
  - *(e)* division rules for elementary membranes*:* $[u]^{e_1}_h \to [v]^{e_2}_h [w]^{e_3}_h$*, for* $h \in H$*,* $e_1, e_2, e_3 \in \{+,-,0\}$*,* $u \in O^+$*,* $v, w \in O^*$*;*

The following result shows how a P system with active membranes starting with n_1 compartments and having no more than n_2 simultaneously present ones can be simulated by a kP system using only rules of type (a).

Theorem 2. *If Π is a P system with active membrane having n_1 initial compartments and utilising no more than n_2 compartments at any time, then there is a kP system, Π', of degree 1 and using only rules of type (a), rewriting and communication rules, such that Π' simulates Π.*

Proof. Let us denote $J_0 = \{(i,h)|1 \leq i \leq n_2, h \in H\}$; for a multiset $w = a_1 \dots a_m$, (w,i,h), $(i,h) \in J_0$, denotes $(a_1,i,h)\dots(a_m,i,h)$. Let us consider the P system with active membranes, $\Pi = (O, H, \mu, w_{1,0}, \dots, w_{n_1,0}, R, i_0)$. The polarizations of the n_1 compartments are all 0, i.e., $e_1 = \dots = e_{n_1} = 0$.

We construct Π' using $T = \{t_1\}$, where $t_1 = (R'_1, \sigma_1)$ (where R'_1 and σ_1 will be defined later) as follows:
$\Pi' = (A, \mu', C_1, 1)$ where:

- $A = \bigcup_{(i,h)\in J_0}\{(a,i,h)|a \in O \cup \{+,-,0\} \cup \{\delta\}\}$, where δ is a new symbol; let us denote by $= \overline{\delta_{all}}$ the guard $\neg = (\delta,1,1) \wedge \dots \wedge \neg = (\delta, n_2, \mid H \mid)$, $\mid H \mid$ is the number of elements in H ($= \overline{\delta_{all}}$ stands for none of the (δ,i,h), $(i,h) \in J_0$);
- $\mu' = []_1$;
- $C_1 = (t_1, w'_{1,0})$, and
  - $w'_{1,0} = (w_{1,0},1,h_1)\dots(w_{n_1,0},n_1,h_{n_1})(e_1,1,h_1)\dots(e_{n_1},n_1,h_{n_1})$, $e_1 = \dots = e_{n_1} = 0$; let $J_c = J_0 \setminus \{(i,h_i)|1 \leq i \leq n_1\}$ (J_c denotes indexes available for new compartments and $J_0 \setminus J_c$ the set of indexes of the current compartments);
  - R'_1 contains the following rules
    1. for each $h \in H$ and each rule $[u \to v]^e_h \in R$, $e \in \{+,-,0\}$, we add the rules $(u,i,h) \to (v,i,h)$ $\{= (e,i,h) \wedge = \overline{\delta_{all}}\}$, $1 \leq i \leq n_2$; these rules are applied to every multiset containing elements with $h \in H$, only when the polarization (e,i,h) appears and none of the (δ,j,h') appears;
    2. for each $h \in H$ and each rule $u[]^{e_1}_h \to [v]^{e_2}_h \in R$, $e_1,e_2 \in \{+,-,0\}$, we add the rules $(u,j,l)(e_1,i,h) \to (v,i,h)(e_2,i,h)$ $\{= \overline{\delta_{all}}\}$, $1 \leq i \leq n_2$, j is the parent of i of label l; these rules will transform (u,j,l) corresponding to u from the parent compartment j to (v,i,h) corresponding to v from compartment i of label h, the polarization is changed; for each polarization, (e_1,i,h) only one single rule can be applied at any moment of the computation;
    3. for each $h \in H$ and each rule $[u]^{e_1}_h \to []^{e_2}_h v \in R$, $e_1,e_2 \in \{+,-,0\}$, we add the rules $(u,i,h)(e_1,i,h) \to (v,j,l)(e_2,i,h)$ $\{= \overline{\delta_{all}}\}$, $1 \leq i \leq n_2$, j is the parent of i of label l;
    4. for each $h \in H$ and each rule $[u]^e_h \to v \in R$, $e \in \{+,-,0\}$, we add the rules $(u,i,h)(e,i,h) \to (v,j,l)(\delta,i,h)$ $\{= \overline{\delta_{all}}\}$, $1 \leq i \leq n_2$, j is the parent of i of label l; all the elements corresponding to those in compartment i must be moved to j - this will happen in the presence of (δ,i,h) when no other transformation will take place; this is obtained by using rules $(a,i,h) \to (a,j,l)$ $\{= (\delta,i,h)\}$, $a \in O$ and $(\delta,i,h) \to \lambda$ $\{= (\delta,i,h)\}$; the set of available indexes will change now to $J_c = J_c \cup \{(i,h)\}$;

5. for each $h \in H$ and each rule $[u]_h^{e_1} \rightarrow [v]_h^{e_2}[w]_h^{e_3} \in R$, $e_1, e_2, e_3 \in \{+, -, 0\}$; if j_1, j_2 are the indexes of the new compartments, we add $(\underline{u}, i, h)(e_1, i, h) \rightarrow (v, j_1, k_1)(e_2, j_1, k_1)(w, j_2, k_2)(e_3, j_2, k_2)(\delta, i, h)$ $\{= \overline{\delta_{all}}\}$, $1 \leq i \leq n_2$; the content corresponding to compartment i should be moved to j_1 and j_2, hence rules $(a, i, h) \rightarrow (a, j_1, k_1)(a, j_2, k_2$ $\{= (\delta, i, h)\}$, $a \in O$ and finally $(\delta, i, h) \rightarrow \lambda$ $\{= (\delta, i, h)\}$; J_c is updated, $J_c = J_c \cup \{(i, h)\} \setminus \{(j_1, k_1), (j_2, k_2)\}$.

The size of the multiset obtained in i_0 by using Π computation is the same as the size of the multiset in Π, when only (a, i_0, h) are considered, minus 1 (the polarization is also included). □

4 Case Study - Static Sorting

In this section we analyse the newly introduced kP systems by comparing them with established P system classes by using them to specify a static sorting algorithm. This algorithm was first written with symport/antiport rules [4] and then reconsidered in some other cases [2]. The specification below mimics this algorithm.

4.1 Static Sorting with kP Systems

Let us consider a kP system having the following $n = 6$ compartment types: $t_i = (R_i, \sigma_i)$ and corresponding compartments $C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, where $w_{1,0} = a^3; w_{2,0} = a^6p; w_{3,0} = a^9; w_{4,0} = a^5p; w_{5,0} = a^7; w_{6,0} = a^8p$.

The rules of the set R_i, $1 \leq i \leq n$, are :
$r_1 : a \rightarrow (b, i-1)$ $\{\geq p\}$, only for $i > 1$
$r_2 : p \rightarrow p'$
$r_3 : p' \rightarrow (p, i-1)$, for i even and $r_3' : p' \rightarrow (p, i+1)$, for i odd
$r_4 : ab \rightarrow a(a, i+1)$, $i < n$
$r_5 : b \rightarrow a$, $i < n$.

We assume that any two compartments, C_i, C_{i+1}, $1 \leq i < n$, are connected. The aim of this problem is to order the content of these compartments such that the highest element (a^9) will be in the leftmost compartment, C_1, and the smallest one (a^3) in the rightmost compartment, C_n, ($n = 6$).

Remark 8. The functioning of the kP systems is presented below:

- $A = \{a, b, p, p'\}$ is the set of objects;
- the rule r_1 is absent from the compartment C_1;
- the last two rules, r_4, r_5, are only present in compartments C_1 to C_{n-1};
- for $n = 2k + 1$ we need an auxiliary compartment, C_{n+1}, which will start with an initial multiset p and will contain a set of rules with $r_2 : p \rightarrow p'$ and $r_3 : p' \rightarrow (p, n)$; whereas C_n should have an additional rule $r_3' : p' \rightarrow (p, n+1)$;

- in each compartment C_i, $\sigma_i = \{r_1, r_2, r_3, r_4\}^\top \{r_5\}^\top$, if i is even; for odd values of i, r_3 is replaced by r_3'; σ_i tells us that firstly the rules from the first set are applied in a maximal parallel manner and then r_5, also in a maximal way;
- σ_i describes an order relationship, $r_1, r_2, r_3, r_4 > r_5$; so we can replace this kP system by a P system with promoters and having an order relationships on the set of rules associated with each membrane.

The table below presents the first steps of the computation. In the first step the only applicable rules are r_1, r_2; given the presence of p, rule r_1 moves all a's from each even compartment to the left compartment as b's and rule r_2 transforms p into p'. Next, rules r_3, r_4, r_5 are applicable; first r_3 and r_4 are applied, this means p' is moved as p to the left compartments and for each ab an a is kept in the current compartment and a b is moved as an a to the right compartment; finally, the remaining b's, if any, are transformed into a's. These two steps implement a sort of comparators between two adjacent compartments moving to the left bigger elements. In the previous steps the comparators have been considered between odd and even compartments. In the next step p's appear in even compartments and the comparators are now acting between an even and an odd compartment. The algorithm does not have a stopping condition. It must stop when no changes appear in two consecutive steps. Given that the algorithm must stop in maximum $2(n-1)$ steps, then we can introduce such a counter, c, in each compartment and rules $c \to c_1$, $c_i \to c_{i+1}$, $1 \leq i \leq 2(n-1)-2$ and $c_{2(n-1)-1}p \to \lambda$.

Compartments - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	a^3	a^6p	a^9	a^5p	a^7	a^8p
1	a^3b^6	p'	a^9b^5	p'	a^7b^8	p'
2	a^6p	a^3	a^9p	a^5	a^8p	a^7
3	a^6p'	a^3b^9	p'	a^5b^8	p'	a^7
4	a^6	a^9p	a^3	a^8p	a^5	a^7p
5	a^6b^9	p'	a^3b^8	p'	a^5b^7	p'

Remark 9. **Bounded number of compartment types.** The above solution is using n compartment types for n compartments. As the rules are the same in each compartment, with two exceptions involving the components at both ends of the system (compartments C_1 and C_n), it is natural to look for a solutions with a bounded number of types. If we use the same type everywhere except for the two margins then we face the problem of replacing the rules using targets with different rules where the targets are now the new types; if these are the same we can no longer distinguish between left and right neighbours, so we should have at least two distinct ones. Additionally, we have to distinguish odd and even positions. Consequently, four types, and two more for the two ends are enough. Are there further simplifications? The answer to this question and the solution in this case are left as exercises to the reader.

4.2 Static Sorting with States

We consider the same n-compartment tissue-like P system structure as in the previous subsection. Additionally, in this case, the rules in each compartment use states; an order relationship between rules in each compartment is also considered. Initial states are s_1 in odd compartments and s_0 otherwise; the content of the 6 regions is illustrated by the first line, step 0, of the table below.

The addition of states is potentially very useful from a modelling point of view since many widely-used modelling languages are state-based and, therefore, such rules were a strong candidate for inclusion in our kP system model. However, as shown below, states can be effectively simulated by rewriting rules, as shown below.

For the algorithm considered, the rules in each compartment and the order relationships are as follows

Compartment 1:
$r_1 : s_0x \to s_0y$
$r_2 : s_0y \to s_1x$
$r_3 : s_1ab \to s_0a(a, 2)$
$r_4 : s_1b \to s_0a$
The rules satisfy: $r_1, r_2, r_3 > r_4$.

Compartment i, $2 \leq i \leq n-1$:
$r_1 : s_0a \to s_1(b, i-1)$
$r_3 : s_1ab \to s_0a(a, i+1)$
$r_4 : s_1b \to s_0a$
The rules satisfy: $r_1, r_3 > r_4$.

Compartment n:
$r_1 : s^0a \to s^1b_{n-1}$
$r_2 : s^1x \to s^1y$
$r_3 : s^1y \to s^1z$
$r_4 : s^1z \to s^0x$

Membranes - Step	C_1	C_2	C_3	C_4	C_5	C_6
0	$s^1 : a^3x$	$s^0 : a^6$	$s^1 : a^9$	$s^0 : a^5$	$s^1 : a^7$	$s^0 : a^8x$
1	$s^1 : a^3b^6x$	$s^1 : \_$	$s^1 : a^9b^5$	$s^1 : \_$	$s^1 : a^7b^8$	$s^1 : x$
2	$s^0 : a^6x$	$s^1 : a^3$	$s^0 : a^9$	$s^1 : a^5$	$s^0 : a^8$	$s^1 : a^7y$
3	$s^0 : a^6y$	$s^1 : a^3b^9$	$s^1 : \_$	$s^1 : a^5b^8$	$s^1 : \_$	$s^1 : a^7z$
4	$s^1 : a^6x$	$s^0 : a^9$	$s^1 : a^3$	$s^0 : a^8$	$s^1 : a^5$	$s^0 : a^7x$
5	$s^1 : a^6b^9$	$s^1 : \_$	$s^1 : a^3b^8$	$s^1 : \_$	$s^1 : a^5b^7$	$s^1 : x$

In the case where we have an odd number of compartments, the $n-$th region must contain an y instead of x. Thus the starting configuration for $n = 7$ is the

following:
$w_{1,0} = a^3x; w_{2,0} = a^6; w_{3,0} = a^9; w_{4,0} = a^5; w_{5,0} = a^7; w_{6,0} = a^8; w_{7,0} = a^{13}y.$

4.3 Static Sorting with P Systems Using Polarizations on Membranes

We now use cell-like P systems with active membranes to specify the same algorithm. P systems with active membranes were introduced with the primary aim of solving NP-complete problems in polynomial (often linear) time [15]. The key features of this variant is the possibility of multiplying the number of compartments during the computation process by using membrane division rules in addition to multiset rewriting and communication rules. Each membrane can have one of the three electrical charges $\{+, -, 0\}$ and a rule can only be executed if the membrane has the required electrical charge; a rule can also change the polarization of the membrane when objects cross it (either in or out).

In our static sorting example compartments with two states were used, so, when the algorithm is implemented using electrical charges, it is expected that two electrical charges would suffice. Indeed, from the list of rules below one may observe that 0 and + are the only polarizations utilised.

There is, however, a problem with this approach, arising from the rule application strategy. In P systems with membrane division and polarizations, only one rule which can change the polarization of a membrane can be applied per step [8]. The sorting algorithm however, employs maximal parallel communication rules to operate the *comparator* procedure between membranes. In order to correctly implement this procedure we will accept maximal parallel communication rules which change the charge of the membrane they traverse to/from *if and only if* they target the same final polarization.

In the case of P systems with polarizations on membranes we will use a cell-like structure with $n = 6$ regions defined below with the initial multisets included and initial polarizations; the implementation of the static sorting with P systems with polarization on membranes is using priorities over the sets of rules.

$$\mu = [[[[[[[a^3x_1]_1^0 a^6x_1]_2^+ a^9x_1]_3^0 a^5x_1]_4^+ a^7x_1]_5^0 a^8x_1]_6^+]_{aux}^0$$

Rules:
"Comparator" rules:
$r_1 : a[]_j^0 \rightarrow [b]_j^0, 1 \leq j \leq n;$
$r_2 : [ab]_j^0 \rightarrow a[a]_j^+, 1 \leq j \leq n;$
$r_3 : [b \rightarrow a]_j^0, 1 \leq j \leq n;$

Rules for switching polarities between adjacent membranes:
$r_4 : [x_1 \rightarrow x_2]_j^i, 1 \leq j \leq n;$
$r_5 : [x_2]_j^0 \rightarrow y_1[]_j^+, 1 \leq j \leq n;$
$r_6 : [x_2]_j^+ \rightarrow y_1[]_j^0, 1 \leq j \leq n;$
$r_7 : [y_1 \rightarrow y_2]_j^i, 1 < j \leq n+1;$

$r_8 : y_2[]_j^0 \rightarrow [x_1]_j^+, 1 \leq j \leq n;$
$r_9 : y_2[]_j^+ \rightarrow [x_1]_j^0, 1 \leq j \leq n;$
where $i \in \{0, +\}$; and the order relationship $r_1, r_2, r_4, r_5, r_6, r_7, r_8, r_9 > r_3$.

M/S	$[]_1$	$[]_2$	$[]_3$	$[]_4$	$[]_5$	$[]_6$	$[]_{aux}$
0	$[a^3x_1]^0$	$[a^6x_1]^+$	$[a_9x_1]^0$	$[a^5x_1]^+$	$[a^7x_1]^0$	$[a^8x_1]^+$	$[\_]^0$
1	$[a^3b^6x_2]^0$	$[x_2]^+$	$[a^9b^5x_2]^0$	$[x_2]^+$	$[a^7b^8x_2]^0$	$[x_2]^+$	$[\_]^0$
2	$[a^6]^+$	$[a^3y_1]^0$	$[a^9y_1]^+$	$[a^5y_1]^0$	$[a^8y_1]^+$	$[a^7y_1]^0$	$[y_1]^0$
3	$[a^6]^+$	$[a^3b^9y_2]^0$	$[y_2]^+$	$[a^5b^8y_2]^0$	$[y_2]^+$	$[a^7y_2]^0$	$[y_2]^0$
4	$[a^6x_1]^0$	$[a^9x_1]^+$	$[a_3x_1]^0$	$[a^8x_1]^+$	$[a^5x_1]^0$	$[a^7x_1]^+$	$[\_]^0$
5	$[a^6b^9x_2]^0$	$[x_2]^+$	$[a_3b^8x_2]^0$	$[x_2]^+$	$[a^5b^7x_2]^0$	$[x_2]^+$	$[\_]^0$

There are no additional requirements in the case where $n = 2k + 1$, however we always entail an extra auxiliary membrane to enable *out* communication of the $n-$th membrane, therefore allowing it to switch polarity.

A similar implementation of the static sorting algorithm can be obtained by using P systems with labels on membranes. As illustrated in [1], we can encode electrical charges in strings of the membrane labels, in order to differentiate between the two necessary states. For each membrane h_i we synthesise its complementary label h_i', which is changed to by a communication rule. We leave this as an exercise to the reader.

A number of (preliminary) conclusions can be drawn from the above case study:

- kP systems are conceptually closer to tissue P systems than cell-like P systems; in our case studies, this is reflected by the similarity between the specifications using kP systems and tissue P systems, respectively. On the other hand, the model realised using the cell-like P system variant is significantly more complex.
- In terms of complexity, the three implementations are roughly equivalent. The kP system executes in each step one more rule then the P system with states; this rule is either r_2 or r_3 (dealing with p). On the other hand, the number of rules applied in each compartment for every step by cell-like P systems is similar to the case of kP systems.

5 Specification Languages

In this section a specification language covering the entire model of the kP system will be described in Section 5.1 and a specification language with a different syntax for a subset of the same model will be presented in Section 5.2.

5.1 Specification Language for kP Systems

Our Kernel P system research is to be complemented by a set of tools targeting the simulation and trace analysis of various models on the one hand, but also the formal

verification by means of automated model checking on the other hand. In order to utilise such applications, a formal, unambiguous representation of the model is required. We have attained this objective by designing a modelling language capable of mapping the kernel P system specification into a machine readable representation. We call this language kP–Lingua.

There are two principal and naturally opposing precepts which influenced and guided the development of kP–Lingua:

1. Conciseness, simplicity, minimalism: the language must consist of a minimal set of (meta-)descriptive symbols, keywords or constructs, such that it satisfies the necessity for an unambiguous syntax.
2. Clarity, coherence, intelligibility: each statement or sequence of statements must overly express an entity with its associated values or a binding between two or more entities in the model; each proposition should be transparent in its context and intuitive, intelligible in the absence of a reference manual.

This proposal stands at the confluence of these orthogonal aspirations and the formal description of the computational model. Since kP systems explicitly apply the *type - instance* paradigm with respect to compartments, a model definition reflects this distinction in its structure, which is bi-partitioned as follows:

1. Type definitions - encompassing the instruction set, organised in accordance with the type's associated execution strategy.
2. Instance definitions and interlinking - establish the set of compartments and related connections, assembling the graph-like structure of membranes.

A type is declared using the keyword `type` followed by the name of the type - any combination of alphanumeric characters excluding the restricted keywords. The body of a type declaration consists of a succession of guarded rules or rule ensembles (choice, arbitrary execution and maximal parallel execution blocks) as specified in the type's execution strategy. A rule is represented as a guarded transition, symbolised by an arrow, between two terms. We illustrate the syntax of a type definition and its constituents with a comprehensive example:

Example 4. A type definition in kP–Lingua.

```
type C1 {
    2a, 3b -> c .
    >= 2c & > 2b : b, c -> a .

    choice {
        b -> 2b .
        < 3b : b ->  3b .
    }

    max {
      a -> a, a(C2), {a, 2b}(C3) .
```

```
    }

    2c -> -(C2) .
    2b -> \-(C2) .
    = 5a : a -> [3a, 3b](C1) [3b](C2) [3a](C3) .
}
```

In this example we define type `C1` with the following sequence of rules: a rewriting rule which takes two `a` objects and three `b` objects and produces a `c`; a guarded rewriting rule which yields an object `a` if and only if there are at least two `c`'s and more than two `b`'s in the compartment the rule is applied on; next we have a `choice` block with two rewriting rules of which one is guarded, followed by a maximally parallel block where the rewrite communication rule is exhaustively executed, producing an object `a` inside the membrane and sending an object `a` to compartments of type `C2`, one `a` and two `b`'s to membranes of type `C3` respectively; next, a link creation rule expends two `c` objects to establish a new connection with an instance of type `C2`; conversely, the following rule breaks a link with a compartment of type `C2`, using two `b`'s; finally, a guarded membrane division rule takes one object `a` and divides the compartment into three distinct compartments of types `C1`, `C2`, `C3` respectively, if the number of `a`'s in the membrane is precisely five.

The newly created cells are initialised with the a copy of the multiset contained in the divisible compartment, however, one `a` is substituted with the multiset denoted by the value enclosed in the square brackets.

In kP–Lingua every statement terminates with a full stop and, similar to other programming languages, each block which groups a set of statements together is enclosed in curly braces.

An instance is declared as a typed multiset which also designates the initial configuration of the compartment. Hence, a membrane of type `C1` containing two `a` and three `b` objects is encoded as `{2a, 3b} (C1)`. We underline our choice of a consistent notation both for type references and multiset values across different declarative contexts. A variable is considered to be a *type variable* if it is enveloped by parentheses. Any other identifier within the scope of a type definition is interpreted as an object and is prefixed by its multiplicity (if greater than one). Membrane instances may also bear an identifier, a variable name nominating a specific compartment in a 'link' statement. We further illustrate instantiation and binding in Example 5.

Example 5. Instantiation and interlinking of compartments expressed in kP–Lingua.

```
m1 {a, b, 3c} (C1).
m2 {10 xx, 10 xy} (C2).
m1 - m2 .
m2 - m3 {} (C2) - {5 xx, 5 xy} (C2) - {m, 2n} (C3) .
m1 - m3 .
```

In the first line, compartment `m1` of type `C1` is declared, with an initial multiset consisting of an `a`, a `b` and three `c` objects. `m2` of type `C2` is defined analogous. The

third statement is a 'link' instruction, connecting the two previously instantiated compartments. In kernel P systems links are bidirectional and consequently, the *connect* binary operator (`-`) is neutral to the order of its arguments. This, however, becomes relevant when we consider a succession of contiguous vertices in our graph of compartments. Line four emphasises a more condensed way to link instances together: the first one is a reference to compartment `m2` which connects to a new compartment of type `C2` with an empty multiset and label `m3`; this is further joined with an anonymous membrane of type `C2` and finally, a second unlabelled instance of type `C3`. The last line of the example connects `m1` with `m3`, demonstrating the necessity for identifiers and referencing when organising instances in a non-linear structure (i.e., one that is more complex than a list).

We conclude this section by noting the two remaining elements which are not featured in the examples, namely membrane dissolution, symbolised by '#' and the arbitrary execution block respectively. We also acknowledge the expressive power of kP–Lingua whose syntax can intuitively represent a kernel P system model with its plethora of components, using no more than four keywords (`type`, `choice`, `arbitrary`, `max`), five delimiters (`()`, {}, `[]`, `:` and `.`), eight relational operators (`=`, `!=`, `<` , `<=`, `>`, `>=`, `&`, `|`) and four meta-symbols (`->`, `-`, `\-`, `#`) to identify an instruction. A complete EBNF formal description of kP–Lingua's syntax is available in the Appendix.

5.2 Specification Language for a Subset of kP Systems

The previous section has described in detail the specification language for kernel P systems (kP systems). However, a previous specification language was provided to specify and simulate simple kernel P systems (skP systems), a simpler version not taking into account some of the detailed current features of this model of computation. The next subsections describe the syntax, the methodology and the software environment provided with this alternative simpler model, illustrating the process of specification and simulation through a case study, the NP-complete Partition problem.

P–Lingua structures

The following structures, including kP–related features, have been added into P–Lingua version 4, which will be shortly available [16]. The current version of P–Lingua describes simple kP systems, a subset of kP systems [11]. Thus, it does not support link creation rules, link destruction rules nor execution strategies (only maximal parallelism). These features might be incorporated in future releases.

Guards: A guard g denoting a relational expression $\#_a(w)\gamma n$ (see Section 3) is represented as $\{\gamma' a * n\}$, where γ' is a representation of γ such as `<` (for $<$), `<=` (for $\leq$), `=` (for $=$), `<>` (for $\neq$), `>=` (for $\geq$) and `>` (for $>$). As illustrative examples, {`<=c*2`} represents the guard denoting $\#_c(w) \leq 2$, whereas{`>=b`} represents $\#_b(w) \geq 1$. As described in Section 4, guards denoting relational expressions (namely relational guards) can be used in Boolean expressions (namely

Boolean guards). Boolean operators involved in Boolean expressions, $\wedge$ and $\vee$, are represented as `&&` and `||`, respectively. For instance, `{<=a*2}&&{<=b}` represents the guard $\{\leq a^2 \wedge \leq b\}$. Similarly, `{<=a*2}||{<=b}` represents the guard $\{\leq a^2 \vee \leq b\}$. $\wedge$ and $\vee$ operators can be combined to describe complex guards, such as
`{<=a*2}&&{<=b}||{<=a*3}&&{<=c*3}`.
A rule r $\{g\}$ is defined as
`@guard` g `?` r.
For instance, rule $a \rightarrow b$ $\{= a^2\}$ is defined as
`@guard {=a*2} ? [a --> b]`.

Membrane initialisation: Membrane initialisation enables users to define membranes in the initial configuration. According to the membrane structure of the system defined, the syntax for membrane differs. In this respect,
`mu(`$label_1$`)+=[`$multiset$`]'`$label_2$`;`
is used for cell–like membrane structures (i.e, those of PDP systems [7]), whereas
`mu(0)*=[`$multiset$`]'`$label$`;`
is used for tissue–like membrane structures, such as those of kernel P systems. The former instruction adds a new membrane labelled $label_2$ with associated multiset $multiset$ as a child of membrane $label_1$, whereas the latter adds a new membrane labelled $label$ with associated multiset $multiset$ to the initial configuration.

New rules: In order to define the currently supported subset of kernel P systems, some rules defined in Section 6 have been incorporated, which are:

Rewriting and communication rules: A rewriting and communication rule $x \rightarrow y$ $\{g\}$, 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, is represented as
`@guard` g `? [`x`]'`t_0 `--> [`a_1`]'`t_1`, ..., [`a_h`]'`t_h,
with t_0 being the current compartment. In contrast to the definition given in Section 6, the P–Lingua implementation of these rules does not require t_0 to be linked to every compartment $t_i, 1 \leq i \leq h$.

Structure changing rules: A structure changing rule
$[x]_{t_{l_i}} \rightarrow [y_1]_{t_{i_1}} \ldots [y_p]_{t_{i_p}}$ $\{g\}$,
where $x \in A^+$ and y_j has the form $y_j = (a_{j,1}, t_{j,1}) \ldots (a_{j,h_j}, t_{j,h_j})$ like in rewriting and communication rules, is represented in P–Lingua as
`@guard` g `? [`x`]'`t_{l_i} `|--> [`y_1`]'`t_{i_1}`, ..., [`y_p`]'`t_{i_p}`;`.
If any $y_j, 1 \leq j \leq p$ contains the special symbol `@d`, then membrane t_{i_j} is dissolved. These rules can be used in conjunction with membrane internal iterators (see below), resulting in arbitrary division and relabelling rules such as
`[a]'1 |--> [b]'2 &{[c,d{i}]'{i}}:{3<=i<=n};`.

Internal iterators: These iterators enable users to define arbitrary, parameter–dependent multisets, membrane structures and guards. These iterators expand the possibilities for defining rules and initial configurations. The syntax for internal iterators is `&{`*items*`}:{`*index_ranges*`}`, except for ∨–joined guards, which is `|{`*items*`}:{`*index_ranges*`}`. Unless otherwise stated, different internal iterators can be combined in the same rule or sentence, but they cannot be nested. Internal iterators can be used within three contexts:

Multiset internal iterators: The syntax for these iterators is `&{`*multiset*`}:{`*index_ranges*`}`. These iterators allow the extension of multisets in rule definitions. For instance, given a value for n and a set of values for $e_i, 1 \leq i \leq n$, the rules $[a \rightarrow b, c_i, d_i^{e_i}, 1 \leq i \leq n]_1$ are represented as `[a --> b, &{c{i}, d{i}*e{i}}:{1<=i<=n}'1;`.

Membrane internal iterators: The syntax for these iterators is `&{[`*multiset*`]'{`*label*`}}:{`*index_ranges*`}`. These iterators allow to specify communications to various compartments occurring on the right–hand side of the rules only. The left–hand side communication cannot be specified by using this mechanism. For instance, given a value for n and values for $e_i, 1 \leq i \leq n$, rewriting and communication rules $x \rightarrow (a_i^{e_i}, t_i), 1 \leq i \leq n$, where $x, a_i \in A, 1 \leq i \leq n$ and t_j indicates a compartment type from T, are represented as
`[x]'`t_0 `--> &{[a{i}*e{i}]'{i}}:{1<=i<=n};`,
with t_0 being the current compartment.

Guard internal iterators: These iterators are a special case, as they have two possible syntaxes, according to the Boolean operators involved. These forms are
`&{`*guard*`}:{`*index_ranges*`} |{`*guard*`}:{`*index_ranges*`}`.
The construct guards joined by ∧ operators (∧–joined guards), whilst the latter uses ∨ operators (∨–joined guards). In addition, they are the only internal iterators which can be nested, in the form of an ∧–joined guard inside an ∨–joined guard. On the other hand, ∨–joined guards cannot be defined into ∧–joined guards. As an example, given a value for n and a value for m,
`@guard |{{&{{<=B{i,j}*2}}:{1<=j<=m}}}:{1<=i<=n} ? [a-->b]'1;`
can be applied iff, prior to the application of the rule, there exists at least one value $i, 1 \leq i \leq n$, such that the cardinality of each object $B_{i,j}, 1 \leq j \leq m$, in membrane 1, is greater than or equal to 2.

There are some constraints regarding indexes in internal iterators; they cannot be part of numerical expressions (such as `&{a{i+1}}:{1 <= i <= 10}`) nor be used as indexes for constants (such as `&{a{g{i}}}:{1 <= i <= 10}`). In addition, names used for indexes in any internal iterator cannot be used anywhere else, including another internal iterator.

A Case Study - Partition Problem

In the previous section we have introduced the subset of kP systems which is included in P–Lingua platform, as well as the main features of the software tool. This section describes a case study to be modelled and simulated, as explained in the next section.

The problem we will focus on is the well-known NP-complete problem called the *partition problem*. This is formulated as follows: let V be a finite set and $weight$, an additive function on V with positive integer values. It is requested to find, if exists, a partition of V, denoted V_1, V_2, such that $weight(V_1) = weight(V_2)$.

A solution to this problem is provided in [9] by using a recogniser tissue P system with cell division and symport/antiport rules. In this case study we adapt the solution to *skP systems*.

Let $V = \{v_1, \ldots, v_n\}$ be a finite set with $weight(v_i) = k_i$, where k_i is a positive integer, $1 \leq i \leq n$. The following *skP system* is built, depending on n (the number of elements in the set), in order to check whether there is a partition, V_1, V_2, with $weight(V_1) = weight(V_2) = k$ (please note we also check that the weights of both subsets are k). The set of component types, $T = \{t_1, t_2\}$, $t_i = (R_i, \sigma_i), 1 \leq i \leq 2$. R_1 and R_2 are given as follows:

- R_1 contains
  $r_{1,1} : S \to (yes, 0)\ \{\geq T\}$,
  $r_{1,2} : S \to (no, 0)\ \{\geq F \wedge < T\}$;
  $r_{1,1}$ or $r_{1,2}$ sends an answer *yes* or *no*, respectively, to the environment;
- R_2 contains
  membrane division rules:
  $r_{2,i} : [A_i]_2 \to [B_i A_{i+1}]_2 [A_{i+1}]_2, 1 \leq i < n$,
  $r_{2,n} : [A_n]_2 \to [B_n X]_2 [X]_2, 1 \leq i < n$;
  these rules generate in n steps all the subsets of V (2^n subsets); each of them being a potential V_1 and V_2 its complement;
  rewriting rules:
  $r_{2,i,j} : v_i v_j \to v\ \{= B_i \wedge \neq B_j \wedge = X \vee \neq B_i \wedge = B_j \wedge = X\}$,
  $1 \leq i < j \leq n$,
  $r_{2,n+1} : X \to Y$; and
  rewriting and communication rules:
  $r_{2,n+2} : Y \to (F, 1)\ \{\geq v_1\ \vee \ldots \vee \geq v_n \vee \neq v^k\}$,
  $r_{2,n+3} : Y \to (T, 1)\ \{< v_1\ \wedge \ldots \wedge < v_n \wedge = v^k\}$.

The execution strategies are given by $\sigma_i = Lab(R_i)^\top, 1 \leq i \leq 2$, i.e., maximal parallelism. The skP system is given by $skII_3(n) = (A, \mu, C_1, C_2, 0)$, where:

- A is the alphabet;
- μ is given by the graph with edge $(1, 2)$;
- $C_1 = (t_1, w_{1,0})$, $C_2 = (t_2, w_{2,0})$, where $w_{1,0} = S$, $w_{2,0} = A_1 code(n)$, with $code(n) = {v_1}^{k_1} \ldots {v_n}^{k_n}$ being the code of the weights of the elements in V, and being k, half the sum of the k_i values.

The computation leads to an answer, *yes* or *no*, in $n+3$ steps. In fact, in n steps all the subsets of V are generated, as it can be observed above. In step $n+1$, in each compartment C_2 every occurrence of an element v_i of the subset of V is paired up with an element v_j of the complement as many times as the weights allow to, by using $r_{2,i,j}$ as many times as possible; objects X are simultaneously transformed into objects Y. The step $n+2$ consists of sending either T or F to compartment C_1 depending on whether all the elements of the subset and their complements are paired up and the number of pairs is k (i.e., the weight of the partition), or the weights of the subsets are different or are not equal to k (rules $r_{2,n+3}$ and $r_{2,n+2}$ are respectively used). Finally, in step $n+3$ the answer is provided by using one of the rules of C_1.

This solution might be easily changed to only verify that $weight(V_1) = weight(V_2)$, by simply removing the condition referred to v^k in guards attached to rules $r_{2,n+2}$ and r_{2n+3}.

skP Systems Simulation

The previous sections have introduced some background details about P–Lingua specification language and its new features with regard to skP systems, along with the description of a case study, the Partition problem, modelled within this approach. This section has the aim of outlining the basic facts concerning the simulation environment to work with the above presented model.

An integrated methodology for modelling, simulation, analysis and formal verification of skP systems was first presented in [11]. This methodology was supported by the software environment provided by MeCoSim and P–Lingua. P–Lingua provides the P systems designer with a powerful and expressive language to specify skP systems or families of them, possibly including variable parameters whose values depend on the specific instance of the skP system to be generated, such as n for the size of the input set. MeCoSim provides a customisable and extensible visual environment to enable the end user interacting with the P system model implementation. The user provides the input data by custom visual tables, and then runs the simulations (by using the simulation engine of pLinguaCore, that includes simulators for skP systems). Finally, custom visual outputs are generated in the form of tables, charts and/or graphs.

The cited methodology has been applied to our case study, that is, partition problem. The described model has been specified in P–Lingua format, having n and the *weights* in $code(n)$ as variable parameters, depending on the specific set to analyse. P–Lingua file and some additional details about the interface and simulation are available at [17].

In addition, a custom application has been defined in MeCoSim, enabling the user entering n, k and the different weights k_1 to k_n for the elements in the input set, as showed in Fig 1. As part of the custom application definition, a mapping is

set to translate the input data into parameters for the model written in P–Lingua, possibly after performing additional tasks over the input data. This way, when the user introduces the specific input data and clicks *Simulate!*, the values for the parameters are calculated, the initial configuration is generated, the specific skP system is instantiated, and then the computation runs until a halting configuration is reached.

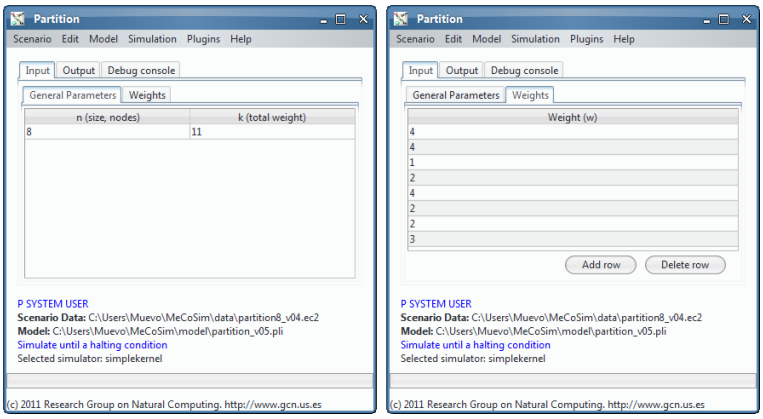


Fig. 1. Input parameters in MeCoSim for Partition

After the simulation has finished, the desired custom outputs are provided to the end user from the results of the computation, depending on the required information to be shown. For instance, a skP systems designer might be interested to know the contents of every membrane, or find out the answer, *yes* or *no*, to the problem. Both outputs have been set in MeCoSim custom application, as shown in Fig 2.

The explained process should be enough to solve the decision problem, but a small additional effort could lead us to provide additional information, such as the elements contained in every specific valid partition, in an automatic way. This additional stuff has been provided by extending the original model with some additional informational objects, and some extra output charts defined for the custom output. The details can be found again in [17], but an example of output chart for a specific partition is shown in Fig. 3.

6 Conclusions

The kP system introduced in this work represents a low level modelling language. Its syntax and informal semantics and some examples have been introduced and discussed. A case study based around a simple sorting algorithm has allowed us to compare different specifications of this using various types of P systems. Finally

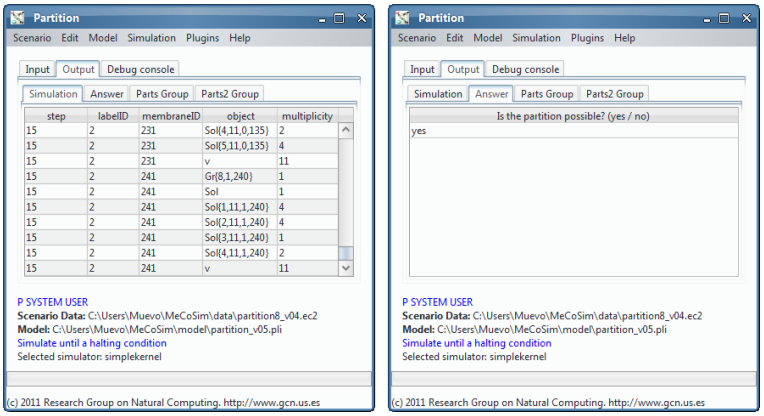


Fig. 2. Output tables in MeCoSim for Partition

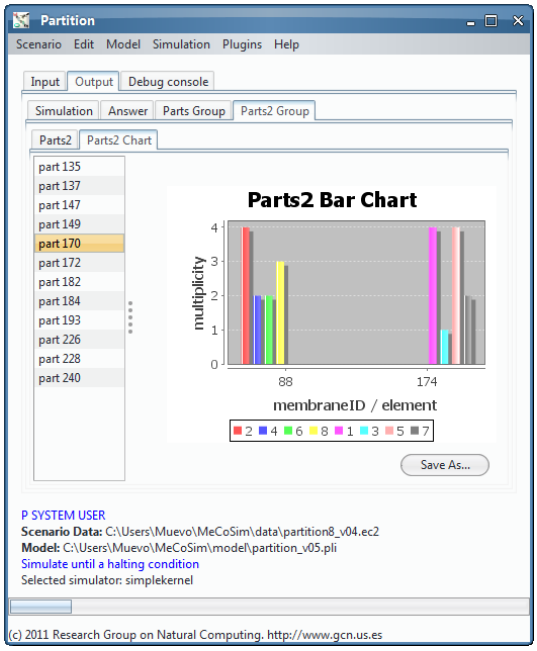


Fig. 3. Partitions Chart in MeCoSim

two specification languages and an implementation are discussed. In the next stage an implementation using the SPIN model checker is expected.

Acknowledgement. The work of MG, FI and LM was partially supported by the MuVet project, Romanian National Authority for Scientific Research (CNCS – UEFISCDI), grant number PN-II-ID-PCE-2011-3-0688; CD's work was supported by a DTA PhD studentship. MGQ, LVC and MJPJ were supported by project TIN 2012-37434 from "Ministerio de Economía y Competitividad" of Spain, and "Proyecto de Excelencia con Investigador de Reconocida Valía P08-TIC-04200" from Junta de Andalucía, both co-financed by FEDER funds. MGQ was also supported by the National FPU Grant Programme from the Spanish Ministry of Education.

References

1. A. Alhazov, L. Pan, Gh. Păun, Trading Polarizations for Labels in P Systems with Active Membranes, *Acta Informatica*, 41, 111-144, 2004.
2. A. Alhazov, D. Sburlan, Static Sorting P Systems. In [5], 215 – 252, 2006.
3. M. Ben-Ari, *Principles of the SPIN Model Checker*, Springer, 2008.
4. R. Ceterchi, C. Martín-Vide, P Systems with Communication for Static Sorting. In *Pre-Proceedings of Brainstorming Week on Membrane Computing, Tarragona, February 2003*, M. Cavaliere, C. Martín-Vide, Gh. Păun, eds., Technical Report no 26, Rovira i Virgili Univ., Tarragona, 101–117, 2003.
5. G. Ciobanu, Gh. Păun, M. J. Pérez-Jiménez, eds., *Applications of Membrane Computing*, Springer, 2006.
6. M. Clavel, F.J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, Maude: Specification and Programming in Rewriting Logic, *Theoretical Computer Science*, 285, 187 – 243, 2002.
7. M.A. Colomer, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic P system over a standard virtual ecosystem. *Natural Computing*, 11, 369–379, 2012.
8. D. Díaz-Pernil, M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, A Uniform Family of Tissue P Systems with Cell Division Solving 3-COL in a Linear Time, *Theoretical Computer Science*, 404, 76 – 87, 2008.
9. D. Díaz-Pernil, M. A. Gutiérrez-Naranjo, M. J. Pérez-Jiménez, A. Riscos-Núñez. A linear time solution to the partition problem in a cellular tissue-like model. *Journal of Computational and Theoretical Nanoscience*, 7, 5, 884 – 889, 2010.
10. M. Gheorghe, F. Ipate, C. Dragomir, Kernel P Systems. In *Membrane Computing, Tenth Brainstorming Week, BWMC 2012, Sevilla, Spain, February 2009*, M. A. Martínez-del-Amor, Gh. Păun, F. Romero-Campero, eds., Universidad de Sevilla, 153 – 170, 2012.
11. M. Gheorghe, F. Ipate, R. Lefticaru, M.J. Pérez-Jiménez, A. Turcanu, L. Valencia, M. Garca-Quismondo, L. Mierlă. 3-COL problem modelling using simple kernel P systems. International Journal of Computer Mathematics, online version (http://dx.doi.org/10.1080/00207160.2012.743712).
12. M. Gheorghe, V. Manca, F.J. Romero-Campero, Deterministic and Stochastic P Ssytems for Modelling Cellular Processes, *Natural Computing*, 9, 457–473, 2010.

13. R. Nicolescu, M. J. Dinneen, Y.-B. Kim, Structured Modelling with Hyperdag P Systems: Part A. In *Membrane Computing, Seventh Brainstorming Week, BWMC 2009, Sevilla, Spain, February 2009*, R. Gutiérrez-Escudero, M. A Gutiérrez-Naranjo, Gh. Păun, I. Pérez-Hurtado, eds., Universidad de Sevilla, 85 – 107, 2009.
14. Gh. Păun, *Membrane Computing: An Introduction*, Springer, 2002.
15. Gh. Păun, G. Rozenberg, A. Salomaa, eds., *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
16. http://www.p-lingua.org – P–Lingua web page.
17. http://www.p-lingua.org/mecosim/doc/case_studies/partition.html – Partition case study at MeCoSim web page.

7 Appendix

7.1 The EBNF formal description of kP–Lingua's syntax

kpsystem ::= {statement};

statement ::= type definition | instantiation | link;

type definition ::= 'type', space, [space], identifier, [space], '{', [space], {(rule | rule ensemble), [space]}, '}';

rule ensemble ::= ('choice' | 'max' | 'arbitrary'), space, [space], '{', [space], {rule}, [space], '}';

rule ::= [guard, [space], ':'], non-empty multiset, '->' (empty multiset | non-empty multiset | targeted multiset | link creation | link destruction | dissolution | division), [space], '.';

multiset ::= empty multiset | non-empty multiset;

empty multiset ::= '{}';

multiset atom ::= [multiplicity], [space], object;

non-empty multiset ::= multiset atom | (multiset atom, [space], ',', [space], non-empty multiset);

type reference ::= '(', identifier, ')';

targeted multiset atom ::= (multiset atom, [space], type reference) | ('{', non-empty multiset, '}', [space], type reference);

targeted multiset ::= targeted multiset atom | (targeted multiset atom, [space], ‘,’ [space], targeted multiset);

link creation ::= ‘-’, [space], type reference;

link destruction ::= ‘\-’, [space], type reference;

dissolution ::= ‘#’;

division atom ::= ‘[’, [space], [non-empty multiset], [space], ‘]’;

division ::= {division atom, [space], type reference [space]};

instance ::= [identifier], space, [space], ‘{’, multiset, ‘}’, [space], type reference;

instantiation ::= (instance, [space], ‘.’) | (instance, [space], ‘,’, [space], instantiation);

link operand ::= instance | identifier;

link ::= (link operand, [space], ‘-’, [space], link operand) | link, [space], ‘-’, [space], link operand, ‘.’;

letter ::= ? A-Za-z ?;

digit ::= ? 0-9 ?;

alphanumeric ::= letter | ‘_’ | digit;

identifier ::= letter, {alphanumeric};

object ::= letter, {alphanumeric | ‘”};

space ::= ? any space character ?;

Rete Algorithm for P System Simulators

Carmen Graciani, Miguel A. Gutiérrez-Naranjo, Agustín Riscos-Núñez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Sevilla
{cgdiaz,magutier,ariscosn}@us.es

Summary. The Rete algorithm is a well-known algorithm in rule-based production systems which builds directed acyclic graphs that represent higher-level rule sets. This allows the rule-based systems to avoid complete re-evaluation of all conditions of the rules each step in order to check the applicability of the rules and, therefore, the computational efficiency of the production systems is improved. In this paper we study how these ideas can be applied in the improvement of the design of computational simulators in the framework of Membrane Computing.

1 Introduction

Rules is one of the most used paradigms in Computer Science for dealing with information. Given two pieces of knowledge V and W, expressed in some language, the rule $V \rightarrow W$ is usually considered as a causal relation between V and W. The interpretation of the rule can change according to the context, but roughly speaking, the rule $V \rightarrow W$ claims that the statement W can be *derived* from the statement V. The problem of knowing if a piece of information G can be obtained via *derivation* from a set of current statements A and a set of rules R arises in a natural way. This is usually called a *reasoning problem* and it will be denoted by $\langle A, R, G \rangle$.

In Computer Science, there are two basic methods for seeking a solution of a reasoning problem, both of them based on the inference rule known as Modus Ponens:

$$\frac{V \quad V \rightarrow W}{W}$$

which allows to obtain W from the rule $V \rightarrow W$ and the piece of information W. The first method is data-driven and it is known as *forward chaining*, the latter is query-driven and it is called *backward chaining* [1]. A study of these methods in the framework of Membrane Computing can be found in [12, 13].

The piece of information V (the left-hand side of the rule or LHS) is usually split into unit pieces $v_1, v_2, \ldots, v_n$. The forward chaining derivation of W (the right

Forward chaining
INPUT: A reasoning problem $\langle A, R, G\rangle$
INITIALISE: $Memory = A, Deduced = \emptyset$
if $G \in Memory$ **then**
return `true`
end if
while $Memory \neq Deduced$ **do**
$Deduced \leftarrow Memory$
for all $(v_1 v_2 \dots v_n \rightarrow W) \in R$ such that
$\{v_1, v_2, \dots, v_n\} \subseteq Deduced$ **do**
if $W = G$ **then**
return `true`
else
$Deduced \leftarrow Deduced \cup \{W\}$
end if
end for
end while
return `false`

Fig. 1. From a computational point of view, the reasoning problem $\langle A, R, G\rangle$, can be solved with the forward chaining algorithm

hand side of the rule or RHS) according to the Modus Ponens via the rule $V \rightarrow W$ needs to check if the statements $v_1, v_2, \dots, v_n$ belong to the set of statements currently accepted. Figure 1 shows a detailed description of the forward chaining method.

The key point of this algorithm is to check for all rules $v_1 v_2 \dots v_n \rightarrow W$ whether $\{v_1, v_2, \dots, v_n\} \subseteq Deduced$ or not. A naive algorithm for this checking consists on

1. Enumerating the rules and the *Deduced* set.
2. Performing a sequential pattern matching between them.

In the framework of Expert Systems [10], a solution to this problem was proposed by Charles L. Forgy in his Ph.D. dissertation at the Carnegie-Mellon University in 1979 [8, 9]. The solution was called the *Rete*[1] *algorithm*. The Rete algorithm places pieces of information in the nodes of a graph and gets faster response time than the naive algorithm of checking one by one the information units in the LHS of the rules.

In spite of the notable differences between the semantics of the rules in Expert Systems and in Membrane Computing, the problem of checking if the restrictions of the LHS of the rule hold is common in both paradigms. In Membrane Computing, a rule of the form

$$u_1^{n_1} \dots u_k^{n_k} \left[v_1^{m_1} \dots v_l^{m_l} \right]_i^{\alpha} \rightarrow u' \left[v' \right]_i^{\alpha'}$$

[1] *Rete* means net in Latin.

is applicable if, in the current configuration, there exists a membrane labelled by i, with polarisation α, containing enough objects $v_1 \dots v_l$ and such that its surrounding membrane contains enough objects $u_1 \dots u_k$. Although the *application* of the rule is different in both paradigms (in Membrane Computing, the objects in the LHS are consumed and the objects in the RHS are created; in Expert Systems the *information* in the LHS does not change, and the one in the RHS is considered *true*), in both cases it is necessary a *checking* of the conditions in order to decide the applicability of the rule.

In this paper we explore if the successful ideas underlying the Rete algorithm can be adapted to the current P systems simulators and contribute to improve their efficiency so that they can face medium-size instances of real life problems.

The paper is organised as follows: Next, we recall some preliminaries on the derivation process in logic and rule-based expert systems. In Section 3 a short description of the Rete algorithm is provided. Section 4 shows how this algorithm can be adapted to P system simulators. Some final remarks and lines for future research are provided in the last section.

2 Production Systems

Next, we recall some preliminaries on production systems and the derivation of pieces of knowledge by using rules.

2.1 Formal Logic Preliminaries

An *atomic formula* (also called an *atom*) is a formula with no deeper structure. An atomic formula is used to express some fact in the context of a given problem. The *universal set* of atoms is denoted with U. A *knowledge base* is a construct $KB = (A, R)$ where $A = \{a_1, a_2, \dots, a_n\} \subseteq U$ is the set of known atoms and R, a set of rules of the form $V \to W$ with $V, W \subseteq U$, is the set of *production rules*.

In propositional logic, the *derivation* of a proposition is done via the inference rule known as Generalised Modus Ponens

$$\frac{P_1, P_2, \dots, P_n \quad P_1 \wedge P_2 \wedge \dots \wedge P_n \to Q}{Q}$$

The meaning of this rule is as follows: if $P_1 \wedge P_2 \wedge \dots \wedge P_n \to Q$ is a production rule and $P_1, P_2, \dots, P_n \subseteq A$ then Q can be derived from this knowledge. Given a knowledge base $KB = (A, R)$ and an atomic formula $g \in U$, we say that g can be derived from KB, denoted by $KB \vdash g$, if there exists a finite sequence of atomic formulas $F_1, \dots, F_k$ such that $F_k = g$ and for each $i \in \{1, \dots, k\}$ one of the following claims holds:

- $F_i \in A$.
- F_i can be derived via Generalised Modus Ponens from R and the set of atoms $\{F_1, F_2, \dots, F_{i-1}\}$

3 Rule-based Expert Systems

Instead of viewing computation as a specified sequence of operations, production systems view computation as the process of applying transformation rules in a sequence determined by the data.

A classical production system has three major components: (1) a global database (or working memory) that contains facts or assertions about the particular problem being solved, (2) a rulebase that contains the general knowledge about the problem domain, and (3) a rule interpreter that carries out the problem solving process.

The facts in the global database can be represented in any convenient formalism. The rules have the form `IF <condition> THEN <action>`

In general, the LHS or condition part of a rule can be any pattern that can be matched against the database. It is usually allowed to contain variables that might be bound in different ways, depending upon how the match is made. Once a match is made, the right-hand-side (RHS) or action part of the rule can be executed. In general, the action can be any arbitrary procedure employing the bound variables. In particular, it can result in addition/elimination of facts to the database, or modification of old facts in the database.

What follows is the basic operation for the rule interpreter (this operation is repeated until no more rules are applicable):

1. The condition part of each rule (LHS) is tested against the current state.
2. If it matches, then the rule is said to be *applicable*.
3. From the applicable rules, one of them is chosen to be applied.
4. The actions of the selected rule are performed.

Production systems may vary on the expressive power of conditions in production rules. Accordingly, the pattern matching algorithm which collects production rules with matched conditions may vary.

3.1 The Rete Algorithm

The Rete algorithm is a well-known algorithm for efficiently checking the many pattern/many object pattern match problem [8], and it has been widely used mainly in production systems. In rule-based systems the checking process takes place repeatedly. This algorithm takes advantage of two empirical observations:

- *Temporal redundancy:* The application of the rules does not change all the current knowledge. Only some pieces of information are changed and the remaining ones (probably, most of them) keep unchanged.
- *Structural similarity:* Several rules can (partially) share the same conditions in the LHS.

This algorithm provides a generalised logical description of an implementation of functionality responsible for matching data from the current state of the

system against productions rules in a pattern-matching. It reduces or eliminates certain types of redundancy through the use of node sharing. It stores partial matches when performing joins between different fact types. This allows the rule-based systems to avoid complete re-evaluation of all facts each step. Instead, the production system needs only to evaluate the changes to working memory.

The Rete algorithm builds directed acyclic graphs that represent higher-level rule sets. They are generally represented at run-time using a network of in-memory objects. These networks match rule conditions (patterns) to facts (relational data tuples) acting as a type of relational query processor, performing projections, selections and joins conditionally on arbitrary numbers of data tuples. In other words the set of rules is preprocessed yielding a network in which each node comes from a condition of a rule. If two or more rules share a condition then they usually share that node in the constructed network. The path from the root node to a leaf node defines a complete rule LHS.

Facts flow through the network and are filtered out when they fail a condition. At any given point, the contents of the network captures all the checked conditions for all the present facts.

This network (a directed graph) has four kind of nodes:

- **Root:** acts as input gate to the network. Receives the changes in the knowledge base and then those tokens pass to the root successors.
- **Alpha nodes:** perform conditions which depend on just one pattern. If the test succeeds, then received token passes to the node successors. There are different alpha nodes depending on the considered pattern.
- **Beta nodes:** perform inter-patterns conditions, for example, if two patterns have a common variable. It receives tokens from two nodes and stores the tokens that arrive from each parent in two different memories. If a token arrives from one of its input, then the condition will be checked against all the tokens in the another input's local memory. For each successfully checked pair, a new token, combining both of them, passes to the node successors.
- **Terminal nodes:** receive tokens which match all the patterns of the LHS of a rule and produce the output of the network.

For example, if the following set of production rules and facts are considered, then the network displayed in Figure 2 will be created. The figure also shows how tokens corresponding to different facts pass through the network.

```
Rule: R1                          Fact: f1  H1(2, 1).
    Exists H2(Y, Z, Z).           Fact: f2  H1(2, 4).
    Exists H1(X, Y > 3).          Fact: f3  H2(4, 3, 3).
         => ...                   Fact: f4  H2(5, 9, 9).

Rule: R2                          Fact: f5  H3(3).
    Exists H2(Y, Z, Z).           Fact: f6  H3(9).
    Exists H3(Z).
         => ...
```

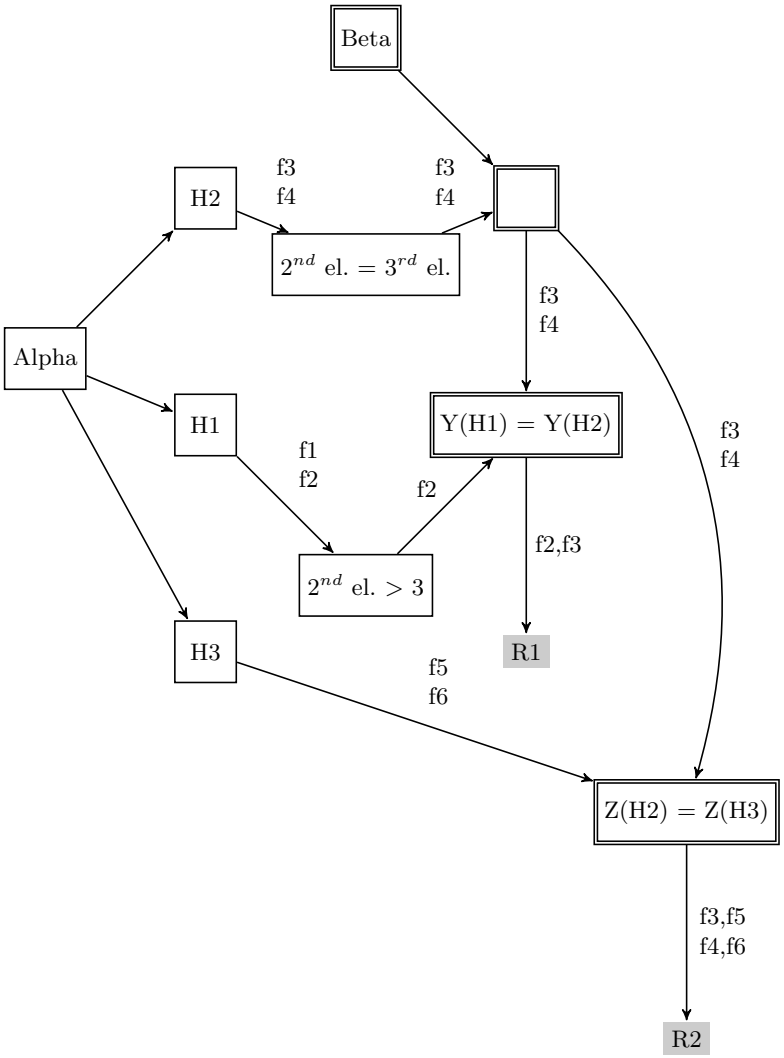


Fig. 2. Example of a Rete network and tokens flow

The most important issue regarding performance is the order of the conditions in the LHS of the rule. This lead us to consider the following strategies in order to improve the efficiency.

- Most specific to most general. If the rule activation can be controlled by a single data, place it first.
- Data with the lowest number of occurrences in the working memory should go near the top.

- Volatile data (ones that are added and eliminated continuously) should go last, particularly if the rest of the conditions are mostly independent.

With those strategies we are trying to minimise (in general) the number of *beta* nodes that will exist in the network and, therefore, the number of checks performed until a token arrived in a terminal node.

4 Membrane Computing

In this section we explore how the Rete algorithm can be adapted to Membrane Computing simulators. For a first approximation we have chosen to focus on rules handling polarisations, which can be written in the following form

$$\mathtt{u}_1^{\mathtt{n}_1} \cdots \mathtt{u}_\mathtt{k}^{\mathtt{n}_\mathtt{k}} [\mathtt{v}_1^{\mathtt{m}_1} \cdots \mathtt{v}_\mathtt{l}^{\mathtt{m}_\mathtt{l}}]_\mathtt{i}^\alpha \rightarrow \mathtt{u}'[\mathtt{v}']_\mathtt{i}^{\alpha'}$$

($\mathtt{k}$ and/or $\mathtt{l}$ can be 0) with $\mathtt{u}_1, \ldots, \mathtt{u}_\mathtt{k}, \mathtt{v}_1, \ldots, \mathtt{v}_\mathtt{l} \in \Gamma$, $\mathtt{u}', \mathtt{v}' \in \Gamma^*$, and either $\mathtt{v}_1^{\mathtt{m}_1} \cdots \mathtt{v}_\mathtt{l}^{\mathtt{m}_\mathtt{l}} \neq \lambda$ or $\mathtt{v}' \neq \lambda$.

This rule is associated with any membrane with label i. In such a rule we can distinguish three kinds of conditions:

- Membrane label is i and charge must be α: $[]_\mathtt{i}^\alpha$
- Outside the membrane there must be at least $\mathtt{n}_\mathtt{j}$ copies of element $\mathtt{u}_\mathtt{j}$: $\mathtt{u}_\mathtt{j}^{\mathtt{n}_\mathtt{j}}$
- Inside the membrane there must be at least $\mathtt{m}_\mathtt{i}$ copies of element $\mathtt{v}_\mathtt{i}$: $[\mathtt{v}_\mathtt{i}^{\mathtt{m}_\mathtt{i}}]$

Now conditions can be reordered in order to follow the proposed strategies for production systems. For example, consider the following rules:

- (R1) $\mathtt{b}^3[\mathtt{ef}]_2^+ \rightarrow \mathtt{u}[\mathtt{v}]_2^\alpha$
- (R2) $\mathtt{b}^3[\mathtt{fe}^2]_2^+ \rightarrow \mathtt{u}'[\mathtt{v}']_2^{\alpha'}$.

We can describe them as follows in order to put at the beginning common conditions:

- (R1) $[]_2^+[\mathtt{f}]\mathtt{b}^3[\mathtt{e}] \rightarrow \mathtt{u}[\mathtt{v}]_2^\alpha$
- (R2) $[]_2^+[\mathtt{f}]\mathtt{b}^3[\mathtt{e}^2] \rightarrow \mathtt{u}'[\mathtt{v}']_2^{\alpha'}$.

To complete the example let us now consider a configuration where a membrane labelled by 2 has positive charge, objects $\{\mathtt{f}^3, \mathtt{e}^7, \mathtt{o}^4, \mathtt{x}\}$ are inside it and the objects $\{\mathtt{c}, \mathtt{b}^8, \mathtt{g}^3\}$ are in the surrounding membrane. This configuration leads to the applicability of both rules.

Figure 3 shows the network associated to the rules of this example and how objects of the considered configuration go through different nodes. When there is not enough objects to pass trough a node they remain in it. Notice that from the output of the network we deduce that each rule can be used at most two times.

Figure 4 shows a generic algorithm to simulate a P system with active membranes using such networks. The halting conditions can be: A prefixed number of repetitions, there is no rule that can be applicable, occurrence of specific objects...

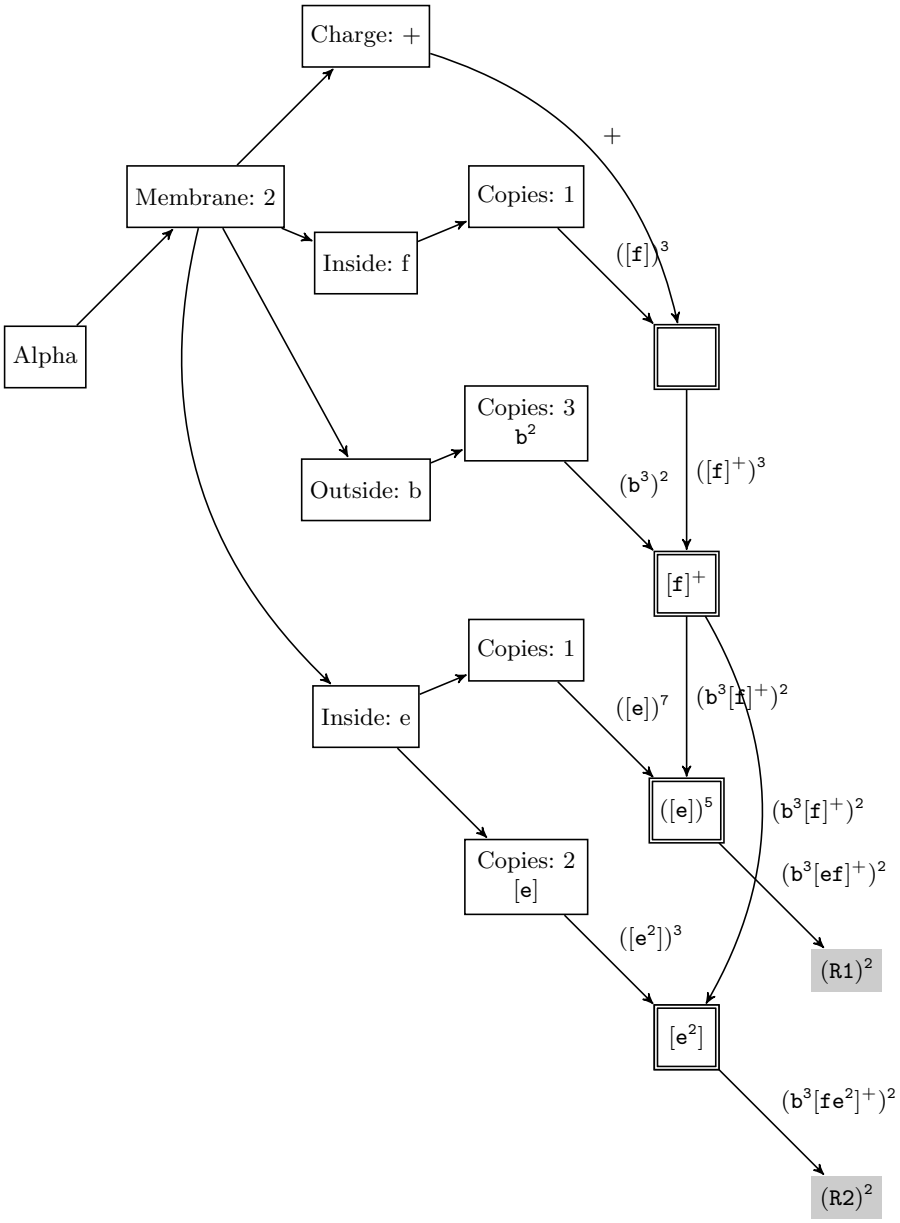


Fig. 3. Rete Network for P system

```
Create the network associated to the rules of the system
Include in the network objects from the initial configuration
while some halting condition do
  while there is no unmarked rule in the output of the network do
    Select one rule from the output of the network (or a set)
    Eliminate in the network all elements from the LHS of that rule
    Mark the charge if its going to change (and marked any rule of
    the output that change the charge for the same membrane)
    Accumulate the objects that must be included in the system
  end while
  Eliminate marks to rules of the output
  Change in the network the marked charges
  Add all the accumulated objects
end while
```

Fig. 4. Generic pseudocode of a simulation algorithm

5 Final Remarks

Let us notice some final considerations:

- As there exists a big number of different P systems models (both syntactically and semantically different), it is not possible to melt together all of them to have a single way to construct the network. So, the basic lines shown in this paper should be adapted to each specific model in order to improve the efficiency of the designed simulator.
- One of the key points of the efficiency of the algorithm is the proper order in the conditions of the LHS of the rule and this is a final choice of the designer of the P system. For example, electrical charge is usually used as a controlling condition, but it is the user who decided its role.
- Little syntactic or semantic changes on the model can have drastic influence on the efficiency of the algorithm. As an illustrative example, we can consider two similar models such that in the first one the membranes are injectively labelled and in the second, two different membranes can share the same label. This apparently slight difference requires a major change in the algorithm.

Recent applications of P system techniques to real-world problems (e.g.,[2, 7]) require more and more efficient simulators. In the similar way to other areas in Computer Science, the availability of huge amount of data, together with the iteration of probabilistic process in an attempt of simulating natural processes needs of more and more efficient algorithms.

In this paper we recover a successful algorithm from the Expert System field and propose a first attempt to consider it in the Membrane Computing framework. The implementation is currently under development, and in principle it will be inserted into a simulator within the P-Lingua framework. However, upon completing the implementation, we are convinced that it will be possible to export this

technique into any other P system simulator. The adaptation of the algorithm has been made by considering that the computer where the software runs has only one processor and, in this way, the software simulation of the P systems is made sequentially in an one-processor machine. Nonetheless, new hardware architectures are being used for simulating P systems [3, 4, 5, 6, 15, 16, 17], so the parallel versions of the Rete algorithm [11, 14] and their relations with parallel simulators of P systems should be considered in the future.

Acknowledgements

The authors acknowledge the support of the Project of Excellence with *Investigador de Reconocida Valía* of the Junta de Andalucía, grant P08-TIC-04200 and the project TIN2012-37434 of the Ministerio de Economía y Competitividad of Spain, both cofinanced by FEDER funds.

References

1. Apt, K.R.: Logic Programming. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B), pp. 493–574. The MIT Press (1990)
2. Cardona, M., Colomer, M.Á., Margalida, A., Palau, A., Pérez-Hurtado, I., Pérez-Jiménez, M.J., Sanuy, D.: A computational modeling for real ecosystems based on P systems. Natural Computing 10(1), 39–53 (2011)
3. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Implementing P systems parallelism by means of GPUs. In: Păun, G. *et al.* (eds.) Workshop on Membrane Computing. Lecture Notes in Computer Science, vol. 5957, pp. 227–241. Springer, Berlin Heidelberg (2009)
4. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulating a P system based efficient solution to SAT by using GPUs. Journal of Logic and Algebraic Programming 79(6), 317–325 (2010)
5. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Hurtado, I., Pérez-Jiménez, M.J.: Simulation of P systems with active membranes on CUDA. Briefings in Bioinformatics 11(3), 313–322 (2010)
6. Cecilia, J.M., García, J.M., Guerrero, G.D., Martínez-del-Amor, M.A., Pérez-Jiménez, M.J., Ujaldon, M.: The GPU on the simulation of cellular computing models. Soft Computing 16(2), 231–246 (2012)
7. Colomer, M.A., Margalida, A., Sanuy, D., Pérez-Jiménez, M.J.: A bio-inspired computing model as a new tool for modeling ecosystems: The avian scavengers as a case study. Ecological Modelling 222(1), 33 – 47 (2011)
8. Forgy, C.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19(1), 17–37 (1982)
9. Forgy, C.L.: On the efficient implementation of production systems. Ph.D. thesis, Department of Computer Science, Pittsburgh, PA, USA (1979)
10. Giarratano, J., Riley, G.: Expert systems: principles and programming. Thomson Course Technology (2005)

11. Gupta, A., Forgy, C., Newell, A., Wedig, R.: Parallel algorithms and architectures for rule-based systems. ACM SIGARCH Computer Architecture News 14(2), 28–37 (1986)
12. Gutiérrez-Naranjo, M.A., Rogojin, V.: Deductive databases and P systems. Computer Science Journal of Moldova 12(1), 80–88 (2004)
13. Ivanov, S., Alhazov, A., Rogojin, V., Gutiérrez-Naranjo, M.A.: Forward and backward chaining with P systems. International Journal on Natural Computing Research 2(2), 56–66 (2011)
14. Kuo, S., Moldovan, D.: The state of the art in parallel production systems. Journal of Parallel and Distributed Computing 15(1), 1 – 26 (1992)
15. Nguyen, V., Kearney, D., Gioiosa, G.: An extensible, maintainable and elegant approach to hardware source code generation in reconfig-P. Journal of Logic and Algebraic Programming 79(6), 383–396 (2010)
16. Peña-Cantillana, F., Díaz-Pernil, D., Berciano, A., Gutiérrez-Naranjo, M.A.: A parallel implementation of the thresholding problem by using tissue-like P systems. In: Real, P. *et al.* (eds.) CAIP (2). Lecture Notes in Computer Science, vol. 6855, pp. 277–284. Springer (2011)
17. Peña-Cantillana, F., Díaz-Pernil, D., Christinal, H.A., Gutiérrez-Naranjo, M.A.: Implementation on CUDA of the smoothing problem with tissue-like P systems. International Journal of Natural Computing Research 2(3), 25–34 (2011)

On Controlled P Systems

Kamala Krithivasan[1], Gheorghe Păun[2,3], Ajeesh Ramanujan[1]

[1]Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai-36, India
kamala@iitm.ac.in, ajeeshramanujan@gmail.com

[2]Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucureşti, Romania, and

[3]Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, ghpaun@gmail.com

Summary. We introduce and briefly investigate P systems with controlled computations. First, P systems with *label restricted* transitions are considered (in each step, all rules used have either the same label, or, possibly, the empty label, λ), then P systems with the computations controlled by languages (as in context-free controlled grammars). The relationships between the families of sets of numbers computed by the various classes of controlled P systems are investigated, also comparing them with length sets of languages in Chomsky and Lindenmayer hierarchies (characterizations of the length sets of ET0L and of recursively enumerable languages are obtained in this framework). A series of open problems and research topics are formulated.

1 Introduction

Most investigations in membrane computing deal with cell-like distributed computing devices (P systems) which process multisets of objects (symbols) in the compartments defined by membranes. That is, the data structure used is the multiset, sets with multiplicities associated with their elements; as a consequence, in a natural way, the results of computations are numbers. However, numerous researches were devoted to computations which have as results strings over given alphabets (in this way, the P systems generate/compute languages). Details and references can be found in [5]. A concise presentation of this research direction, also indicating a series of recent developments and several research topics, is provided by [4].

One of the suggestions in [4] is to associate a control language to a P system, in the way already well-known in formal language theory, e.g., in the case of

context-free controlled grammars (see [2]). The difficulty in the case of P systems is the parallelism of computations: arbitrarily many rules can be used in the same step. There are two ways to overcome this difficulty. The first one, followed in [1], assumes the computations sequential, but here we follow the way suggested in [6] and further explored in [7]: in a step one may use only rules with the same label from a given set of labels, maybe also rules having no label (we say that such a rule has an empty label, denoted by λ). (Note that the sequential mode is not obtained as a particular case, considering the rules labeled in a one-to-one manner and without using the empty label: each single rule $r : a \to u$ should be used as many times as a appears in a multiset.)

Several possibilities appear: to allow rules with empty labels or not; in the latter case, to allow steps when only rules with empty labels are used or not; to have a control language which is finite, regular, or from a subregular family of languages other than the finite ones. Part of these possibilities will be considered here, for non-cooperating P systems and for catalytic P systems.

Some delicate issues appear in comparison with the standard definition of successful computations in P systems (where successful means halting). In the case when no rule is labeled with λ, then in the end of the control word the computation ends, hence we do not need to consider the halting condition. On the contrary, when λ steps are possible, the halting condition should be preserved, as the computation can continue forever by means of λ-steps without interacting with the control word. Moreover, the rules are used in the maximally parallel manner, which means that if no rule can be applied, then the maximally applicable multiset of rules is the empty one; this means that no rule (with the specified label is applied), but still we consider this as a step of the computation. In the case of rules with a nonempty label, one symbol of the control word is "consumed", hence a change in the system configuration (taking into account both the objects and the control word) is obtained, but a λ-step where no rule is applied changes nothing and the computation can continue forever. That is why we impose the restriction that after a λ-step when no rule can be applied, no further λ-step is permitted. This is important in ensuring the halting of computations. Note that a rule of the form $\lambda : a \to a$ can be applied forever to a multiset which contains the object a: nothing is changed, but the rule is effectively applied, this is not a λ-step when no rule is applied.

As expected, by imposing restrictions on the way the rules of a P system are used the computing power is increased. This is confirmed for several cases, both by comparing the power of classes of controlled P systems to each other and to (the sets of numbers associated with) classes of languages in Chomsky and Lindenmayer hierarchies. In this framework, new characterizations of the length sets of ET0L languages and of recursively enumerable languages are obtained. Still, many problems remain open, while further related research topics can be considered (we formulate part of these problems and topics in the last section of the paper).

As we mentioned before, P systems with computations controlled by means of regular languages were also considered in [1], but in a restricted case: the computations were considered sequential and, moreover, the halting condition was replaced with a more powerful condition.

2 Prerequisites

We assume the reader to be familiar with basic notions and results in formal language theory (for details, one can consult [9]) and in membrane computing (see, e.g., [5] and the area website from [10]), that is why we introduce here only the notations we use.

For an alphabet V, we denote by V^* the set of all strings over V, including the empty string, denoted with λ; $V^* - \{\lambda\}$ is denoted by V^+.

A Chomsky grammar is a tuple $G = (N, T, S, P)$, where N is the nonterminal alphabet, V is the terminal alphabet, $S \in N$ is the axiom, and P is the set of rewriting rules. If the rules are of the forms $A \to aB,\ A \to a$, for $A, B \in N, a \in T$, then the grammar is said to be regular. (We omit the rules of the form $A \to \lambda, A \in N$, as they can be removed without changing the generated language, possibly having only a rule $S \to \lambda$ in the case when $\lambda \in L(G)$; however, as usual in formal language theory, in what follows the empty string is ignored when comparing the power of two string processing devices. Correspondingly, number 0 is ignored when comparing the power of two number computing devices.)

We denote by FIN, REG, RE the families of finite, regular, and recursively enumerable languages, respectively. In general, for a family FL of languages, we denote by NFL the family of length sets of languages in FL; formally, $NFL = \{length(L) \mid L \in FL\}$, where $length(L) = \{|x| \mid x \in L\}$ and $|x|$ is the length of the string x. $NREG$ is the family of semilinear sets of numbers (sometimes denoted by $SLIN_1$), and NRE is the family of sets of numbers which can be computed by Turing machines.

A regularly controlled context-free grammar (with appearance checking) is a 6-tuple $G = (N, T, S, P, K, F)$, where $G_0 = (N, T, S, P)$ is a usual context-free grammar (nonterminal alphabet, terminal alphabet, axiom, set of rules), $K \subseteq Lab^*$ is a regular language over an alphabet Lab of labels associated in a one-to-one manner to rules in P (thus, we can imagine that $K \subseteq P^*$, with the rules considered elements of an alphabet) and $F \subseteq Lab$. A derivation in G is a terminal derivation in G_0 which follows a control word $w \in K$, in the appearance checking mode: if a rule $r : A \to u$ is to be used, r not in F, then the rule must be used, otherwise (if A is not present in the sentential form) the derivation is blocked; if $r \in F$ and A appears in the sentential form, then the rule must be used, but if A does not appear in the sentential form, then the rule is skipped and one passes to the next label indicated by the control word w. All terminal words generated in this way form the language $L(G)$. One knows that context-free grammars (using λ-rules) with regular control languages characterize RE (hence, in terms of length set, characterize NRE).

We will also need the notion of an ET0L system (extended tabled interactionless Lindenmayer system). Such a device is a quadruple $\gamma = (V, T, w, P)$, where V is the total alphabet, $T \subseteq V$ is the terminal alphabet, $w \in V^+$ is the axiom, and P is the finite set of tables; a table is a set of rules of the form $a \to u, a \in V, u \in V^*$, which is *complete*, i.e., for each $a \in V$ there is a rule $a \to u$ in the table. The derivation starts from w; in a derivation step $w \Longrightarrow w'$ we use a table in P, and this means rewriting in parallel all symbols from w using the rules in the table. The generated language, $L(\gamma)$, consists of all strings in T^* generated in this way. The families of languages of this form is denoted by $ET0L$. It is known that using or not λ-rules in ET0L systems makes no difference in the generative power, the same family $ET0L$ is obtained, and that $ET0L \subset RE$ and $NET0L \subset NRE$. If the terminal alphabet T is not present, we have a (non-extended) tabled interactionless Lindenmayer system, in short, a T0L system; then all strings generated are accepted in the language $L(\gamma)$ (hence each step of a derivation produces a string). It is known that $T0L \subset ET0L$ and $NREG \subset NT0L \subseteq NET0L$ (see [8]).

In what concerns the classes of P systems we consider in this paper, they are the *cell-like transition P systems* (in short, P systems), specifically, with *non-cooperating* and with *catalytic* rules. Such a system is a tuple $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m)$ where O is the alphabet of objects, $C \subseteq O$ is the set of catalysts (this component is present only in the catalytic systems and it is omitted in non-cooperating P systems), μ is the membrane structure, with m membranes, w_i is the multiset of objects present in region i of μ in the initial configuration, and R_i is the set of rules present in region i of μ; these rules are of the forms $a \to u$ and $ca \to cu$, where $a \in O, c \in C, u \in (\{b_{here}, b_{out} \mid b \in O\} \cup \{b_{in_j} \mid b \in O, 1 \leq j \leq m\})^*$; the target indication *here* is omitted. If $C = \emptyset$, hence all rules are of the form $a \to u$, then Π is called *non-cooperating*. The computation proceeds in a maximally parallel way and it provides an output only if it halts, a configuration is reached where no rule can be applied. The result of a computation is the number of objects which are sent out of the system during the computation (objects b which appear in the form b_{out} in the right hand side of rules used in the skin region are sent out of the system, into the environment). The number m of membranes in μ is called the *degree* of the system.

The set of numbers generated by a P system Π is denoted by $N(\Pi)$. The family of sets $N(\Pi)$ generated by P systems of degree at most m is denoted by $NP_m(ncoo)$ when using non-cooperating rules and $NP_m(cat_i)$ when using catalytic rules, with at most i catalysts present in the system. If the number of membranes is not bounded, then the subscript m is replaced by $*$. The following results are known: $NP_*(ncoo) = NREG$, $NP_1(cat_2) = NRE$, but the size of the family $NP_*(cat_1)$ is not known (it is believed that it is strictly included in NRE). As only the case of one catalyst is of interest, in what follows we will investigate only this case.

3 Label Restricted P Systems

Consider a catalytic P system $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m)$ (of course, if $C = \emptyset$, then we have a non-cooperating system) and associate with each rule in sets $R_1, \ldots, R_m$ a label, which can be either a symbol from an alphabet H or it can be λ; thus, the rules are written in the form $r : u \to v$, with $r \in H$, or $\lambda : u \to v$. We add then the alphabet of labels to the system, in the form $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, H)$ and we say that Π is *labeled*. (Note that this time the labeling is not necessarily one-to-one like in controlled context-free grammars.)

A computation in a labeled P system Π is *label restricted* if in each step one uses (in the maximally parallel manner) only rules with the empty label and rules labeled with the same label in H. A step where only rules $\lambda : u \to v$ are used is called a *λ-step*.

The computations proceed exactly as in a usual P system: we start from the initial configuration, we proceed through maximally parallel steps (which are label restricted), and we get a result (in the environment) after the computation halts. Only halting computations provide a result.

Two cases can be distinguished: using only labels in H (indicated by lr) or also allowing empty labels (indicated by lr_λ). Correspondingly, we obtain four families of sets of numbers: $NP_*(ncoo, lr), NP_*(cat_1, lr)$, and $NP_*(ncoo, lr_\lambda), NP_*(cat_1, lr_\lambda)$, respectively. (Of course, when the number of membranes is bounded, the subscript of NP specifies the bound.)

Note the important detail that lr_λ indicates that rules $\lambda : u \to v$ are allowed and, moreover, λ-steps are allowed. A possible case of interest would be to allow rules with the empty label, but not λ-steps (i.e., to ask that in each step at least a rule with a non-empty label to be used); this case remains as a research topic.

We will mention now, in the form of lemmas, a series of relations about families defined up to now, and later we will synthesize all of them (as well as some results from the literature) in a diagram theorem.

Lemma 1. *$NP_*(\alpha) = NP_1(\alpha)$ and $NP_*(\alpha, \beta) = NP_1(\alpha, \beta)$, $\alpha \in \{ncoo, cat_1\}, \beta \in \{lr, lr_\lambda\}$.*

Proof. The first equality is known, the second one can be proved in the same way: objects in a membrane i are indexed with i, and then all membranes different from the skin membrane can be omitted, the rules are handling indexed objects in the same way as in the compartments of the initial membrane structure. The target indications in the right hand side of rules are easily implemented by changing the subscripts of objects.

According to this lemma, from now on we will use only P systems with only one membrane, and the subscript $*$ in the notation of the generated families is omitted.

Lemma 2. *$NP(ncoo, \alpha) \subseteq NP(cat_1, \alpha), \alpha \in \{lr, lr_\lambda\}$.*

Proof. Directly from the definition.

Lemma 3. $NP(\alpha, lr) \subseteq NP(\alpha, lr_\lambda)$, $\alpha \in \{ncoo, cat_1\}$.

Proof. Directly from the definition.

Lemma 4. $NP(\alpha) \subseteq NP(\alpha, lr)$, $\alpha \in \{ncoo, cat_1\}$.

Proof. Consider a system $\Pi = (O, C, \mu, w_1, R_1)$ and add to it the set $H = \{r\}$, with the same label r associated with all rules in R_1. Denote by Π' the obtained labeled P system. Clearly, no restriction is imposed on using the rules of Π', hence $N(\Pi) = N(\Pi')$.

Lemma 5. $NET0L \subseteq NP(ncoo, lr)$.

Proof. Let $\gamma = (V, T, w, P)$ be an ET0L system with n tables. Label all rules in table i with $r_i, 1 \leq i \leq n$, and let P_1 be the union of all these tables. We construct the labeled P system

$$\Pi = (O, [\quad]_1, w, R_1, H),$$

where

$$O = V \cup \{\#\},$$
$$R_1 = P_1 \cup \{f : A \to \# \mid A \in V - T\} \cup \{f : \# \to \#\} \cup \{f : a \to (a, out) \mid a \in T\},$$
$$H = \{r_i \mid 1 \leq i \leq n\} \cup \{f\}.$$

In each step of a computation in Π we use either only rules from a table of γ or rules with the label f. If these latter rules are used before completing a terminal derivation in γ, then the trap object $\#$ is introduced, and the computation never halts. In the end of the computation, if the derivation in γ is not terminal, the rules with the label f must be used – in this way we check whether the derivation in γ was terminal; simultaneously, all terminal symbols of γ are sent to the environment, hence the computation halts. Thus, we have, $length(L(\gamma)) = N(\Pi)$.

Somewhat surprisingly, we also have the following result.

Lemma 6. $NRE = NP(cat_1, lr)$.

Proof. We only have to prove the inclusion $\subseteq$, the opposite one is a consequence of the Turing-Church thesis (it can also be proved by a direct, straightforward but cumbersome construction of a Turing machine simulating a label restricted P system).

Let us consider a regularly controlled context-free grammar $G = (N, T, S, P, K, F)$, with $K \subseteq Lab^*$, where Lab is an alphabet of labels associated in a one-to-one manner with rules in P. Consider a regular grammar $G_K = (N_K, Lab, S_K, P_K)$ generating the language K; assume the rules in P_K to be labeled in a one-to-one manner with symbols in a set Lab_K. We construct the labeled catalytic P system

$$\Pi = (O, \{c\}, [\quad]_1, cS_K SE, R_1, H),$$

where:

$$O = N \cup T \cup N_K \cup \{c, E, t, \#\},$$
$$H = \{(s, r) \mid s \in Lab_K, r \in Lab\} \cup \{f\},$$

and the set R_1 is constructed as follows:

1. For $s : X_K \to rY_K \in P_K$ and $r : A \to u \in P$ such that $r \notin F$, we introduce in R_1 the following rules:

   $(s, r) : X_K \to Y_K$,
   $(s, r) : Z_K \to \#,\ Z_K \in N_K, Z_K \neq X_K$,
   $(s, r) : cA \to cu_{here}u_{out},\ \ u_{here}$ contains all nonterminal symbols of u and u_{out} contains all terminal symbols of u, with the subscript *out*,
   $(s, r) : cE \to c\#$,
   $(s, r) : t \to \#$;

2. For $s : X_K \to rY_K \in P_K$ and $r : A \to u \in P$ such that $r \in F$, we introduce in R_1 the following rules:

   $(s, r) : X_K \to Y_K$,
   $(s, r) : Z_K \to \#,\ Z_K \in N_K, Z_K \neq X_K$,
   $(s, r) : cA \to cu_{here}u_{out},\ \ u_{here}$ contains all nonterminal symbols of u and u_{out} contains all terminal symbols of u, with the subscript *out*,
   $(s, r) : t \to \#$;

3. For $s : X_K \to r \in P_K$ and $r : A \to u \in P$ such that $r \notin F$, we introduce in R_1 the following rules:

   $(s, r) : X_K \to t$,
   $(s, r) : Z_K \to \#,\ Z_K \in N_K, Z_K \neq X_K$,
   $(s, r) : cA \to cu_{here}u_{out},\ \ u_{here}$ contains all nonterminal symbols of u and u_{out} contains all terminal symbols of u, with the subscript *out*,
   $(s, r) : cE \to c\#$,
   $(s, r) : t \to \#$;

4. For $s : X_K \to r \in P_K$ and $r : A \to u \in P$ such that $r \in F$, we introduce in R_1 the following rules:

$(s,r) : X_K \to t$,
$(s,r) : Z_K \to \#,\ Z_K \in N_K, Z_K \neq X_K$,
$(s,r) : cA \to cu_{here}u_{out}$, u_{here} contains all nonterminal symbols of u and u_{out} contains all terminal symbols of u, with the subscript *out*,
$(s,r) : t \to \#$;

5. We also consider the following rules:

$$f : E \to \lambda,$$
$$f : t \to \lambda,$$
$$f : Z_K \to \#,\ Z_K \in N_K,$$
$$f : A \to \#, A \in N,$$
$$f : \# \to \#.$$

We start by introducing both the axiom S of G and the axiom S_K of G_K in the initial configuration of Π, together with the catalyst c and the auxiliary object E. The catalyst has the role of ensuring that the rules of P are simulated in a sequential way, not in a parallel one.

The objects X_K ensure the fact that the computation in Π follows the same sequence of rules in P as requested by a control word from K. Together with simulating a derivation step in G_K (performed by a rule $s : X_K \to rY_K$ or $s : X_K \to r$) we also simulate the rule $r : A \to u$ from P, and this is done by the rules of Π with the label (s,r).

Consider a rule $(s,r) : cA \to cu_{here}u_{out}$ in Π, corresponding to the rule $r : A \to u$ in P. If the object A is present and we use the rule $(s,r) : cA \to cu_{here}u_{out}$, this corresponds to a derivation step in G. If the object A is present and, instead of $(s,r) : cA \to cu_{here}u_{out}$ we use the rule $(s,r) : cE \to c\#$, then no result is obtained, the computation never halts. If the object A is not present and $r : A \to u$ is a rule from F (remember that the rules of P are labeled in a one-to-one manner with symbols in Lab), then the rule cannot be applied, hence nothing is changed (the rule $(s,r) : cE \to c\#$ is not present in this case). If the object A is not present and the rule r is not from F, then the trap-object $\#$ is introduced by means of the rule $(s,r) : cE \to c\#$. This object will evolve forever by means of the rule $f : \# \to \#$, hence the computation will never halt. In any moment, in between steps which use rules (s,r), also rules $f : A \to \#, A \in N$, can be used, but this will lead to a non-halting computation.

After ending the simulation of a computation in G, we can use an f-rule – and this must be done, as otherwise the computation is not completed (not halted). In this way, we check whether the derivation in G was a terminal one; if not, a rule $f : A \to \#$ can be effectively used and the computation in Π will never halt.

If the derivation in G_K is not correctly simulated, then again the trap object $\#$ is introduced; this is ensured by the rules of the form $(s,r) : Z_K \to \#$ introduced for each s and r.

In the end of the derivation, if the symbol # was introduced, then the rule $f : \# \to \#$ can be used forever, hence the computation in Π will never stop.

If the derivation in G_K is terminal, hence the object t is introduced, but the derivation in G is not terminal, then for each rule $(s, r) : cA \to z$ which is used from now on we have to also use $(s, r) : t \to \#$ and the computation never halts. Conversely, if the derivation in G ends first, then rules $(s, r)X_K \to z$ can be used only if $r \in F$, otherwise also the rule $(s, r) : cE \to c\#$ must be used, but this does not change the multiset generated by G. Thus, the derivations in G_K and G end in the right way.

Therefore, only (and all) terminal derivations in G which follow control words in the language K can be simulated by halting computations in Π, that is, $N(\Pi) = length(L(G))$.

4 Controlled P Systems

In the previous label restricted computations in a labeled P system all sequences of labels are allowed, which corresponds to using H^* as a control language. The control language can be a particular one, and this leads to *controlled P systems*.

Such a system is of the form $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, H, K)$ where $(O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, H)$ is a labeled P system and $K \subseteq H^*$ is a language in a given family FL. A computation in Π has to follow a control word in K. If the empty label is not used, then the computation stops in the moment when the control word ends, irrespective of the fact whether there are rules which can be applied to the reached configuration. Of course, if rules with the empty label (and hence λ-steps) are allowed, then arbitrarily many λ-steps can be performed in between steps where rules with the labels indicated by the word in K are used. This makes necessary the returning to the halting condition in the case lr_λ: after the last step where rules with a label in H is used, we have no control about the end of the computation, arbitrarily many λ-steps can be done. That is why we again impose the restriction that after a λ-step where no rule can be applied we cannot continue with another λ-step (note that such a restriction is not imposed for steps which correspond to non-empty labels: after a step when rules with a label in H should be used, even if no rule is applicable, we pass to the next label in the control word and again it may happen that no rule is applicable, but the control word is "consumed" symbol by symbol).

The computation is done in the maximally parallel manner, in the label restriction framework. This means that when rules with a label r have to applied, a maximal multiset of applicable rules with this label is used. In particular, such a maximal multiset may be empty: no rule with label r can be applied. This is a powerful feature, as it works in a way similar to the appearance checking feature in regulated rewriting (in particular, in controlled context-free grammars).

We denote by $NCP_m(\alpha, \beta, FL)$ the family of sets $N(\Pi)$ generated by controlled P systems Π of degree at most m, with rules of types $\alpha \in \{ncoo, cat_1\}$, with the

rules labeled in the sense of $\beta \in \{lr, lr_\lambda\}$, and with the control language in the family FL. As usual, m is replaced with $*$ when no bound on the number of membranes is considered.

Lemma 7. *$NCP_*(\alpha, \beta, FL) = NCP_1(\alpha, \beta, FL)$ for all $\alpha \in \{ncoo, cat_1\}$, $\beta \in \{lr, lr_\lambda\}$, and any family FL.*

Proof. The same ideas as in the proof of Lemma 1 can be used.

Thus, from now on only P systems with only one membrane will be considered and no subscript is used in the notations of the generated families of sets of numbers.

Lemma 8. *$NCP(ncoo, \alpha, FL) \subseteq NCP(cat_1, \alpha, FL)$ for $\alpha \in \{lr, lr_\lambda\}$, $NCP(\alpha, lr, FL) \subseteq NCP(\alpha, lr_\lambda, FL)$, for $\alpha \in \{ncoo, cat_1\}$ and all FL, and $NCP(\alpha, \beta, FL) \subseteq NCP(\alpha, \beta, FL')$ for $\alpha \in \{ncoo, cat_1\}, \beta \in \{lr, lr_\lambda\}$, for all families $FL \subseteq FL'$.*

Proof. Directly from the definitions.

Lemma 9. *$NFL \subseteq NCP(ncoo, lr, FL)$ for all families FL.*

Proof. Let $L \subseteq V^*$ be a language in a family FL. We construct the controlled P system

$$\Pi_L = (\{b\} \cup V, [\ \ ]_1, b, \{a : b \to ba_{out} \mid a \in V\}, V, L).$$

Taking a string $w \in L$ as a control word, the rules of Π_L produce symbol by symbol the string w and halts when the string ends (no λ-step is possible). Therefore, $N(\Pi_L) = length(L)$.

In what follows, we consider only control languages in the families FIN and REG.

Lemma 10. *$NCP(\alpha, lr, FIN) = NFIN, \alpha \in \{ncoo, cat_1\}$.*

Proof. All computations in a controlled P system where no rule has the empty label and the control language is finite are finite, hence the inclusion $\subseteq$ follows. The opposite inclusion follows from Lemma 9.

Lemma 11. *$NCP(ncoo, lr_\lambda, FIN)$ contains non-semilinear sets.*

Proof. Consider the system

$$\Pi = (\{a\}, [\ \ ]_1, a, \{\lambda : a \to aa,\ r : a \to a_{out}\}, \{r\}, \{r\}).$$

After $n \geq 0$ λ-steps (during this time we produce 2^n copies of object a), if we use the rule $r : a \to a_{out}$ (as requested by the control language), then all copies of a should be sent out (hence the rule $\lambda : a \to aa$ should not be used at this step), otherwise the computation will never halt (the rule $r : a \to a_{out}$ cannot be used for a second time). Therefore, $N(\Pi) = \{2^n \mid n \geq 0\}$, which is not a semilinear set.

Lemma 12. *$NCP(ncoo, lr, REG)$ contains non-semilinear sets.*

Proof. Consider the system

$$\Pi = (\{a\}, [\ \ ]_1, a, \{r : a \to aa,\ r' : a \to a_{out}\}, \{r, r'\}, \{r^* r'\}).$$

After $n \geq 0$ steps when we produce 2^n copies of object a by means of rule r, we have to also use rule r', which sends all objects to the environment, hence the computation halts. Therefore, $N(\Pi) = \{2^n \mid n \geq 0\}$.

Lemma 13. *$NP(\alpha) \subseteq NCP(\alpha, lr_\lambda, FIN)$, $\alpha \in \{ncoo, cat_1\}$.*

Proof. Take a P system $\Pi = (O, C, [\ \ ]_1, w_1, R_1)$ and construct the controlled system

$$\Pi' = (\{a\} \cup O, C, [\ \ ]_1, a, \{r : a \to w_1\} \cup \{\lambda : u \to v \mid u \to v \in R_1\}, \{r\}, \{r\}).$$

After one initial step, when producing the initial multiset of Π, the system Π' continues exactly as in Π, with arbitrarily many λ-steps. The computation in Π' halts if and only if the corresponding computation in Π halts, hence we get $N(\Pi) = N(\Pi')$.

Lemma 14. *$NCP(ncoo, lr_\lambda, REG) \subseteq NET0L$.*

Proof. Consider a controlled P system $\Pi = (O, [\ \ ]_1, w_1, R_1, H, K)$ with $K \in REG$. Let $G = (N, H, S, P)$ be a regular grammar for the language K. We construct the following ET0L system:

$$\gamma = (V, \{b\}, Sw_1, U), \text{ where}$$
$$V = N \cup H \cup O \cup \{r_t \mid X \to r \in P, X \in N, r \in H\} \cup \{b, \#\},$$

and U contains the following tables (in each case, so-called completion rules are added, i.e., rules of the form $a \to a$ for all symbols $a \in V$ for which no rule was already specified in the table; we do not explicitly specify these completion rules; remember that in each rule $a \to u$ of Π, the string u is composed of symbols $a \in O$ and $a_{out}, a \in O$; in the rules of the tables below, $b(u)$ denotes the string obtained by replacing each a_{out} from $u \in (O \cup \{a_{out} \mid a \in O\})^*$ by b):

1. $\{X \to rY\}$
   $\cup \{Z \to \# \mid Z \in N, Z \neq X\}$
   $\cup \{r_t \to \#\}$
   $\cup \{a \to b(u) \mid r : a \to u \in R_1\}$
   $\cup \{a \to b(u) \mid \lambda : a \to u \in R_1\}$,
   for each rule $X \to rY \in P,\ X, Y \in N, r \in H$;
2. $\{X \to r_t\}$
   $\cup \{Z \to \# \mid Z \in N, Z \neq X\}$
   $\cup \{r_t \to \#\}$
   $\cup \{a \to b(u) \mid r : a \to u \in R_1\}$
   $\cup \{a \to b(u) \mid \lambda : a \to u \in R_1\}$,
   for each rule $X \to r \in P, X \in N, r \in H$;

3. $\{X \to X \mid X \in N\} \cup \{r_t \to r_t\}$
   $\cup \{a \to b(u) \mid \lambda : a \to u \in R_1\}$;
4. $\{r_t \to \lambda\}$
   $\cup \{Z \to \# \mid Z \in N\}$
   $\cup \{a \to \# \mid$ there is a rule $\lambda : a \to u \in R_1\}$
   $\cup \{a \to \lambda \mid$ there is no rule $\lambda : a \to u \in R_1\}$
   $\cup \{r \to \lambda \mid r \in H\}$.

In each derivation step of γ one both generates a label $r \in H$, according to the rules of the regular grammar G, and one simulates all rules with that label or rules with the empty label (tables of type 1). λ-steps can be intercalated in-between steps which use at least one rule with a label in H (tables of type 3). In the end of the derivation in G one introduces the symbol r_t and one simulates rules with the label r and rules with label λ (tables of type 2).

After introducing a label r_t, one can continue by simulating arbitrarily many λ-steps (tables of type 3). The string can become a terminal one only by using the table of type 4. This table checks whether the control word is completed, that is, the derivation in G is terminal (if this is not the case, then a rule $Z \to \#, Z \in N$, has to be used), whether the computation in Π halted (if not, then rules $a \to \#$ can be used), erases all symbols $a \in O$ for which there is no rule $\lambda : a \to u \in R_1$, all $r \in H$, as well as r_t for $r \in H$. Note that all objects a_{out} were replaced by b, which is the terminal symbol of γ.

The tables cannot be used in the wrong moment, because of the rules of the form $Z \to \#$.

Consequently, $N(\Pi) = length(L(\gamma))$, and this completes the proof.

5 A Synthesis Theorem

Putting together all previous lemmas as well as known relations between families in Chomsky and Lindenmayer hierarchies, we obtain the following result.

Theorem 1. *The relations from Figure 1 hold. The arrows indicate inclusions while the dotted arrows correspond to strict inclusions.*

We have not mentioned in this diagram the position of the family $NT0L$. For it we have the following relations: $NREG \subset NT0L \subseteq NCP(ncoo, lr, REG) \subseteq NET0L$. The inclusion $NT0L \subseteq NCP(ncoo, lr, REG)$ can be proved in a way similar to the proof of Lemma 5.

Lemma 15. $NT0L \subseteq NCP(ncoo, lr, REG)$.

Proof. Let $\gamma = (V, w, P)$ be a T0L system with n tables. Label all rules in table i with $r_i, 1 \leq i \leq n$, let H be the set of all these labels, and let P_1 be the union of all the tables (with labeled rules). Consider the labeled P system

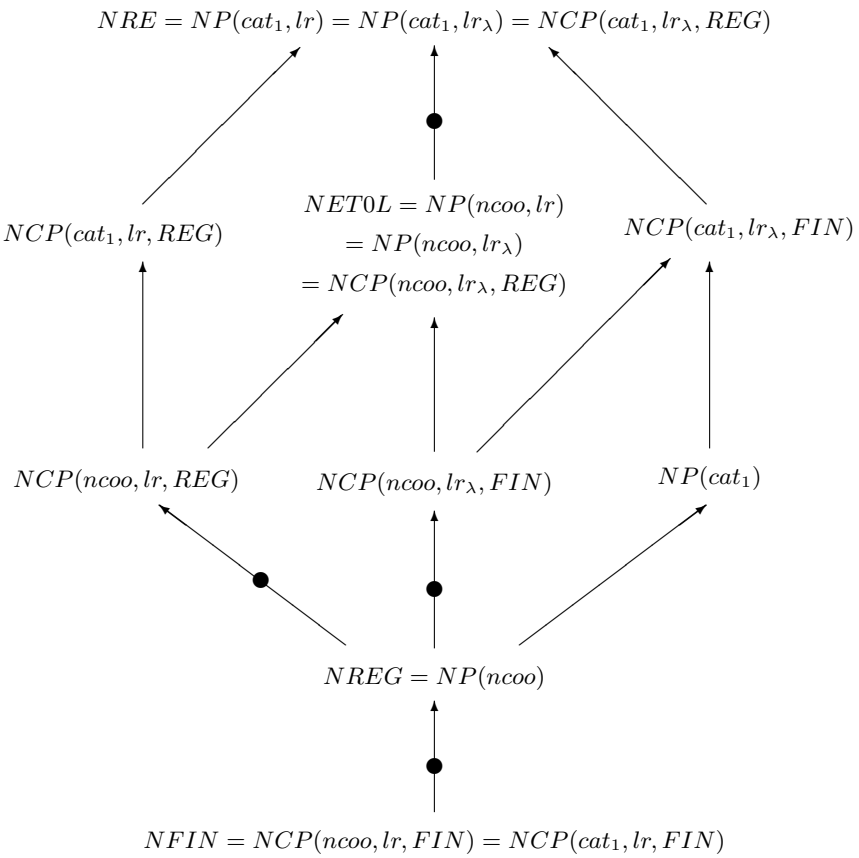


Fig. 1. The hierarchy of families of sets of numbers generated by controlled transition P systems

$$\Pi = (V, [\ \ ]_1, w, R_1, H \cup \{f\}, H^*\{f\}),$$
$$R_1 = P_1 \cup \{f : a \to (a, out) \mid a \in V\}.$$

In each step of a computation in Π we use only rules from a table of γ, chosen according to a control word in H^*. When the control word ends, we sent all objects to the environment, and the computation stops. As H^* contains all possible sequences of tables, we have $length(L(\gamma)) = N(\Pi)$.

Note that a similar language H^* cannot be used in the ET0L case, because we have to check whether the computation in Π ends when the ET0L system has generated a terminal string.

6 Further Research Topics

First of all, it is an open problem whether or not the inclusions in Figure 1 which were not shown to be proper are strict and whether the families not linked by a path in this diagram are comparable. Note the difference between families $NCP(\alpha, lr, FL)$ and $NP(\alpha, lr_\lambda)$: the halting is different in the two cases and this makes difficult their comparison (when we do not have λ-steps and we use a control word, the computation halts when the control word ends, without checking whether further rules could be applied to the obtained configuration).

Then, besides the lr and lr_λ cases considered here, further possibilities seem to be of interest. First, we may impose that in each step when a nonempty label r is indicated by the control word, at least one rule with this label is used (hence we do not allow steps where the maximal multiset of applicable rules is empty – or it contains only rules with the empty label). Also, in the lr_λ case, we may not allow λ-steps after ending the control word.

Besides considering sets of numbers computed by P systems, we can also consider vectors of numbers, counting the multiplicity of different objects in the output (this corresponds to Parikh sets of languages). Whether or not results different from those in Figure 1 are obtained for vectors remains to be checked. (The question is not trivial, in view of the fact that it is an open problem which of the next inclusions is proper: $NREG \subseteq NP(cat_1) \subseteq NRE$). Another direction of research is to consider other classes of P systems instead of transition P systems. For symport/antiport systems we cannot obtain too much, as these systems are universal even with severe restrictions on the complexity of the systems in terms of the number of membranes and the size of rules. Still, the idea of label restricted computations is useful in the case of small symport/antiport systems: the following example was given in [4]:

$$\Pi = (\{a, b\}, [\ \ ]_1, a, \{a, b\}, \{r_1 : (a, out; aa, in),\ r_2 : (a, out; b, in)\}, 1).$$

After using for some $n \geq 0$ steps rule r_1 (2^n copies of a are obtained in the system), the second rule should be used – otherwise the computation cannot stop. The output is obtained in membrane 1 and we have $N(\Pi) = \{2^n \mid n \geq 0\}$, which is not semilinear. The same result is obtained if the control language $r_1^* r_2$ is added to the system Π.

Similarly, one has to consider the case of spiking neural P systems – the class of P systems where the idea of control words (and hence of label restricted computations) was introduced, [6].

Acknowledgements. The work of Gh. Păun has been supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200, co-financed by FEDER funds. Useful discussions with Rudi Freund during 11th Brainstorming Week on Membrane Computing, Sevilla, 4-8 February 2013, are gratefully acknowledged.

References

1. A. Alhazov, R. Freund, H. Heikenwälder, M. Oswald, Y. Rogozhin, S. Verlan: Sequential P systems with regular control. *Membrane Computing International Conference. CMC 2012, Budapest, Hungary. Invited and Selected Papers* (E.Csuhaj-Varju, M. Gheorghe, G. Vaszyl, eds.), Springer, LNCS 7762, 2013, in press.
2. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory*. Springer, Berlin, 1989.
3. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.*, 61 (2000), 108–143 (see also TUCS Report 208, November 1998, `www.tucs.fi`).
4. Gh. Păun, M.J. Pérez-Jiménez: Languages and P systems: Recent developments, *Computer Sci. J. of Moldova*, 20, 2 (59), 2012, 112–132.
5. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
6. A. Ramanujan, K. Krithivasan: Control words of spiking neural P systems. *Romanian J. of Information Science and Technology*, to appear.
7. A. Ramanujan, K. Krithivasan: Control words of transition P systems. Paper in preparation, 2012.
8. G. Rozenberg, A. Salomaa: *The Mathematical Theory of L Systems*. Academic Press, New York, 1980.
9. G. Rozenberg, A. Salomaa, eds.: *Handbook of Formal Languages*. 3 volumes, Springer, Berlin, 1998.
10. The P Systems Website: `http://ppage.psystems.eu`.

An Application of the PCol Automata in Robot Control

Miroslav Langer[1], Luděk Cienciala[1], Lucie Ciencialová[1], Michal Perdek[1], and Alice Kelemenová[1]

Institute of Computer Science
and
Research Institute of the IT4Innovations Centre of Excellence,
Silesian University in Opava, Czech Republic
{miroslav.langer, ludek.cienciala, lucie.ciencialova, michal.perdek, alice.kelemenova}@fpf.slu.cz

Summary. The P colonies were introduced in [6] as a variant of the bio-inspired computational models called membrane systems or P systems. In [2] we divided agents into the groups according the function they provide; we introduced the modularity on the P colonies. PCol automata are an extension of the P colonies by the tape (see [1]). This is an accepting computational device based on the very simple computational units. In this paper we combine the approach of the modules in the P colonies and of the PCol automata and we introduce the PCol automaton driven robot.

1 Introduction

Recently, the robotics has been more and more expanding and intervening in various branches of science like biology, psychology, genetics, engineering, cognitive science, neurology etc. An effort to create robots with an artificial intelligence which are able to cogitate or solve various types of problems refers to hardware and software limits. Many of these limits are managed to be eliminated by the interdisciplinary approach which allows creating new concepts and technics suitable for the robot control and facture of the new hardware.

Very robot control is often realised by the classical procedures known from the control theory (see [9]), concepts inspired by the biology, evolution concepts (see [5]) or with use of the decentralized approaches (see [8]).

The autonomous robot's behaviour and its control are realized by the control unit. Robots are equipped with the various types of sensors, cameras, gyroscopes and further hardware which all together represents the robots perception. These hardware components provide to the control unit the information about the actual state of the environment in which the robot is present and also the information about the internal states of the robot. After the transformation of these inputs

there are generated a new data which are forwarded to the actuators like the wheels, robotic arm etc. Thus the robot can pass the obstacle by using the sensors and adjusting the speed of the particular wheels. So the objective of the control unit is to transform input signals to the output signals which consequently affect the behaviour of the robot. These changes in the behaviour cause that the robot interacts with the environment and in the sight of the observer the robot seems to be intelligent. Transformation of these signals can be done computationally in various ways with use of the knowledge or fuzzy knowledge systems, artificial neural networks, or just with use of the membrane systems or the P systems as it will be shown in this paper.

The development, design and the realization of the new approaches and techniques through which is possible to realize function of the control unit is one of the key subject of the development of the artificial intelligence and the robotics.

P colonies were introduced in 2004 as abstract computing devices composed from independent single membrane agents, reactively acting and evolving in a shared environment. P colonies reflect motivation from colonies of grammar systems, i.e. the idea of the devices composed from as simple as possible agents placed in a common environment; the system, which produce nontrivial emergent behaviour, using the environment only as the communication medium with no internal rules. P colonies consist of single cells "floating" in a common environment. Sets of rules of cells are structured to simple programs in P colonies. Rules in a program have to act in parallel in order to change all objects placed into the cell, in one derivation step. Objects are grouped into cells or they can appear in their completely passive environment in which these cells act. We assume that the environment contains several copies of the basic environmental objects (denoted in the formal definition of P colonies by e), as many as needed to perform a computation. Moreover the environment can contain also finite number of non-basic objects, and each entity contains a fixed (intuitively small) number of (possibly identical) objects.

Cells as basic computing agents of P colonies are of as much as possible restricted complexity and the capability. Each agent is associated with a small number of objects present inside it and with a set of rules forming programs for processing these objects.

Two types of rules are considered, namely the evolution rules acting inside agents, and the communication rules providing elementary interactions between the agents and the environment.

Each of the evolution rules is able to rewrite one object in the agent into another object which will remain inside this agent. Evolution rule is denoted by $a \rightarrow b$. The communication rules consist in the mutual exchanging of one object inside the agent, and one object in its environment. We denote communication rule by $c \leftrightarrow d$, where the object appearing in the agent is written in the left side of the relation $\leftrightarrow$. Moreover checking rules are considered to extend the abilities of agents follows: assume that communication rule can be chosen from two possibilities with the first one having higher priority. The rule associated with the agent

with greater priority has to be active. The agent checks the possibility to execute the communication rule having higher priority. Otherwise, the second communication rule can be treated. We denote a checking rule being a pair $c \leftrightarrow d/c' \leftrightarrow d'$. A *P colony with checking rules will be called also a P colony with priority.*

The program of an agent allows changing all objects in the cell simultaneously and deterministically by different rules, so the number of objects in an agent is identical with the number of rules in each of its programs.

P colony starts a computation with given objects in the environment and in each agent. We associate a result with a halting computation, in the form of the number of copies of a distinguished object in the environment. Both parallel as well as sequential computational mode of P colonies is discussed depending on the amount of agents acting in one derivation step. In the first case, each agent which can apply any of its programs has to choose one non-deterministically, and apply it; in the sequential case one agent, non-deterministically chosen, is allowed to act. P colonies are computationally complete, i.e. all the number sets computable by Turing machines are computable also by P colonies. This gives the interpretation that the environment is essential as a medium for communication and for storing information during the computation, even with no structure and no information in the environment at the beginning of the computation. The power of cooperating agents of a very restricted form can be dramatically different from the power of individual agents. For overview on P colonies we refer to [7], [3], [4].

Pcol automaton was introduced in order to describe the situation, when P colonies behave according to the direct signals from the environment (see [1]). This modification of the P colony is constructed in order to recognize input strings. In addition to the writing and communicating rules usual for a P colony cells in Pcol automata have also tape rules. Tape rules are used for reading next symbol on the input tape and changing an object in cell(s) to the read symbol. Depending on the way how tape rules and other rules can take a part in derivation process several computation modes are treated. After reading the whole input word, computation ends with success if the Pcol automaton reaches one of its accepting configurations. So, in accordance with finite automata, Pcol automata are string accepting devices based on the P colony computing mechanisms.

Agents can be grouped to different modules specified for some activities as illustrated in [2] for P colonies. This approach will be used in the present paper for Pcol automata illustrated in the case of robot control.
Throughout the paper we assume that the reader is familiar with the basics of the formal language theory.

2 Preliminaries on the P colonies

P colony is a computing device composed from the environment and the independent organisms called agents or cells. The agents live in the environment. Each agent is represented by a collection of objects embedded in a membrane, which

is constant during the computation. A set of programs, which are composed from the rules, is associated with each agent. The rule can be either evolution rule or communication rule. The evolution rules are of the form $a \rightarrow b$. It means that the object a inside the agent is rewritten (evolved) to the object b. The communication rules are of the form $c \leftrightarrow d$. When the communication rule is performed, the object c inside the agent and the object d in the environment swap their places. Thus after execution of the rule, the object d appears inside the agent and the object c is placed in the environment.

In [6] the set of programs was extended by the checking rules. These rules give an opportunity to the agents to opt between two possibilities. The rules are in the form r_1/r_2. If the checking rule is performed, then the rule r_1 has higher priority to be executed over the rule r_2. It means that the agent checks whether the rule r_1 is applicable. If the rule can be executed, then the agent is compulsory to use it. If the rule r_1 cannot be applied, then the agent uses the rule r_2.

The environment contains several copies of the basic environmental object denoted by e. The number of the copies of e in the environment is sufficient, it means that each agent which wants to receive the symbol e from the environment using the communication rule will receive it.

We will handle parallel model of P colonies with checking rules (denoted by $NPCOL_{par}K$) in this paper. At each step of the parallel computation each agent tries to apply one usable program. If the number of applicable programs is higher than one, then the agent chooses one of the rules nondeterministically and the maximal possible number of agents is active at each step of the computation. Each P colony is characterised by three characteristics; the capacity k, the degree n and the height h; denoted by $NPCOL_{par}K(k, n, h)$. The capacity k is the number of the objects inside each agent, the degree n is the number of agents in the P colony, the height h is the maximal number of programs associated with the agent of the P colony.

2.1 Modularity in the therms of P colonies

The research of the P colonies suggested that particular agents are providing the same function during the computation. This served as the inspiration to introduce the modules in the P colonies. In the [2] we grouped agents of the P colony simulating computation of the register machine into the modules. The agents providing subtraction were classified into the subtraction module, agents providing addition were sorted into the addition module, agents controlling the computation were grouped into the control module, etc. The program of simulated register machine is stored in the control module, so changing the program of the register machine does not mean reprograming all the agents of the P colony but the change of the control module.

The inspiration to introduce modularity was found in living organisms where group of cells providing one function evolved into the organs and whole organism is composed by these organs.

In this paper, this approach to define modules will be used for the robot control. One module for each module of the robot (sensors, actuators) will be defined. For planning the robots action will be used the tape; PCol automaton.

2.2 PCol automata

By extending the P colony by the input tape we obtain a string accepting/ recognizing device; the PCol automaton (see [1]). The input tape contains the input string which can be read by the agents. The input string is the sequence of the symbols. To access the tape the agents use special tape rules (T-rules). The rules not accessing the tape are called non-tape rules (N-rules). The computation and use of the T-rules is very similar to the use of the rules in the P colonies. Once any of the agents uses its T-rule, the actual symbol on the tape is considered as read. The only difference between the tape and the environmental symbol is that the tape symbol can access arbitrary many agents at the same time.

Definition 1. *PCol automaton of the capacity k and with n agents, $k, n \geq 1$ is a construct*

$$\Pi = (A, e, V_E, (O_1, P_1), \ldots (O_n, P_n), F), \text{ where}$$

- *A is a finite set, an alphabet of the PCol automaton, its elements are called objects;*
- *e is an environmental object, $e \in A$;*
- *V_E is a multiset over $A - \{e\}$ defining the initial content of the environment;*
- *$(O_i, P_i), 1 \leq i \leq n$ is an i-th agent*
  - *O_i is a multiset over A defining the initial content of the agent, $|O_i| = k$,*
  - *P_i is a finite set of the programs,$P_i = T_i \cup N_i,\ T_i \cap N_i = \emptyset$, where every program is formed from k rules of the following types:*
    - *the tape rules (T-rules for short)*
      - *$a \overset{T}{\rightarrow} b$ are called the rewriting T-rules;*
      - *$a \overset{T}{\leftrightarrow} b$ are called the communicating T-rules;*
    - *the non-tape rules (N-rules for short)*
      - *$a \rightarrow b$ are called the rewriting N-rules;*
      - *$c \leftrightarrow d$ are called the communicating N-rules;*
    - *T_i is a set of tape programs (T-programs for short) consisting from one T-rule and $k-1$ N-rules.*
    - *N_i is a set of non-tape programs (N-programs for short) consisting only from N-rules.*
- *F is a set of accepting configurations of the PCol automaton, each state is a $(n+1)$-tuple $(v_E; v_1, \ldots, v_n)$, where:*
  - *$v_E \subseteq (A - \{e\})^*$ is a multiset of the objects different from the object e placed in the environment;*
  - *$v_i,\ 1 \leq i \leq n$ is a content of the i-th agent;*

The configuration of the PCol automaton is $(n+2)$-tuple $(w_T; w_E; w_1, \ldots, w_n)$, where $w_T \in A^*$ the unprocessed (unread) part of the input string, $w_E \in (A - \{e\})^*$ is a multiset of the objects different from the object e placed in the environment of the PCol automaton and $w_i,\ 1 \leq i \leq n$ is a content of the i-th agent.

The computation starts in the starting configuration defined by the input string on the tape the initial content of the environment and the initial content of the agents. Actual symbol on the input tape we consider as read iff at least one agent uses its T-program in the particular derivation step.

We define the rule r in following way:

$$r = \left(a \xrightarrow{T/\sqcup} b\right) \Rightarrow \begin{cases} left\,(r) = a \\ right\,(r) = b \\ export\,(r) = \varepsilon \\ import\,(r) = \varepsilon \end{cases}$$

$$r = (c \xleftrightarrow{T/\sqcup} d) \Rightarrow \begin{cases} left\,(r) = \varepsilon \\ right\,(r) = \varepsilon \\ export\,(r) = c \\ import\,(r) = d \end{cases}$$

For each configuration $(w_E, w_1, \ldots, w_n)$ we define set of applicable programs $P_{(w_E, w_1, \ldots, w_n)}$:

- $\forall p, p' \in P, p \neq p', p \in P_i, p' \in P_j \Rightarrow i \neq j$
- for each $p \in P$ and $p \in P_i$ $left(p) \cup export(p) = w_i$
- $\bigcup_{p \in P} import(p) \subseteq w_E$

For each configuration $(w_E, w_1, \ldots, w_n)$ we define set of all sets of applicable programs $\mathcal{P}_{(w_E, w_1, \ldots, w_n)}$

For the configuration $(w_E, w_1, \ldots, w_n)$ and the input symbol a we define:

- **t-transition,** $\Rightarrow_t^a$: If there is at least one set of applicable programs $P \in \mathcal{P}$ such that each $p \in P$ is the T-program with T-rule of the form $x \xrightarrow{T} a$ or $x \xleftrightarrow{T} a, x \in A$ and P is the maximal set (there does not exists other set $P' \in \mathcal{P}$ where $|P'| > |P|$ fulfilling stated conditions).
- **n-transition,** $\Rightarrow_n$: If there is at least one set of applicable programs $P \in \mathcal{P}$ such that each $p_i \in P$ is the N-program and P is the maximal set.
- **tmin-transition** $\Rightarrow_{tmin}^a$: If there is at least one set of applicable programs $P \in \mathcal{P}$ such that there exists at least one program P is the T-program and it is in the form $x \xrightarrow{T} a$ or $x \xleftrightarrow{T} a, x \in A$, it can contain also the N-programs and P is the maximal set.
- **tmax-transition** $\Rightarrow_{tmax}^a$: If there is at least one set of applicable programs $P \in \mathcal{P}$ such that $P = P_N \cup P_T$ where P_N is a set of nontape programs and P_T is a maximal set of applicable tape programs of the form $x \xrightarrow{T} a$ or $x \xleftrightarrow{T} a, x \in A$, and $P = P_N \cup P_T$ is maximal;

We denote

$(w_E, w_1, \ldots, w_n) \Rightarrow^{a/\sqcup}_{trans} (w'_E, w'_1, \ldots, w'_n)$,
$trans = \{t, n, tmin, tmax\}$

where: for each j, $1 \leq j \leq n$ for which there exists $p \in P \wedge p \in P_j$, $w'_j = right(p) \cup import(p)$, if there does not exists $p \in P \wedge p \in P_j$ so $w'_j = w_j$; $w'_E = w_E - \bigcup_{p \in P} import(p) \cup \bigcup_{p \in P} export(p)$.

PCol automaton works in the *t(tmax, tmin) mode of computation* if it uses only t- (tmax-, tmin-) transitions. PCol automaton works in the *nt (ntmax or ntmin) mode of computation* if at any computation step it may use a t- (tmax- or tmin-) transition or an n-transition. A special case of the *nt* mode is called *initial*, denoted by *init*, if the computation of the automaton is divided in two phases: first it reads the input strings using t-transitions and after reading all the input symbols it uses n-transitions to finish the computation.

The computation ends by (types of acceptance):

halting (halt) - there does not exist an applicable set of programs corresponding to the computation mode. Computation is successful if it ends and the whole input tape is read.

reading the last input symbol (lastsym) - the computation (successfully) ends if the last input symbol is read and there does not exist set of applicable programs corresponding to the computation mode. The computation is unsuccessful if it ends before reading the last symbol from the input tape.

final state reached (finstate) - the computation ends whenever the last symbol is read as far as the automaton would not stop further. The computation is successful if the input tape is read and PCol automaton reaches the configuration from the set of the final states F.

The language accepted by the PCol automaton Π is defined as a set of the words for which there exist successful computation in particular mode and type of acceptance.

Definition 2. $L(\Pi, mod, acc) = \{w \in A^* | w$ *is accepted by the computation in the mode* mod *with type of acceptance* $acc\}$,

where $mod \in \{t, nt, tmax, ntmax, tmin, ntmin, init\}$ *and* $acc \in \{halt, lastsym, finstate\}$.

3 Robot control using the PCol automaton

Main advantage of using PCol automaton in the controlling robot behaviour is the parallel proceeding of the data done by very primitive computational units using very simple rules.

By conjunction modularity and PCol automaton we obtain a powerful tool to control robot behaviour. PCol automaton is parallel computation device. Collaterally working autonomous units sharing common environment provide fast computation device. Dividing agents into the modules allows us to compound agents

controlling single robot sensors and actuators. All the modules are controlled by the main controlling unit. Input tape gives us an opportunity to plan robot actions. Each input symbol represents a single instruction which has to be done by the robot, so the input string is the sequence of the actions which guides the robot to reach his goal; performing all the actions.

We construct a PCol automaton with four modules: *Control unit*, *Left actuator controller*, *Right actuator controller* and *Infra-red receptor*. Entire automaton is amended by the *input* and *output filter*. The input filter codes signals from the robots receptors and spread the coded signal into the environment. In the environment there is the coded signal used by the agents. The output filter decodes the signal from the environment which the actuator controllers sent into it. Decoded signal is forwarded to the robots actuators.

The control unit is the main module which controls the computation. It reads the sequence of the actions from the input tape. According to the type of the action read from the tape it asks the data from the sensors modules by sending particular objects into the environment. If the answer from the sensors allows to perform the action, the control unit sends the command to the actuator controllers to perform demanded action. After sending the command to the actuator controllers the control unit waits for the announcement of the successful or the unsuccessful performance of the demanded action from the actuator controllers. If the action was fulfilled then the control unit continues in reading the input tape and performs following action.

The infra-red receptors consume all the symbols released into the environment by the input filter. It releases actual information from the sensors on demand of the control unit. The infra-red receptors remove unused data from the environment.

The left and right actuator controllers wait for the activating signal from the control unit. After obtaining the activating signal the controllers try to provide demanded action by sending special objects - coded signal for the output filter into the environment. When the action is performed successfully the actuators send the announcement of the successful end of the action to the control unit, the announcement of the unsuccessful end of the action otherwise.

Let us introduce the formal definition of the mentioned PCol automaton: $\Pi = (A, e, V_E, (O_1, P_1), \ldots (O_7, P_7), \emptyset)$, where

$A = \{0_L, 0_R, 1_L, 1_R, e, F_F, \overline{F_F}, F_L, \overline{F_L}, F_R, \overline{F_R}, G_F, G_L, G_R, I_F, I_L, I_R, M_F, M_L, M_R, N_F, \overline{N_F}, N_L, \overline{N_L}, N_R, \overline{N_R}, R, R_T, W_F, W_L, W_R, W_T\}$,

$V_E = \{e\}$,

Control Unit:

$O_1 = \{ T, e\}$,

$P_1 = \{ < R_T \xrightarrow{T} M_F; e \to e >; < G_F M_F \to M_F M_F >; < M_F M_F \to ee >;$
$< e \leftrightarrow G_F / e \leftrightarrow R; M_F \to M_F >; < R M_F \to e W_T >;$
$< R_T \xrightarrow{T} M_L; e \to e >; < G_L M_L \to M_L M_L >; < M_L M_L \to ee >;$
$< e \leftrightarrow G_L / e \leftrightarrow R; M_L \to M_L >; < R M_L \to e W_T >;$
$< R_T \xrightarrow{T} M_R; e \to e >; < G_R M_R \to M_R M_R >; < M_R M_R \to ee >;$
$< e \leftrightarrow G_R / e \leftrightarrow R; M_R \to M_R >; < R M_R \to e W_T >;$
$< ee \to e W_T >; < W_T \to e; e \leftrightarrow R_T > \}$

$O_2 = \{ T, e\}$,

$P_2 = \{ < R_T \xrightarrow{T} M_F; e \to I_F >; < I_F \leftrightarrow e; M_F \to W_F >;$
$< W_F \to e; e \leftrightarrow F_F / e \leftrightarrow N_F >; < F_F \to G_F; e \to e >;$
$< G_F \leftrightarrow e; e \to W_T >; < R_T \xrightarrow{T} M_L; e \to I_L >;$
$< I_L \leftrightarrow e; M_L \to W_L >; < W_L \to e; e \leftrightarrow F_L / e \leftrightarrow N_L >;$
$< F_L \to G_L; e \to e >; < G_L \leftrightarrow e; e \to W_T >; < R_T \xrightarrow{T} M_R; e \to I_R >;$
$< I_R \leftrightarrow e; M_R \to W_R >; < W_R \to e; e \leftrightarrow F_R / e \leftrightarrow N_R >;$
$< F_R \to G_R; e \to e >; < G_R \leftrightarrow e; e \to W_T >; < R \leftrightarrow e; e \to W_T >;$
$< W_T \to e; e \leftrightarrow R_T > \}$

Infra-red module:

$O_3 = \{ e, e\}$,

$P_3 = \{ < e \leftrightarrow \overline{F_F}; e \to F_F >; < F_F \leftrightarrow I_F / F_F \to e; \overline{F_F} \to e >;$
$< e \leftrightarrow \overline{N_F}; e \to N_F >; < N_F \leftrightarrow I_F / N_F \to e; \overline{N_F} \to e >;$
$< I_F e \to ee; > \}$

$O_4 = \{ e, e\}$,

$P_4 = \{ < e \leftrightarrow \overline{F_L}; e \to F_L >;$
$< F_L \leftrightarrow I_L / F_L \to e; \overline{F_L} \to e >; < e \leftrightarrow \overline{N_L}; e \to N_L >;$
$< N_L \leftrightarrow I_L / N_L \to e; \overline{N_L} \to e >; < I_L e \to ee; > \}$

$O_5 = \{ e, e\}$,

$P_5 = \{ < e \leftrightarrow \overline{F_R}; e \to F_R >;$
$< F_R \leftrightarrow I_R / F_R \to e; \overline{F_R} \to e >; < e \leftrightarrow \overline{N_R}; e \to N_R >;$
$< N_R \leftrightarrow I_R / N_R \to e; \overline{N_R} \to e >; < I_R e \to ee; > \}$

Left Actuator controller:

$O_6 = \{ e, e\}$,

$P_6 = \{ < e \leftrightarrow M_F; e \to 1_L >; < M_F \to R_T; 1_L \leftrightarrow e >;$
$< e \leftrightarrow M_R; e \to 1_L >; < M_R \to R_T; 1_L \leftrightarrow e >;$
$< e \leftrightarrow M_L; e \to 0_L >; < M_L \to R_T; 0_L \leftrightarrow e >;$
$< R_T \leftrightarrow e; e \to e > \}$

Right Actuator controller:

$O_7 = \{ e, e\}$,

$P_7 = \{ < e \leftrightarrow M_F; e \to 1_R >; < M_F \to R_T; 1_R \leftrightarrow e >;$
$< e \leftrightarrow M_R; e \to 0_R >; < M_R \to R_T; 0_R \leftrightarrow e >;$
$< e \leftrightarrow M_L; e \to 1_R >; < M_L \to R_T; 1_R \leftrightarrow e >;$
$< R_T \leftrightarrow e; e \to e > \}$

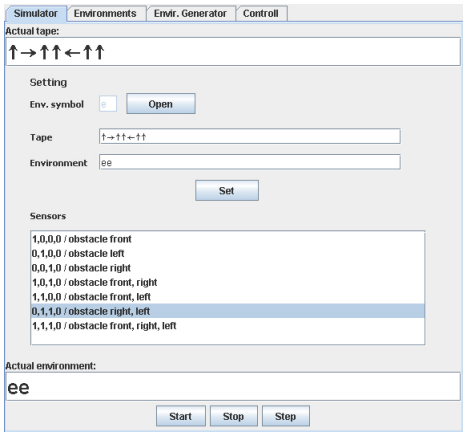


Fig. 1. Simulator

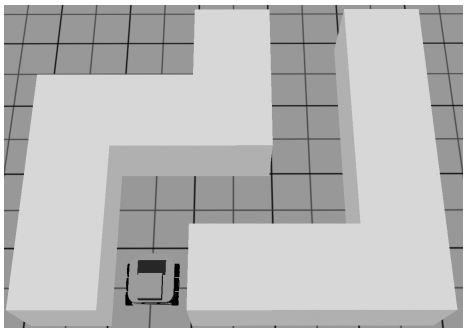


Fig. 2. Starting position

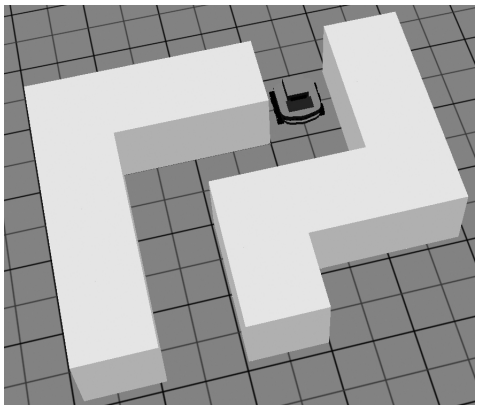


Fig. 3. Ending position

The robot driven by this very simple PCol automaton is able to follow the instruction on the tape safely without crashing into any obstacle. If the instruction cannot be proceeded, the robot stops. This solution is suitable for known robots environment. Following the instructions on the tape (picture 1) the robot can move from its starting position (picture 2) to the final destination (picture 3). If the environment is changed before or during the journey, the robot cannot reach the final place but it also will not crash.

4 Conclusion

We have shown the basic possibilities of controlling the robot using the PCol automaton and modular approach. With respect to the fact that P colonies are computationally complete device (see [2]) the further research will be dedicated to the more precise control and the possibilities of processing the information from other sensors, especially from the camera. Fulfilling more complex tasks and more autonomous behaviour (e.g. attempt go round the obstacle if it is not possible to go in demanded direction, skipping unrealizable tasks, etc.) is also direction of the further research.

By extending the robot by the acting modules like e.g. mechanic tongs the robot can fulfil more complicated tasks. To control such a device by the PCol automaton we just need to add a new control module and extend set of programs of the main control unit. Such an extension is thanks to the modularity very easy.

Remark 1.
This work was partially supported by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), by SGS/7/2011 and by project OPVK no. CZ.1.07/2.2.00/28.0014.

References

1. Cienciala, L., Ciencialová, L. Csuhaj-Varjú, Vazsil, G.: *PCol Automata: Recognizing Strings with P colonies.* Eight Brainstormung Week on Membrane Computing (In Martínez del Amor, M. A., Păun, G., Hurtado de Mendoza, I. P., Riscon-Núnez, A. (eds.)), Sevilla, 2010, pp. 65–76.
2. Cienciala, L., Ciencialová, L., Langer, M.: *Modularity in P Colonies with Checking Rules.* In: Revised Selected Papers 12 th International Conference CMC 2011 (Gheorge, M., Păun, Gh., Rozenber, G., Salomaa, A., Verlan, S. eds.), Springer, LNCS 7184, 2012, pp. 104-120.
3. Csuhaj-Varjú, E., Kelemen, J., Kelemenová, A., Păun, Gh., Vaszil, G.: *Cells in environment: P colonies*, Multiple-valued Logic and Soft Computing, 12, 3-4, 2006, pp. 201–215.
4. Csuhaj-Varjú, E., Margenstern, M., Vaszil, G.: *P Colonies with a bounded number of cells and programs.* Pre-Proceedings of the 7^{th} Workshop on Membrane Computing (H. J. Hoogeboom, Gh. Păun, G. Rozenberg, eds.), Leiden, The Netherlands, 2006, pp. 311–322.

5. Floreano, D. and Mattiussi, C.: *Bio-inspirated Artificial Inteligence: Theories, Methods, and Technologies*, MIT Press, 2008.
6. Kelemen, J., Kelemenová, A.: *On P colonies, a biochemically inspired model of computation.* Proc. of the 6^{th} International Symposium of Hungarian Researchers on Computational Intelligence, Budapest TECH, Hungary, 2005, pp. 40–56.
7. Kelemenová, A.: *P Colonies.* In: The Oxford Handbook of Membrane Computing eds. Gh.Paun, G. Rozengerg, A. Salomaa Oxford University Press, Oxford, 2009, 584–593
8. Weiss, G.: *Multiagent systems. A modern approach to distributed artificial intelligence*, MIT Press, Cambridge, Massachusetts, 1999
9. Wit, C. C., Bastin, G., Siciliano, B.: *Theory of Robot Control*, Springer-Verlag New York, 1996.

Turing Incompleteness of Asynchronous P Systems with Active Membranes

Alberto Leporati[1], Luca Manzoni[1,2], and Antonio E. Porreca[1]

[1] Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
{leporati,luca.manzoni,porreca}@disco.unimib.it
[2] I3S Research Lab.,
University Nice Sophia Antipolis,
CS 40121 – 06903 Sophia Antipolis CEDEX, France

Summary. We prove that asynchronous P systems with active membranes without division rules can be simulated by place/transition Petri nets, and hence are computationally weaker than Turing machines. This result holds even if the synchronisation mechanisms provided by electrical charges and membrane dissolution are exploited.

1 Introduction

P systems with active membranes [5] are parallel computation devices inspired by the structure and functioning of biological cells. A tree-like hierarchical structure of membranes divides the space into regions, where *multisets* of objects (representing chemical substances) are located. The system evolves by means of rules rewriting or moving objects, and possibly changing the membrane structure itself, by dissolving or dividing membranes.

Under the *maximally parallel* updating policy, whereby all components of the system that can evolve concurrently during a given computation step are required to do so, these devices are known to be computationally universal. Alternative updating policies have also been investigated. In particular, *asynchronous* P systems with active membranes [3], where any, not necessarily maximal, number of non-conflicting rules may be applied in each computation step, have been proved able to simulate partially blind register machines [4], computation devices equivalent under certain acceptance conditions to place/transition Petri nets and vector addition systems [6]. This simulation only requires object evolution (rewriting) rules and communication rules (moving objects between regions).

In an effort to further characterise the effect of asynchronicity on the computational power of P systems, we prove that asynchronous P systems can be simulated by place/transition Petri nets, and as such they are not computationally equivalent

to Turing machines: indeed, the reachability of configurations and the deadlock-freeness (i.e., the halting problem) of Petri nets are decidable [1]. This holds even when membrane dissolution, which provides an additional synchronisation mechanism (besides electrical charges) whereby all objects are released simultaneously from the dissolving membrane, is employed by the P system being simulated. Unfortunately, this result does not seem to immediately imply the equivalence with partially blind register machines, as the notion of acceptance for Petri nets employed here is by halting and not by placing a token into a "final" place [4].

The paper is organised as follows: in Section 2 we recall the relevant definitions; in Section 3 we prove that asynchronous P systems are computationally equivalent to *sequential* P systems, where a single rule is applied during each computation step; in Section 4 we show that dissolution rules in sequential P systems can be replaced by a form of generalised communication rule; finally, in Section 5 we show how P systems using generalised communication rules can be simulated by Petri nets, thus proving our main result. Section 6 contains our conclusions and open problems.

2 Definitions

We recall the definition of P systems with active membranes and its various operating modes.

Definition 1. *A* P system with active membranes *of initial degree* $d \geq 1$ *is a tuple* $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$*, where:*

- Γ *is an alphabet, i.e., a finite nonempty set of* objects*;*
- Λ *is a finite set of labels for the membranes;*
- μ *is a membrane structure (i.e., a rooted* unordered *tree) consisting of* d *membranes injectively labelled by elements of* Λ*;*
- $w_{h_1}, \ldots, w_{h_d}$*, with* $h_1, \ldots, h_d \in \Lambda$*, are strings over* Γ*, describing the initial multisets of objects located in the* d *regions of* μ*;*
- R *is a finite set of* rules*.*

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

The following four kinds of rules are employed in this paper.

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

- *Send-in communication rules*, of the form $a\,[\,]_h^\alpha \to [b]_h^\beta$
  They can be applied to a membrane labeled by h, having charge α and such that the external region contains an occurrence of the object a; the object a is sent into h becoming b and, simultaneously, the charge of h is changed to β.
- *Send-out communication rules*, of the form $[a]_h^\alpha \to [\,]_h^\beta\, b$
  They can be applied to a membrane labeled by h, having charge α and containing an occurrence of the object a; the object a is sent out from h to the outside region becoming b and, simultaneously, the charge of h is changed to β.
- *Dissolution rules*, of the form $[a]_h^\alpha \to b$
  They can be applied to a membrane labeled by h, having charge α and containing an occurrence of the object a; the membrane h is dissolved and its contents are released in the surrounding region unaltered, except that an occurrence of a becomes b.

The most general form of P systems with active membranes [5] also includes *membrane division rules*, which duplicate a membrane and its contents; however, division rules are not used in this paper.

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

- Each object and membrane can be subject to at most one rule per step, except for object evolution rules (inside each membrane several evolution rules having the same left-hand side, or the same evolution rule can be applied simultaneously; this includes the application of the same rule with multiplicity).
- When several conflicting rules can be applied at the same time, a nondeterministic choice is performed; this implies that, in general, multiple possible configurations can be reached after a computation step.
- In each computation step, all the chosen rules are applied simultaneously (in an atomic way). However, in order to clarify the operational semantics, each computation step is conventionally described as a sequence of micro-steps as follows. First, all evolution rules are applied inside the elementary membranes, followed by all communication and dissolution rules involving the membranes themselves; this process is then repeated to the membranes containing them, and so on towards the root (outermost membrane). In other words, the membranes evolve only after their internal configuration has been updated. For instance, before a membrane dissolution occurs, all chosen object evolution rules must be applied inside it; this way, the objects that are released outside during the dissolution are already the final ones.
- The outermost membrane cannot be dissolved, and any object sent out from it cannot re-enter the system again.

In the *maximally parallel* mode, the multiset of rules to be applied must be maximal (i.e., no further rule can be added without creating conflicts) during each step. In the *asynchronous* mode, any nonempty multiset of applicable rules can be

chosen. Finally, in the *sequential* mode, exactly one rule per computation step is applied. In the following, only the latter two modes will be considered.

A *halting computation* of the P system Π is a finite sequence of configurations $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_n)$, where $\mathcal{C}_0$ is the initial configuration, every $\mathcal{C}_{i+1}$ is reachable from $\mathcal{C}_i$ via a single computation step, and no rule can be applied in $\mathcal{C}_n$. A *non-halting* computation $\boldsymbol{\mathcal{C}} = (\mathcal{C}_i : i \in \mathbb{N})$ consists of infinitely many configurations, again starting from the initial one and generated by successive computation steps, where the applicable rules are never exhausted.

The other model of computation we will employ is Petri nets. In particular, with this term we denote place/transition Petri nets with weighted arcs, self-loops and places of unbounded capacity [2]. A Petri net N is a triple (P, T, F) where P is the set of *places*, T the set of *transitions* (disjoint from P) and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. The arcs are weighted by a function $w\colon F \to (\mathbb{N} - \{0\})$. A *marking* (i.e., a configuration) is a function $M\colon P \to \mathbb{N}$. Given two markings M, M' of N and a transition $t \in T$ we say that M' is reachable from M via the firing of t, in symbols $M \to_t M'$, if and only if:

- for all places $p \in P$, if $(p,t) \in F$ and $(t,p) \notin F$ then $M(p) \geq w(p,t)$ and $M'(p) = M(p) - w(p,t)$;
- for all $p \in P$, if $(t,p) \in F$ and $(p,t) \notin F$ then $M'(p) = M(p) + w(t,p)$;
- for all $p \in P$, if both $(p,t) \in F$ and $(t,p) \in F$ then $M(p) \geq w(p,t)$ and $M'(p) = M(p) - w(p,t) + w(t,p)$.

Petri nets are nondeterministic devices, hence multiple markings may be reachable from a given configuration. We call *halting computation* a sequence of markings $(M_0, \ldots M_n)$ where $M_0 \to_{t_1} M_1 \to_{t_2} \cdots \to_{t_n} M_n$ for some $t_1, \ldots, t_n$, and no transition may fire in M_n. Several problems related to the reachability of markings and halting configurations (or *deadlocks*) are decidable [1].

3 Asynchronicity and Sequentiality

In this section we show how it is possible to find, for every asynchronous P system, a *sequential* P system that is equivalent to the original one in the sense that they both halt on the same inputs and produce the same outputs.

The main idea is that each asynchronous step where more than one rule is applied can be substituted by a sequence of asynchronous steps where the rules are reordered and applied one at a time.

Proposition 1. *Let Π be a P system with active membranes using object evolution, communication, and dissolution rules. Then, the asynchronous and the sequential updating policies of Π are equivalent in the following sense: for each asynchronous (resp., sequential) computation step $\mathcal{C} \to \mathcal{D}$ we have a series of sequential (resp., asynchronous) steps $\mathcal{C} = \mathcal{C}_0 \to \cdots \to \mathcal{C}_n = \mathcal{D}$ for some $n \in \mathbb{N}$.*

Proof. Every asynchronous computation step $\mathcal{C} \to \mathcal{D}$ consists in the application of a finite multiset of rules $\{e_1, \ldots, e_p, c_1, \ldots, c_q, d_1, \ldots, d_r\}$, where $e_1, \ldots, e_p$ are object evolution rules, $c_1, \ldots, c_q$ are communication rules (either send-in or send-out), and $d_1, \ldots, d_r$ are dissolution rules.

Since evolution rules do not change any charge nor the membrane structure itself, the computation step $\mathcal{C} \to \mathcal{D}$ can be decomposed into two asynchronous computation steps $\mathcal{C} \to \mathcal{E} \to \mathcal{D}$, where the step $\mathcal{C} \to \mathcal{E}$ consists in the application of the evolution rules $\{e_1, \ldots, e_p\}$, and the step $\mathcal{E} \to \mathcal{D}$ in the application of the remaining rules $\{c_1, \ldots, c_q, d_1, \ldots, d_r\}$. Notice that in $\mathcal{E}$ there still exist enough objects to apply these communication and dissolution rules, since by hypothesis $\mathcal{C} \to \mathcal{D}$ is a valid computation step.

Furthermore, notice how there is no conflict between object evolution rules (once they have been assigned to the objects they transform). Therefore, the application of the rules $\{e_1, \ldots, e_p\}$ can be implemented as a series of sequential steps $\mathcal{C} = \mathcal{C}_0 \to \cdots \to \mathcal{C}_p = \mathcal{E}$.

Each membrane can be subject to at most a single rule of communication or dissolution type in the computation step $\mathcal{C} \to \mathcal{D}$; hence, applying one of these rules does not interfere with any other. Thus, these rules can also be serialised into sequential computation steps $\mathcal{E} \to \mathcal{C}_{p+1} \to \cdots \to \mathcal{C}_{p+q+r} = \mathcal{D}$. Once again, all rules remain applicable since they were in the original computation step.

By letting $n = p + q + r$, the first half of the proposition follows. The second part is due to the fact that every sequential computation step is already an asynchronous computation step. □

4 Generalised Communication Rules

In this section we define a variant of P systems with active membranes that we call *generalised communication P systems*, where every evolution, communication, and dissolution rule is replaced by a generalised communication rule, in which an object can at the same time be rewritten and move between membranes while changing their charges. This requires the introduction of an extra membrane charge. Generalised communication rules are introduced in order to simplify the simulation by means of Petri nets, as shown in the next section.

To maintain uniformity in the structure of the rules, we define the external environment as a membrane having label 0 containing all the other membranes and having constant charge.

Definition 2 (Generalised communication rules). *A* generalised communication rule *is a rule of the form*

$$[\cdots[a[\cdots[\,]_{h_n}^{\alpha_n}\cdots]_{h_{i+1}}^{\alpha_{i+1}}]_{h_i}^{\alpha_i}\cdots]_{h_1}^{\alpha_1} \to [\cdots[w[\cdots[\,]_{h_n}^{\beta_n}\cdots]_{h_{j+1}}^{\beta_{j+1}}]_{h_j}^{\beta_j}\cdots]_{h_1}^{\beta_1}$$

On the left-hand side of the rule we have a path in μ (a sequence of nested membranes) consisting of membranes $h_1, \ldots, h_n$ with charges $\alpha_1, \ldots, \alpha_n$, and a single

object a contained in membrane h_i (for some $1 \leq i \leq n$). On the right-hand side the same path appears, with charges $\beta_1, \ldots, \beta_n$, and a multiset w appears in membrane h_j (for some $1 \leq j \leq n$). The rule can be applied when membranes $h_1, \ldots, h_n$ have charges $\alpha_1, \ldots, \alpha_n$ and a copy of a exists inside h_i; the rule removes that copy of a from h_i, adds w to the region h_j, and changes the charges to $\beta_1, \ldots, \beta_n$.

As a special case, $h_1 = 0$ denotes the external environment of the P system; in that case, the charge of h_1 can never be changed.

Definition 3 (Generalised communication P systems). *A generalised communication P system of degree $d \geq 1$ is a structure*

$$\Pi = (\Gamma, \Lambda, \Psi, \mu, w_{h_1}, \ldots, w_{h_d}, R)$$

where the elements Γ, Λ, μ, $w_{h_1}, \ldots, w_{h_d}$ are the same as in standard P systems with active membranes, Ψ is a finite, nonempty set of electrical charges (replacing the standard set of charges $\{+, 0, -\}$), and R consists only of generalised communication rules.

We can now show that generalised communication P systems are equivalent to standard P systems with active membranes (without division rules) when operating under the sequential semantics.

Proposition 2. *Let $\Pi = (\Gamma, \Lambda, \mu, w_{h_1}, \ldots, w_{h_d}, R)$ be a P system with active membranes working in sequential mode and using object evolution, communication, and dissolution rules. Then, there exists a generalised communication P system with active membranes $\Pi' = \big(\Gamma, \Lambda, \{+, 0, -, \bullet\}, \mu, w_{h_1}, \ldots, w_{h_d}, R'\big)$ working in sequential mode, having the same initial configuration $\mathcal{C}_0$ as Π, and such that*

(i) If $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_m)$ is a halting computation of Π, then there exists a halting computation $\boldsymbol{\mathcal{D}} = (\mathcal{C}_0, \mathcal{D}_1, \ldots, \mathcal{D}_n)$ of Π' such that each membrane appearing in both $\mathcal{C}_m$ and $\mathcal{D}_n$ contains the same objects and has the same charge in both configurations; if a membrane has dissolved during the computation $\boldsymbol{\mathcal{C}}$, then the corresponding membrane in $\mathcal{D}_n$ is empty and with charge $\bullet$.

(ii) If $\boldsymbol{\mathcal{D}} = (\mathcal{C}_0, \mathcal{D}_1, \ldots, \mathcal{D}_n)$ is a halting computation of Π', then there exists a halting computation $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \mathcal{C}_1, \ldots, \mathcal{C}_m)$ of Π such that each membrane appearing in both $\mathcal{C}_m$ and $\mathcal{D}_n$ contains the same objects and has the same charge in both configurations; if a membrane has charge $\bullet$ in $\mathcal{D}_n$, then the corresponding membrane dissolves during the computation $\boldsymbol{\mathcal{C}}$.

(iii) Π admits a non-halting computation $(\mathcal{C}_0, \mathcal{C}_1, \ldots)$ if and only if Π' admits a non-halting computation $(\mathcal{C}_0, \mathcal{D}_1, \ldots)$.

Proof. The main idea is to replace every dissolution rule in R with a generalised communication rule setting the charge of the membrane h that would be dissolved with a new charge $\bullet$. After setting the charge of h, the objects inside it must be moved to the nearest surrounding membrane having charge different from $\bullet$, in order to ensure that membranes with the same label contain the same objects in

Π and Π'. Send-in communication rules must also be adapted, since the object to be brought in might be not immediately outside the membrane, as a number of membranes with charge $\bullet$ might be interposed.

Let $[a]^{\alpha}_{h_1} \to b$ be a dissolution rule in R. Then, R' contains the following generalised communication rules:

$$[[a]^{\alpha}_{h_1}]^{\beta}_{h_2} \to [[\,]^{\bullet}_{h_1} b]^{\beta}_{h_2} \qquad \text{for } \beta \in \{+, -, 0, \bullet\}, \tag{1}$$

where h_2 is the parent membrane of h_1 in μ. The remaining objects are sent out from h_1 by means of the following rules:

$$[[a]^{\bullet}_{h_1}]^{\beta}_{h_2} \to [[\,]^{\bullet}_{h_1} a]^{\beta}_{h_2} \qquad \text{for } a \in \Gamma,\ \beta \in \{+, 0, -, \bullet\}. \tag{2}$$

Notice that, if $\beta = \bullet$, then membrane h_2 has been dissolved during a previous computation step; this means that there exists another rule of type (2) sending all the objects out from h_2. Hence, the objects never remain stuck in a membrane with charge $\bullet$, and eventually reach a membrane having a different charge.

An object evolution rule $[a \to w]^{\alpha}_{h}$ is simulated by the following generalised communication rule:

$$[a]^{\alpha}_{h} \to [w]^{\alpha}_{h}. \tag{3}$$

A send-out communication rule $[a]^{\alpha}_{h_1} \to [\,]^{\beta}_{h_1} b$ is replaced by the following rules:

$$[[a]^{\alpha}_{h_1}]^{\gamma}_{h_2} \to [[\,]^{\beta}_{h_1} b]^{\gamma}_{h_2} \qquad \text{for } \gamma \in \{+, 0, -, \bullet\}. \tag{4}$$

where h_2 is the parent membrane of h_1 in μ. As mentioned before, if $\gamma = \bullet$, then a rule of type (2) will move b out of h_2.

Finally, a send-in communication rule $a\ [\,]^{\alpha}_{h_1} \to [b]^{\beta}_{h_1}$ is simulated as follows. Let $(h_n, h_{n-1}, \ldots, h_2, h_1)$ be a sequence of nested membranes surrounding h_1, i.e., a descending path in the membrane tree μ. For every such sequence, we add the following rules to R':

$$[a\ [\cdots [[\,]^{\alpha}_{h_1}]^{\bullet}_{h_2} \cdots]^{\bullet}_{h_{n-1}}]^{\gamma}_{h_n} \to [[\cdots [[b]^{\beta}_{h_1}]^{\bullet}_{h_2} \cdots]^{\bullet}_{h_{n-1}}]^{\gamma}_{h_n} \qquad \text{for } \gamma \in \{+, 0, -\}. \tag{5}$$

This rule moves the object a into h_1 from the nearest membrane outside h_1 having charge in $\{+, 0, -\}$, ignoring any interposed membrane with charge $\bullet$ (corresponding to a dissolved membrane in Π). Observe that the number of descending paths leading to h_1 is bounded above by the depth of μ.

Notice how every rule of R' is exactly of one type among (1)–(5); in particular, given a rule in R' of type (1), (3), (4), or (5), it always possible to reconstruct the original rule in R.

Each computation step of Π consisting in the application of an evolution or send-in communication rule is simulated by a single computation step of Π' by means of a rule of type (3) or (5) respectively.

The dissolution of a membrane h_1 in Π requires a variable number of steps of Π': first, a rule of type (1) is applied, then each object located inside h_1 is

sent out to the nearest membrane surrounding h_1 having a charge different from $\bullet$ by using rules of type (2). The exact number of steps depends on the number of objects located inside h_1 and the number of membranes with charge $\bullet$ they have to traverse. The reasoning is analogous for send-out communication rules, simulated by means of rules of type (4) and (2).

Part (i) of the proposition follows from the semantics of generalised communication rules.

Now let $\boldsymbol{\mathcal{D}} = (\mathcal{D}_0 = \mathcal{C}_0, \mathcal{D}_1, \ldots, \mathcal{D}_n)$ be a halting computation of Π'. Then there exists a sequence of rules $\mathbf{r} = (r_1, \ldots, r_n)$ in R' such that

$$\mathcal{D}_0 \to_{r_1} \mathcal{D}_1 \to_{r_2} \cdots \to_{r_{n-1}} \mathcal{D}_{n-1} \to_{r_n} \mathcal{D}_n$$

where the notation $\mathcal{X} \to_r \mathcal{Y}$ indicates that configuration $\mathcal{Y}$ is reached from $\mathcal{X}$ by applying the rule r. Let $f\colon \mathbb{N} \to \mathbb{N}$ be defined as

$$f(t) = \big|\{r_i : 1 \leq i \leq t \text{ and } r_i \text{ is not of type (2)}\}\big|.$$

We claim that there exists a sequence of rules $\mathbf{s} = (s_1, \ldots, s_m)$ such that the computation $\boldsymbol{\mathcal{C}} = (\mathcal{C}_0, \ldots, \mathcal{C}_m)$ of Π generated by applying the rules of $\mathbf{s}$, i.e.,

$$\mathcal{C}_0 \to_{s_1} \mathcal{C}_1 \to_{s_2} \cdots \to_{s_{m-1}} \mathcal{C}_{m-1} \to_{s_m} \mathcal{C}_m$$

has the following property $P(t)$ for each $t \in \{0, \ldots, n\}$:

> For all $h \in \Lambda$ and $a \in \Gamma$, if h has a charge among $\{+, 0, -\}$ in configuration $\mathcal{D}_t$ of Π', then the number of copies of a contained in the membrane substructure rooted in h is equal in $\mathcal{D}_t$ and $\mathcal{C}_{f(t)}$, and h has the same charge in both configurations. If h has charge $\bullet$ in $\mathcal{D}_t$, then it does not appear in $\mathcal{C}_{f(t)}$ (having dissolved before).

We prove this property by induction on t. The case $t = 0$ clearly holds, since Π and Π' have the same initial configuration: $\mathcal{C}_{f(0)} = \mathcal{C}_0 = \mathcal{D}_0$, as $f(0) = |\varnothing|$.

Now suppose $P(t)$ holds for some $t < n$. If r_{t+1} is a rule of type (2) then an object is sent out from a membrane with charge $\bullet$; therefore, for each object $a \in \Gamma$, the number of copies of a does not change from $\mathcal{D}_t$ to $\mathcal{D}_{t+1}$ in any subtree rooted in a membrane having charge in $\{+, 0, -\}$; furthermore, no membrane changes its charge. Since r_{t+1} is of type (2), we have $f(t+1) = f(t)$ hence $\mathcal{C}_{f(t+1)} = \mathcal{C}_{f(t)}$, and property $P(t+1)$ holds.

On the other hand, if r_{t+1} is not of type (2), then $f(t+1) = f(t) + 1$ by definition. Let $s_{f(t)+1} = s_{f(t+1)}$ be the rule corresponding to the generalised communication rule r_{t+1} as described above (an object evolution rule if r_{t+1} is of type (3), a dissolution rule if r_{t+1} is of type (1), and so on). Observe that if r_{t+1} is applicable in $\mathcal{D}_t$, then $s_{f(t)+1}$ is applicable in $\mathcal{C}_{f(t)}$ by induction hypothesis:

- identically labelled membranes have the same charge in $\mathcal{D}_t$ and $\mathcal{C}_{f(t)}$;
- if r_{t+1} is of type (1), (3), or (4) and uses an object a located inside a membrane h having charge $+$, 0, or $-$ in $\mathcal{D}_t$, then a copy of a also appears in membrane h in $\mathcal{C}_{f(t)}$ for the membrane substructure property;

- if r_{t+1} is of type (5) and uses an object a located inside a membrane h having charge $\bullet$ in $\mathcal{D}_t$, then the object a appears in $\mathcal{C}_{f(t)}$ inside the membrane having the same label as the nearest membrane outside h in $\mathcal{D}_t$ with charge different from $\bullet$.

The configuration $\mathcal{C}_{f(t)+1}$ such that $\mathcal{C}_{f(t)} \rightarrow_{s_{f(t)+1}} \mathcal{C}_{f(t)+1}$, due to the semantics of the corresponding rules applied by Π and Π', is such that the property $P(t+1)$ holds: objects are moved to identically labelled membranes, charges in $\{+, 0, -\}$ are changed in both systems in the same way, and membranes that are set to $\bullet$ by Π' are dissolved by Π.

In particular, $P(n)$ holds: configurations $\mathcal{D}_n$ and $\mathcal{C}_{f(n)}$ have the following properties: membrane substructures rooted in identically labelled membranes contain the same multisets, identically labelled membranes have the same charge if it is in $\{+, 0, -\}$, and membranes having charge $\bullet$ in $\mathcal{D}_n$ do not appear in $\mathcal{C}_{f(n)}$. Notice that $\mathcal{C}_{f(n)}$ is a halting configuration, since otherwise any rule applicable from it could be simulated from $\mathcal{D}_n$ as in statement (i). Furthermore, membranes having charge $\bullet$ in $\mathcal{D}_n$ are empty, otherwise further rules of type (2) could be applied, contradicting the hypothesis that $\mathcal{D}_n$ is a halting configuration. As a consequence, not just the membrane substructures, but the individual membranes contain the same multisets in $\mathcal{D}_n$ and $\mathcal{C}_{f(n)}$, and statement (ii) follows.

Finally, let us consider a non-halting computations of Π. Each time a computation of Π can be extended by one step by applying a rule, that rule can be simulated by Π' using the same argument employed to prove statement (i), thus yielding a non-halting computation of Π'. Vice versa, in a non-halting computation of Π' it is never the case that infinitely many rules of type (2) are applied sequentially, as only finitely many objects exist at any given time, and eventually they reach a membrane having charge different from $\bullet$. As soon as a rule of type (1), (3), (4), or (5) is applied, the corresponding rule can also be applied by Π, thus yielding a non-halting computation. □

5 Simulation with Petri Nets

The generalised communication P systems we introduced in the last section can be straightforwardly simulated by Petri nets.

Proposition 3. *Let $\Pi = (\Gamma, \Lambda, \Psi, \mu, w_{h_1}, \ldots, w_{h_d}, R)$ be a generalised communication P system working in sequential mode. Then, there exists a Petri net N, having $\big((\Lambda \cup \{0\}) \times \Gamma\big) \cup (\Lambda \times \Psi)$ among its places, such that $\mathcal{C} \rightarrow \mathcal{C}'$ is a computation step of Π if and only if $M \rightarrow M'$ is a computation step of N, where*

- *$M(h, a)$ is the number of instances of a in membrane h in $\mathcal{C}$;*
- *$M(0, a)$ is the number of instances of a in the environment in $\mathcal{C}$;*
- *$M(h, \alpha) = 1$ if h has charge α in $\mathcal{C}$, and $M(h, \alpha) = 0$ otherwise;*
- *$M'(h, a)$ is the number of instances of a in membrane h in $\mathcal{C}'$;*
- *$M'(0, a)$ is the number of instances of a in the environment in $\mathcal{C}'$;*

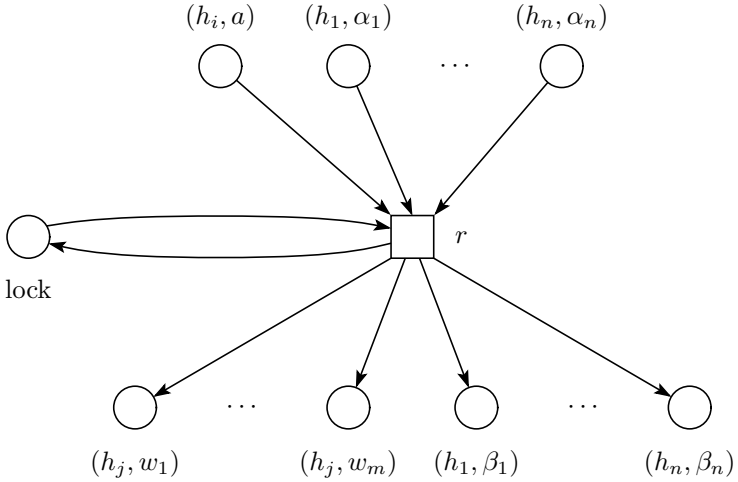
- $M'(h, \alpha) = 1$ *if* h *has charge* α *in* $\mathcal{C}'$*, and* $M'(h, \alpha) = 0$ *otherwise.*

Proof. The set of places of N is defined as $\big((\Lambda \cup \{0\}) \times \Gamma\big) \cup (\Lambda \times \Psi) \cup \{\text{lock}\}$, where lock is a place always containing a single token that is employed in order to ensure the firing of at most one transition per step.

For every generalised communication rule

$$r = [\cdots [a[\cdots [\,]_{h_n}^{\alpha_n} \cdots]_{h_{i+1}}^{\alpha_{i+1}}]_{h_i}^{\alpha_i} \cdots]_{h_1}^{\alpha_1} \to [\cdots [w[\cdots [\,]_{h_n}^{\beta_n} \cdots]_{h_{j+1}}^{\beta_{j+1}}]_{h_j}^{\beta_j} \cdots]_{h_1}^{\beta_1}$$

with $w = w_1 w_2 \cdots w_m$, the net has a transition defined as follows:



Notice that the output places need not be distinct, as the multiset w may contain multiple occurrences of the same symbol; in that case, a weighted arc is used. The output places need not be distinct from the input places either, as w may contain a, or the charge of a membrane may not change. In that case, the net contains a loop.

The initial marking M_0 of N is given by

$$M_0(h, a) = |w_h|_a \qquad M_0(0, a) = 0$$
$$M_0(h, 0) = 1 \qquad M_0(h, \alpha) = 0 \qquad M_0(\text{lock}) = 1$$

for all $h \in \Lambda$, $a \in \Gamma$, and $\alpha \in \Psi - \{0\}$, where $|w_h|_a$ is the multiplicity of a in w_h.

Notice that a transition r in N is enabled exactly when the corresponding rule $r \in R$ is applicable, producing a transition $M \to_r M'$ corresponding to a computation step $\mathcal{C} \to_r \mathcal{C}'$ of Π as required. □

By combining Propositions 1, 2, and 3, we can finally prove our main theorem.

Theorem 1. *For every asynchronous P system with active membranes* Π *using evolution, communication, and dissolution rules there exists a Petri net* N *such that every halting configuration of* Π *corresponds to a halting configuration of* N

and vice versa (under the encoding of Proposition 3, the charge • denoting dissolved membranes), and every non-halting computation of Π corresponds to a non-halting computation of N and vice versa. □

Notice that, given the strict correspondence of computations and their halting configurations (if any) between the two devices, this result holds both for P systems computing functions over multisets (or their Parikh vectors) and those recognising or generating families of multisets (or their Parikh vectors), since the only difference between these various computing modes is the initial configuration and the acceptance condition; these are translated directly into the simulating Petri net.

6 Conclusions

We have proved that asynchronous P systems with active membranes (without division rules) can be simulated by place/transition Petri nets, and hence are not computationally universal. In order to achieve this result, we proved that the asynchronous and the sequential parallelism policies are equivalent, and that membrane dissolution can be replaced by a generalised form of communication, together with an extra membrane charge.

However, the conjectured equivalence of asynchronous P systems and Petri nets does not seem to follow immediately from our result and the previous simulation of partially blind register machines by means of P systems [3]. Indeed, an explicit signalling (putting a token into a specified place) instead of accepting by halting seems to be required in order to simulate Petri nets with partially blind register machines [4]. Directly simulating Petri nets with asynchronous P systems is also nontrivial, since transitions provide a stronger synchronisation mechanism than the limited context-sensitivity of the rules of a P system with active membranes. This equivalence is thus left as an open problem.

We also conjecture that asynchronous P systems with active membrane remain non-universal even when membrane division rules are allowed. However, if this is the case, a different proof technique than that of Section 3 is required, as our current simulation by Petri nets does not support the creation of new membranes.

Acknowledgements

We would like to thank Luca Bernardinello for his advice on the theory of Petri nets. This research was partially funded by Lombardy Region under project NEDD.

References

1. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. Theoretical Computer Science 147, 117–136 (1995)

2. Desel, J., Reisig, W.: Place/transition Petri nets. In: Reisig, W., Rozenberg, G. (eds.) Lectures on Petri nets I: Basic models, Advances in Petri Nets, vol. 1491, pp. 122–173. Springer (1998)
3. Frisco, P., Govan, G., Leporati, A.: Asynchronous P systems with active membranes. Theoretical Computer Science 429, 74–86 (2012)
4. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science 7, 311–324 (1978)
5. Păun, Gh.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics 6(1), 75–90 (2001)
6. Peterson, J.L.: Petri net theory and the modeling of systems. Prentice-Hall (1981)

Improving Universality Results on Parallel Enzymatic Numerical P Systems

Alberto Leporati, Antonio E. Porreca, Claudio Zandron, Giancarlo Mauri

Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca
Viale Sarca 336/14, 20126 Milano, Italy
E-mail: {leporati,porreca,zandron,mauri}@disco.unimib.it

Summary. We improve previously known universality results on enzymatic numerical P systems (EN P systems, for short) working in all-parallel and one-parallel modes. By using a flattening technique, we first show that any EN P system working in one of these modes can be simulated by an equivalent one-membrane EN P system working in the same mode. Then we show that linear production functions, each depending upon at most one variable, suffice to reach universality for both computing modes. As a byproduct, we propose some small deterministic universal enzymatic numerical P systems.

1 Introduction

Numerical P systems have been introduced in [10] as a model of membrane systems inspired both from the structure of living cells and from economics. Each region of a numerical P system contains some numerical variables, that evolve from initial values by means of *programs*. Each program consists of a *production function* and a *repartition protocol*; the production function computes an output value from the values of some variables occuring in the same region in which the function is located, while the repartition protocol distributes this output value among the variables in the same region as well as in the neighbouring (parent and children) ones.

In [10], and also in Chapter 23.6 of [11], some results concerning the computational power of numerical P systems are reported. In particular, it is proved that nondeterministic numerical P systems with polynomial production functions characterize the recursively enumerable sets of natural numbers, while deterministic numerical P systems, with polynomial production functions having non-negative coefficients, compute strictly more than semilinear sets of natural numbers.

Enzymatic Numerical P systems (EN P systems, for short) have been introduced in [13] as an extension of numerical P systems in which some variables, named the *enzymes*, control the application of the rules, similarly to what happens in P systems with promoters and inhibitors [1]. Although in [10] it is claimed

that numerical P systems have been inspired by economic and business processes, the most promising application of their enzymatic version seems to be the simulation of control mechanisms of mobile and autonomous robots [12, 2, 14, 15].

In [17, 16] some results concerning the computational power of enzymatic P systems are reported. In particular, in [17] it is shown that EN P systems with 7 membranes and polynomial production functions of degree 5 involving at most 5 variables, working in the *sequential* mode (at each step, only one of the active programs is applied in each membrane) are universal. The computational power of EN P systems working in the so called *one-parallel* mode — programs are applied in parallel in each membrane, but each variable can appear only in *one* of the production functions — is also investigated, showing universality of these systems with an unlimited number of membranes and *linear* production functions (that is, polynomial functions of degree 1), each involving at most 2 variables. Finally, the universality of (deterministic) EN P systems working in the *all-parallel mode* — in each membrane all programs which can be applied are applied, possibly using the same variable in many production functions — having 254 membranes and polynomial production functions of degree 2 involving at most 253 variables, is established. A considerable improvement of the last result has subsequently been presented in [16], where it is proved that 4 membranes and linear production functions involving at most 6 variables suffice to obtain universal deterministic EN P systems working in the all-parallel mode.

In this paper we continue the study of the computational power of enzymatic numerical P systems. In particular we first show that, given any EN P system Π working either in the one-parallel or in the all-parallel mode, it is possible to build an equivalent EN P system Π' whose structure consists of a single membrane. This *flattening* technique already improves some of the above mentioned results, reducing to 1 the number of membranes required by all-parallel or one-parallel EN P systems to reach universality — albeit, despite this transformation, one-parallel EN P systems still require an *unbounded number of variables*. Then, we prove that for EN P systems working either in the all-parallel or in the one-parallel mode one membrane and linear production functions — each involving at most 1 variable — suffice to reach universality. These results are all obtained by simulating deterministic and/or nondeterministic register machines; by considering a small deterministic universal register machine described in [6], we obtain as byproducts some small deterministic universal EN P systems, working in the all-parallel mode.

A point to be considered is that the output of our EN P systems is defined as the value of some specified variables in a *final configuration*, that is, a configuration which is not changed by further applying programs. This allows us to simplify some of our constructions, but it is a bit different from the way EN P systems produce their output in most existing papers, where some specified output variables are considered, and the output of the system is the set of all values assumed by these variables during the entire computation. However, we prove that each of our EN P systems can be easily modified in order to produce its output according to the latter mode.

The rest of the paper is organized as follows. In section 2 we recall the definitions of EN P systems and register machines, along with the terms, tools and notation that will be used in the following. In section 3 we first show that any EN P system working either in the all-parallel or in the one-parallel mode can be "flattened" to one membrane, and then we prove our universality results on one-membrane EN P systems working in all-parallel or in one-parallel modes. In section 4 we show that the EN P systems used to obtain these results can be modified in order to produce their output into separate variables, as it is usually done in the literature. The conclusions and some directions for further work are given in section 5.

2 Definitions and Mathematical Preliminaries

We denote by $\mathbb{N}$ the set of non-negative integers. An *alphabet* A is a finite non-empty set of abstract *symbols*. Given A, the free monoid generated by A under the operation of concatenation is denoted by A^*; the *empty string* is denoted by λ, and $A^* - \{\lambda\}$ is denoted by A^+. By $|w|$ we denote the length of the word w over A. If $A = \{a_1, \ldots, a_n\}$, then the number of occurrences of symbol a_i in w is denoted by $|w|_{a_i}$; the *Parikh vector* associated with w with respect to $a_1, \ldots, a_n$ is $(|w|_{a_1}, \ldots, |w|_{a_n})$. The *Parikh image* of a language L over $\{a_1, \ldots, a_n\}$ is the set of all Parikh vectors of strings in L. For a family of languages **FL**, the family of Parikh images of languages in **FL** is denoted by **PsFL**. The family of recursively enumerable languages is denoted by **RE**; the family of all recursively enumerable sets of k-dimensional vectors of non-negative integers can thus be denoted by **Ps**(k)**RE**. Since numbers can be seen as one-dimensional vectors, we can replace **Ps**(1) by $\mathbb{N}$ in the notation, thus obtaining $\mathbb{N}$**RE**.

2.1 Enzymatic Numerical P Systems

An *enzymatic numerical P system* (EN P system, for short) is a construct of the form:

$$\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$$

where $m \geq 1$ is the degree of the system (the number of membranes), H is an alphabet of labels, μ is a tree-like membrane structure with m membranes injectively labeled with elements of H, Var_i and Pr_i are respectively the set of variables and the set of programs that reside in region i, and $Var_i(0)$ is the vector of initial values for the variables of Var_i. All sets Var_i and Pr_i are finite. In the original definition of EN P systems [13] the values assumed by the variables may be real, rational or integer numbers; in what follows we will allow instead only integer numbers. The variables from Var_i are written in the form $x_{j,i}$, for j running from 1 to $|Var_i|$, the cardinality of Var_i; the value assumed by $x_{j,i}$ at time $t \in \mathbb{N}$ is

denoted by $x_{j,i}(t)$. Similarly, the programs from Pr_i are written in the form $P_{l,i}$, for l running from 1 to $|Pr_i|$.

The *programs* allow the system to evolve the values of variables during computations. Each program is composed of two parts: a *production function* and a *repartition protocol*. The former can be any function using variables from the region that contains the program. Usually only polynomial functions are considered, since these are sufficient to reach the computational power of Turing machines, as proved in [17]. Using the production function, the system computes a *production value*, from the values of its variables at that time. This value is distributed to variables from the region where the program resides, and to variables in its upper (parent) and lower (children) compartments, as specified by the repartition protocol. Formally, for a given region i, let $v_1, \ldots, v_{n_i}$ be all these variables; let $x_{1,i}, \ldots, x_{k_i,i}$ be some variables from Var_i, let $F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})$ be the production function of a given program $P_{l,i} \in Pr_i$, and let $c_{l,1}, \ldots, c_{l,n_i}$ be natural numbers. The program $P_{l,i}$ is written in the following form:

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}) \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i} \tag{1}$$

where the arrow separates the production function from the repartition protocol. Let $C_{l,i} = \sum_{s=1}^{n_i} c_{l,s}$ be the sum of all the coefficients that occur in the repartition protocol. If the system applies program $P_{l,i}$ at time $t \geq 0$, it computes the value

$$q = \frac{F_{l,i}(x_{1,i}(t), \ldots, x_{k_i,i}(t))}{C_{l,i}}$$

that represents the "unitary portion" to be distributed to variables $v_1, \ldots, v_{n_i}$ proportionally with coefficients $c_{l,1}, \ldots, c_{l,n_i}$. So each of the variables v_s, for $1 \leq s \leq n_i$, will receive the amount $q \cdot c_{l,s}$. An important observation is that variables $x_{1,i}, \ldots, x_{k_i,i}$ involved in the production function are reset to zero after computing the production value, while the other variables from Var_i retain their value. The quantities assigned to each variable from the repartition protocol are added to the current value of these variables, starting with 0 for the variables which were reset by a production function. As pointed out in [17], a delicate problem concerns the issue whether the production value is divisible by the total sum of coefficients $C_{l,i}$. As it is done in [17], in this paper we assume that this is the case, and we deal only with such systems; see [10] for other possible approaches.

Besides programs (1), EN P systems may also have programs of the form

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})|_{e_{j,i}} \to c_{l,1}|v_1 + c_{l,2}|v_2 + \cdots + c_{l,n_i}|v_{n_i}$$

where $e_{j,i}$ is a variable from Var_i different from $x_{1,i}, \ldots, x_{k_i,i}$ and from $v_1, \ldots, v_{n_i}$. Such a program can be applied at time t only if $e_{j,i}(t) > \min(x_{1,i}(t), \ldots, x_{k_i,i}(t))$. Stated otherwise, variable $e_{j,i}$ operates like an *enzyme*, that enables the execution of the program, but — like it happens also with catalysts — it is neither consumed nor modified by the execution of the program. However, in EN P systems enzymes can evolve by means of other programs, that is, enzymes can receive "contributions" from other programs and regions.

A *configuration* of Π at time $t \in \mathbb{N}$ is given by the values of all the variables of Π at that time; in a compact notation, we can write it as the sequence $(Var_1(t), \ldots, Var_m(t))$, where m is the degree of Π. The *initial configuration* can thus be described as the sequence $(Var_1(0), \ldots, Var_m(0))$. The system Π evolves from an initial configuration to other configurations by means of *computation steps*, in which one or more programs of Π (depending upon the *mode* of computation) are executed. In [17], at each computation step the programs to be executed are chosen in the so called *sequential* mode: one program is nondeterministically chosen in each region, among the programs that can be executed at that time. Another possibility is to select the programs in the so called *all-parallel* mode: in each region, all the programs that can be executed are selected, with each variable participating in all programs where it appears. Note that in this case EN P systems become *deterministic*, since nondeterministic choices between programs never occur. A variant of parallelism, analogous to the maximal one which is often used in membrane computing, is the so called *one-parallel* mode: in each region, all the programs which can be executed can be selected, but the actual selection is made in such a way that each variable participates in only one of the chosen programs. We say that the system reaches a *final configuration* if and when it happens that no applicable set of programs produces a change in the current configuration. In such a case, a specified set of variables contains the output of the computation. Of course, a computation may never reach a final configuration. Note that in the usual definition of EN P systems the output of a computation is instead defined as the collection of values taken by a specified set of variables during the whole computation. In what follows we prove our results both by considering outputs in the final configurations, and by the latter notion of producing the output.

EN P systems can be used to compute functions, in the so called *computing mode*, by considering some *input variables* and *output variables*. The initial values of the input variables are considered the actual arguments of the function, while the value of the output variables in the final configuration (provided that the system reaches it) is viewed as the output of the computed function. If the system never reaches a final configuration, then the computed function is undefined for the specified input values. By neglecting input variables, (nondeterministic) EN P systems can also be used in the *generating mode*, whereas by neglecting output variables we can use (deterministic or nondeterministic) EN P systems in the *accepting mode*, where the input is accepted if the system reaches a final configuration.

A technical detail to take care of is the fact that normally we would like to characterize families of sets of *natural numbers* (sometimes including and sometimes excluding zero), while the input and output variables of EN P systems may also assume negative values. The systems we will propose are designed to produce only non-negative numbers in the output variables when the input variables (if present) are assigned with non-negative numbers. So if the systems are used in the intended way, they always produce meaningful (and correct) results. Another possibility, mentioned in [17] but not considered here, is to filter the output values so that only the positive ones are considered as output.

When using EN P systems in the generating or accepting modes, we denote by $\mathbf{ENP}_m(poly^n(r), app\_mode)$ the family of sets of (possibly vectors of) non-negative integer numbers which are computed by EN P systems of degree $m \geq 1$, using polynomials of degree at most $n \geq 0$ with at most $r \geq 0$ arguments as production functions; the fact that the programs are applied in the sequential, one-parallel or all-parallel mode is denoted by assigning the value *seq*, *oneP* or *allP* to the *app_mode* parameter, respectively. When $app\_mode \in \{seq, oneP\}$ and the P system is deterministic, we write *det* after the *app_mode* parameter; this specification is not needed for all-parallel EN P systems, since they are always deterministic. If one of the parameters m, n, r is not bounded by a constant value, we replace it by $*$.

With this notation, we can summarize the characterizations of $\mathbb{N}\mathbf{RE}$ proved in [17] as follows:

$$\begin{aligned}\mathbb{N}\mathbf{RE} &= \mathbf{ENP}_7(poly^5(5), seq) = \mathbf{ENP}_*(poly^1(2), oneP) \\ &= \mathbf{ENP}_{254}(poly^2(253), allP)\end{aligned}$$

whereas the improvement of the last equality given in [16] can be written as $\mathbb{N}\mathbf{RE} = \mathbf{ENP}_4(poly^1(6), allP)$.

In section 3 we further improve the results concerning EN P systems working in the all-parallel and in the one-parallel modes: in both cases, we will obtain characterizations of $\mathbb{N}\mathbf{RE}$ by using just one membrane, and linear production functions that use each at most one variable.

2.2 Register Machines

In what follows we will simulate register machines, so we briefly recall their definition and some of their computational properties.

An *n–register machine* is a construct $M = (n, P, m)$, where $n > 0$ is the number of registers, P is a finite sequence of instructions bijectively labelled with the elements of the set $\{0, 1, \ldots, m-1\}$, 0 is the label of the first instruction to be executed, and $m-1$ is the label of the last instruction of P. Registers contain non-negative integer values. The instructions of P have the following forms:

- $j : (\text{INC}(r), k, l)$, with $0 \leq j < m$, $0 \leq k, l \leq m$ and $1 \leq r \leq n$.
  This instruction, labelled with j, increments the value contained in register r, then nondeterministically jumps either to instruction k or to instruction l.
- $j : (\text{DEC}(r), k, l)$, with $0 \leq j < m$, $0 \leq k, l \leq m$ and $1 \leq r \leq n$.
  If the value contained in register r is positive then decrement it and jump to instruction k. If the value of r is zero then jump to instruction l (without altering the contents of the register).

A *deterministic* n-register machine is an n-register machine in which all INC instructions have the form $j : (\text{INC}(r), k, k)$; in what follows, we will write these instructions simply as $j : (\text{INC}(r), k)$.

A *configuration* of an n-register machine M is described by the contents of each of its registers and by the program counter, that indicates the next instruction to be executed. Computations start by executing the first instruction of P (labelled with 0), and possibly terminate when the instruction currently executed jumps to label m (we may equivalently assume that P includes the instruction m : HALT, explicitly stating that the computation must halt).

It is well known that register machines provide a simple universal computational model, and that machines with three registers suffice to characterize $\mathbb{N}$**RE** [8]. More precisely, we can use register machines in the computing, generating or accepting mode, obtaining the following results [3, 4, 5]. For the computing mode, we have:

Proposition 1. *For any partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ $(\alpha, \beta > 0)$, there exists a deterministic register machine M with $(\max\{\alpha, \beta\} + 2)$ registers computing f in such a way that, when starting with n_1 to n_α in registers 1 to α, M has computed $f(n_1, \ldots, n_\alpha) = (r_1, \ldots, r_\beta)$ if it halts in the final label m with registers 1 to β containing r_1 to r_β, and all other registers being empty; if $f(n_1, \ldots, n_\alpha)$ is undefined then the final label of M is never reached.*

In accepting register machines, a vector of non-negative integers is accepted if and only if the register machine halts:

Proposition 2. *For any recursively enumerable set $L \subseteq$ **Ps**(α)**RE** of vectors of non-negative integers there exists a deterministic register machine M with $(\alpha + 2)$ registers accepting L in such a way that, when starting with n_1 to n_α in registers 1 to α, M has accepted $(n_1, \ldots, n_\alpha) \in L$ if and only if it halts in the final label m with all registers being empty.*

To generate vectors of non-negative integers, we need nondeterministic register machines:

Proposition 3. *For any recursively enumerable set $L \subseteq$ **Ps**(β)**RE** of vectors of non-negative integers there exists a non-deterministic register machine M with $(\beta + 2)$ registers generating L, i.e., when starting with all registers being empty, M generates $(r_1, \ldots, r_\beta) \in L$ if it halts in the final label m with registers 1 to β containing r_1 to r_β, and all other registers being empty.*

3 Universality of EN P Systems

As stated above, our aim is to improve the universality results shown in [17, 16], concerning all-parallel and one-parallel EN P systems. We first prove that these P systems can be "flattened".

Theorem 1. *Let Π be any computing (or generating, or accepting) EN P system of degree $m \geq 1$, working in the all-parallel or in the one-parallel mode. Then there exists an EN P system Π' of degree 1 that computes (resp., generates, accepts) the same function (resp., family of sets) using the same rule application mode.*

Proof. Let $\Pi = (m, H, \mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)))$ be an EN P system, computing a function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ $(\alpha, \beta \geq 0)$ and working in the all-parallel mode. All the other cases (one-parallel, generating and accepting modes) can be simply deduced from the following argumentation.

Note that each variable $x_{j,i} \in Var_i$ and each program $P_{l,i} \in Pr_i$ already indicates in one of its indexes the region that contains it. We build a new EN P system Π' of degree 1, by putting all the variables and all the programs of Π — keeping both indexes, also in the variables occurring in programs — in the membrane of Π'. Clearly, this establishes a bijection between the variables (resp., programs) of Π and the corresponding variables (resp., programs) of Π', since the presence of both indexes in Π' allows one to keep track of the region of Π from which each variable and each program comes from. So any program $P_{l,i}$ of Π still operates on the correct variables when transformed and put into Π', regardless of whether or not it uses an enzyme. Also input and output variables are preserved, and so the only issue is related with the mode used to select the programs to be applied. If Π works in the sequential mode, then at each computation step only (at most) one program is selected in each region; this means that globally Π executes a set of programs which cannot be captured in Π' by any of the sequential, one-parallel and all-parallel modes. Instead, if Π works in the all-parallel mode then at each computation step all the programs that can be executed are selected, and the same happens in Π' by letting it work in the all-parallel mode. The same applies when Π and Π' work in the one-parallel mode, and so the claim of the theorem follows. □

This result already allows to improve the universality results shown in [17, 16] for all-parallel and one-parallel EN P systems, obtaining the following characterizations of $\mathbb{N}\mathbf{RE}$:

$$\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(6), allP) = \mathbf{ENP}_1(poly^1(2), oneP)$$

However — as stated in the Introduction — despite this simplification, one-parallel EN P systems still require an unbounded number of variables, since each "new" variable in Π' is indexed with the region of Π it comes from.

Anyhow, we can improve both results. We start with the first equality, concerning all-parallel EN P systems.

Theorem 2. *Each partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ $(\alpha > 0,\ \beta \geq 0)$ can be computed by a one-membrane EN P system working in the all-parallel mode, having linear production functions that use each at most one variable.*

Proof. Since all-parallel EN P systems are deterministic, we prove the statement by simulating deterministic register machines. Let $M = (n, P, m)$ be such a machine with n registers, computing f by means of program P. The initial instruction of P has the label 0 and the machine halts if and when the program counter assumes the value m. Observe that according to the result stated in Proposition 1, $n = \max\{\alpha, \beta\} + 2$ is enough. The input values $x_1, \ldots, x_\alpha$ are expected to be in the

first α registers before the computation starts, and the values of $f(x_1, \ldots, x_\alpha)$ — if any — are expected to be in registers 1 to β at the end of a halting computation. Moreover, without loss of generality, we may assume that at the beginning of a computation all the registers except possibly the registers 1 to α contain zero.

We construct the EN P system $\Pi_M = (1, H, \mu, (Var_1, Pr_1, Var_1(0)))$ of degree 1, where:

- $H = \{s\}$ is the label of the only membrane (the skin) of Π_M;
- $\mu = [\,]_s$ is the membrane structure;
- $Var_1 = \{r_1, \ldots, r_n\} \cup \{p_0, \ldots, p_m\}$;
- $Pr_1 = \{2p_j \to 1|r_i + 1|p_k$ for all instructions $j : (\text{INC}(i), k) \in P\} \cup \{-p_j \to 1|r_i,\ r_i + 2|_{p_j} \to 1|r_i + 1|p_l,\ p_j \to 1|p_k,\ r_i - 1|_{p_j} \to 1|p_k$ for all instructions $j : (\text{DEC}(i), k, l) \in P\}$;
- $Var_1(0)$ is the vector of initial values of the variables of Var_1, obtained by putting:
  - $r_i = x_i$ for all $1 \leq i \leq \alpha$;
  - $r_i = 0$ for all $\alpha + 1 \leq i \leq n$;
  - $p_0 = 1$;
  - $p_j = 0$ for all $1 \leq j \leq m$.

The value of register i, for $1 \leq i \leq m$, is contained in variable r_i. The input values $x_1, \ldots, x_\alpha$ are introduced into the P system as the initial values of variables $r_1, \ldots, r_\alpha$. Variables $p_0, \ldots, p_m$ are used to indicate the value of the program counter; at the beginning of each computation step, the variable corresponding to the value of the program counter of M will assume value 1, while all the others will be equal to zero.

The simulation of M by Π_M works as follows. Each increment instruction $j : (\text{INC}(i), k)$ is simulated in one step by the execution of the program

$$2p_j \to 1|r_i + 1|p_k$$

This program is executed at *every* computation step of Π_M; however, when $p_j = 0$ it has no effect: p_j is once again set to zero, and a contribution of zero is distributed among variables r_i and p_k. All variables are thus unaffected in this case. When $p_j = 1$, the production value $2p_j = 2$ is distributed among r_i and p_k, giving a contribution of 1 to each of them. Hence the value of r_i is incremented, the value of p_k passes from 0 to 1, while the value of p_j is zeroed. All the other variables are unaffected, and the system is now ready to simulate the next instruction of M.

Each decrement instruction $j : (\text{DEC}(i), k, l)$ is simulated in one step by the parallel execution of the following programs:

$$-p_j \to 1|r_i \tag{2}$$

$$r_i + 2|_{p_j} \to 1|r_i + 1|p_l \tag{3}$$

$$p_j \to 1|p_k \tag{4}$$

$$r_i - 1|_{p_j} \to 1|p_k \tag{5}$$

If $p_j = 0$, programs (3) and (5) are not enabled (since by construction $r_i \geq 0$ and thus $p_j \leq r_i$), while programs (2) and (4) distribute a contribution of zero to r_i and p_k; before doing so, variable p_j is set to zero, thus leaving its value unchanged. Hence, the case in which $p_j = 0$ causes no problems to the overall simulation.

Now assume that $p_j = 1$ and $r_i > 0$. In this case, the value of r_i should be decremented and the computation should continue with instruction k. Program (2) correctly decrements r_i, and program (4) passes the value of $p_j = 1$ to p_k, thus correctly pointing at the next istruction of M to be simulated. The execution of both programs sets the value of p_j to zero, which is also correct. Programs (3) and (5) have no effect since to be executed it should be $p_j > r_i$, that is, $r_i < 1$ (which means $r_i = 0$, since $r_i \geq 0$ by construction).

Now assume that $p_j = 1$ and $r_i = 0$. In this case, the value of r_i should be kept equal to zero, and the computation should continue with instruction l. Program (2) sends a contribution of -1 to r_i, while program (4) sets — incorrectly — p_k to 1; both programs set p_j to zero. This time, however, programs (3) and (5) are also executed. Both set the value of r_i to zero. After that, program (3) adds 1 to r_i, thus canceling the effect of program (2); as a result, the value assumed by r_i after the execution of the two programs is zero. Program (3) also makes p_l assume the value 1, thus correctly pointing to the next instruction of M to be simulated. Finally, program (5) gives a contribution of -1 to p_k, canceling the effect of program (4); the resulting value of p_k will thus be 0.

It follows from the description given above that after the simulation of each instruction of M the value of every variable r_i equals the contents of register i, for $1 \leq i \leq n$, while the only variable among $p_0, \ldots, p_m$ equal to 1 indicates the next instruction of M to be simulated. When the program counter of M reaches the value m, the corresponding variable p_m assumes value 1. Since no program contains the variable p_m either in the production function or among the enzymes that enable or disable the execution of the program, Π_M reaches a final configuration; the result of the computation is contained in variables $r_1, \ldots, r_\beta$. □

By taking $\beta = 0$ in the previous proof, we get the following result concerning the accepting variant of EN P systems working in the all-parallel mode.

Corollary 1. *For any $L \in \mathbf{Ps}(\alpha)\mathbf{RE}$ there exists a one-membrane EN P system, having linear production functions each depending upon at most one variable, that accepts L by working in the all-parallel mode.*

Proof. We consider a register machine M with $(\alpha + 2)$ registers accepting L according to Proposition 2, and we construct the one-membrane EN P system Π_M that accepts L following the construction given in the proof of Theorem 2. The input values $x_1, \ldots, x_\alpha$ expected to be in the first α registers in M are assigned as initial values to variables r_1 to r_α in Π_M, whereas the initial values of variables $r_{\alpha+1}$ to r_n are 0. The P system Π_M accepts this input if and only if it reaches a final configuration. □

By putting $\alpha = 1$ in Corollary 1, we obtain the following characterization:

$0 : (\text{DEC}(2), 1, 2)$ $1 : (\text{INC}(8), 0)$
$2 : (\text{INC}(7), 3)$ $3 : (\text{DEC}(6), 2, 4)$
$4 : (\text{DEC}(7), 5, 3)$ $5 : (\text{INC}(6), 6)$
$6 : (\text{DEC}(8), 7, 8)$ $7 : (\text{INC}(2), 4)$
$8 : (\text{DEC}(7), 9, 0)$ $9 : (\text{INC}(7), 10)$
$10 : (\text{DEC}(5), 0, 11)$ $11 : (\text{DEC}(6), 12, 13)$
$12 : (\text{DEC}(6), 14, 15)$ $13 : (\text{DEC}(3), 18, 19)$
$14 : (\text{DEC}(6), 16, 17)$ $15 : (\text{DEC}(4), 18, 20)$
$16 : (\text{INC}(5), 11)$ $17 : (\text{INC}(3), 21)$
$18 : (\text{DEC}(5), 0, 22)$ $19 : (\text{DEC}(1), 0, 18)$
$20 : (\text{INC}(1), 0)$ $21 : (\text{INC}(4), 18)$

Fig. 1. The small universal deterministic register machine defined in [6]

$$\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(1), allP)$$

A direct consequence of Theorem 2 is that there exists a *small* universal all-parallel EN P system that computes every possible partial recursive function.

Theorem 3. *There exists a universal all-parallel EN P system of degree* 1*, having* 31 *variables and* 61 *programs.*

Proof. We consider the small universal deterministic register machine M_u described in [6], and illustrated in Figure 1. This machine has $n = 8$ registers and $m = 22$ instructions, and can be used to compute any unary partial recursive function $f : \mathbb{N} \to \mathbb{N}$ as follows. Let $(\varphi_0, \varphi_1, \ldots)$ be a fixed admissible enumeration of the unary partial recursive functions. Since M_u is universal, there exists a recursive function g such that for all natural numbers y, z it holds $\varphi_y(z) = M_u(g(y), z)$. Hence, to compute $f(x)$ we first consider the index y of f in the above enumeration of unary recursive functions. Then we put $g(y)$ and x in registers 2 and 3 of M_u, respectively, and we start the computation; the value of $f(x)$ will be found in register 1 if and when M_u halts.

By following the arguments given in the proof of Theorem 2 we construct the all-parallel EN P system $\Pi_{M_u} = (1, H, \mu, (Var_1, Pr_1, Var_1(0)))$ of degree 1, where:

- $H = \{s\}$ is the label of the only membrane (the skin) of Π;
- $\mu = [\,]_s$ is the membrane structure;
- $Var_1 = \{r_1, \ldots, r_8\} \cup \{p_0, \ldots, p_{22}\}$;
- $Pr_1 = \{2p_j \to 1|r_i + 1|p_k$ for all instructions $j : (\text{INC}(i), k)$ listed in Figure 1$\} \cup \{-p_j \to 1|r_i,\ r_i + 2|_{p_j} \to 1|r_i + 1|p_l,\ p_j \to 1|p_k,\ r_i - 1|_{p_j} \to 1|p_k$ for all instructions $j : (\text{DEC}(i), k, l)$ listed in Figure 1$\}$;

- $Var_1(0)$ is the vector of initial values of the variables of Var_1, obtained by putting:
  - $r_2 = g(y)$, the "code" associated to function f;
  - $r_3 = x$, the input of f;
  - $r_1 = r_4 = r_5 = r_6 = r_7 = r_8 = 0$;
  - $p_0 = 1$;
  - $p_i = 0$ for all $1 \leq i \leq 22$.

This system simulates the operation of M_u, as described in the proof of Theorem 2. Hence, if and when the computation reaches a final configuration, variable r_1 contains the value of $f(x)$.

The number of increment and decrement instructions of M_u are 9 and 13, respectively. Each increment instruction is translated to 1 program of Π_{M_u} while each decrement instruction produces 4 programs, for a total of 61 programs. The variables are $n + m + 1 = 31$. □

We now turn to EN P systems working in the one-parallel mode. We start proving the following theorem.

Theorem 4. *Each partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ $(\alpha, \beta \geq 0)$ can be computed by a one-membrane EN P system working in the one-parallel mode, having linear production functions that use each at most two variables.*

Proof. We proceed like in the proof of Theorem 2, with the difference that here we simulate both deterministic and nondeterministic register machines. Let $M = (n, P, m)$ be a nondeterministic register machine with $n = \max\{\alpha, \beta\} + 2$ registers, that computes f by means of program P. As usual, the input values $x_1, \ldots, x_\alpha$ are expected to be in the first α registers before the computation starts, all the other registers being empty. If and when the computation of M halts, the values of $f(x_1, \ldots, x_\alpha)$ will be found in registers 1 to β.

We construct the one-membrane EN P system $\Pi_M = (1, H, \mu, (Var_1, Pr_1, Var_1(0)))$, where:

- $H = \{s\}$ is the label of the only membrane (the skin) of Π_M;
- $\mu = [\,]_s$ is the membrane structure;
- $Var_1 = \{r_1, \ldots, r_n\} \cup \{p_0, \ldots, p_m\} \cup \{q_0, \ldots, q_m\} \cup \{z_{j,1}, z_{j,2}, z_{j,3}$ for all instructions $j : (\text{INC}(i), k, l) \in P\} \cup \{z_{j,1}, z_{j,2}, z_{j,3}, z_{j,4}, z_{j,5}$ for all instructions $j : (\text{DEC}(i), k, l) \in P\}$;
- $Pr_1 = \{z_{j,1}+3|_{p_j} \to 1|r_i+1|p_k+1|q_k,\ z_{j,1}+3|_{p_j} \to 1|r_i+1|p_l+1|q_l,\ z_{j,2}-1|_{p_j} \to 1|q_j,\ z_{j,3}-1|_{q_j} \to 1|p_j$ for all instructions $j : (\text{INC}(i), k, l) \in P\} \cup \{z_{j,1}-1|_{p_j} \to 1|r_i,\ r_i + 3|_{p_j} \to 1|r_i + 1|p_l + 1|q_l,\ z_{j,2} + 2p_j|_{r_i} \to 1|p_j + 1|p_k,\ z_{j,3} + 2q_j|_{r_i} \to 1|q_j + 1|q_k,\ z_{j,4} - 1|_{p_j} \to 1|q_j,\ z_{j,5} - 1|_{q_j} \to 1|p_j\}$ for all instructions $j : (\text{DEC}(i), k, l) \in P\}$;
- $Var_1(0)$ is the vector of initial values of the variables of Var_1, obtained by putting:
  - $r_i = x_i$ for all $1 \leq i \leq \alpha$;

- $r_i = 0$ for all $\alpha + 1 \leq i \leq n$;
- $p_0 = q_0 = 1$;
- $p_j = q_j = 0$ for all $1 \leq j \leq m$;
- $z_{j,1} = z_{j,2} = z_{j,3} = 0$ for all $0 \leq j < m$ such that $j : (\text{INC}(i), k, l) \in P$;
- $z_{j,1} = z_{j,2} = z_{j,3} = z_{j,4} = z_{j,5} = 0$ for all $0 \leq j < m$ such that $j : (\text{DEC}(i), k, l) \in P$.

Just like in the proof of Theorem 2, the value of register i, for $1 \leq i \leq n$, is contained in variable r_i, and the input values $x_1, \ldots, x_\alpha$ are introduced into the P system as the initial values of variables $r_1, \ldots, r_\alpha$. This time, however, the system uses both variables $p_0, \ldots, p_m$ and $q_0, \ldots, q_m$ to indicate the value of the program counter of M, so that when simulating the j-th instruction of P variables p_j and q_j are both set to 1, while all the others are zero. This double representation of the program counter will allow us to set its value while also using it as an enzyme: precisely, variable p_j will be used as an enzyme to update the value of q_j, and vice versa. The auxiliary variables $z_{j,1}, \ldots, z_{j,5}$, when defined, are used during the simulation of INC and DEC instructions, and are always set to zero.

The simulation of M by Π_M works as follows. Each increment instruction $j : (\text{INC}(i), k, l)$ is simulated in one step by the execution of the following programs:

$$z_{j,1} + 3|_{p_j} \rightarrow 1|r_i + 1|p_k + 1|q_k \tag{6}$$

$$z_{j,1} + 3|_{p_j} \rightarrow 1|r_i + 1|p_l + 1|q_l \tag{7}$$

$$z_{j,2} - 1|_{p_j} \rightarrow 1|q_j \tag{8}$$

$$z_{j,3} - 1|_{q_j} \rightarrow 1|p_j \tag{9}$$

These programs are not executed when $p_j = q_j = 0$, since variables $z_{j,1}$, $z_{j,2}$ and $z_{j,3}$ are zero, hence in this case they have no effect. When $p_j = q_j = 1$, instead, programs (8) and (9) as well as one among programs (6) and (7) are executed, since variable $z_{j,1}$ makes these latter programs compete in the one-parallel mode of application. Assume that program (6) wins the competition (a similar argument holds if (7) wins instead): its effect is incrementing r_i and setting p_k and q_k to 1, thus correctly pointing to the next instruction of M to be simulated. The effect of programs (8) and (9) is giving a contribution of -1 to both p_j and q_j, whose final value will thus be zero. All the other variables are unaffected. If M is deterministic, then the simulation of the instruction $j : (\text{INC}(i), k)$ is performed by using the same programs without (7). In this case no competition occurs between the programs, and so the simulation is deterministic.

Each decrement instruction $j : (\text{DEC}(i), k, l)$ is simulated in one step by the execution of the following programs:

$$z_{j,1} - 1|_{p_j} \to 1|r_i \tag{10}$$
$$r_i + 3|_{p_j} \to 1|r_i + 1|p_l + 1|q_l \tag{11}$$
$$z_{j,2} + 2p_j|_{r_i} \to 1|p_j + 1|p_k \tag{12}$$
$$z_{j,3} + 2q_j|_{r_i} \to 1|q_j + 1|q_k \tag{13}$$
$$z_{j,4} - 1|_{p_j} \to 1|q_j \tag{14}$$
$$z_{j,5} - 1|_{q_j} \to 1|p_j \tag{15}$$

If $p_j = q_j = 0$ then programs (10), (11), (14) and (15) are not enabled, while programs (12) and (13) are enabled only if $r_i > 0$. However, in this case they set to 0 variables p_j and q_j (thus leaving their value unaltered), and distribute a contribution of zero to p_j, q_j, p_k and q_k, thus producing no effect. All the other variables are left unchanged, so no problems occur to the overall simulation.

Now assume that $p_j = q_j = 1$ and $r_i > 0$. In this case, the value of r_i should be decremented and the computation should continue with instruction k. Program (10) correctly decrements r_i, whereas program (11) is not executed since $r_i \geq p_j$. Programs (12) and (13) set to 1 variables p_k and q_k (thus pointing at the next instruction of M to be simulated), and send a contribution of 1 to variables p_j and q_j, after setting their value to zero. On the other hand, programs (14) and (15) send a contribution of -1 to p_j and q_j, so that their final value will be zero.

Now assume that $p_j = q_j = 1$ and $r_i = 0$. In this case, the value of r_i should be kept equal to zero, and the computation should continue with instruction l. Program (10) sends a contribution of -1 to r_i. This time, however, program (11) is also executed; its effect is sending a contribution of 1 to r_i, after setting it to zero (so that its final value will be zero), and setting to 1 the value of variables p_l and q_l. Programs (12) and (13) are inactive, and hence are not executed. Finally, programs (14) and (15) send a contribution of -1 to p_j and q_j, so that their final value will be zero.

It follows from the description given above that after the simulation of each instruction of M the value of every variable r_i equals the contents of register i, for $1 \leq i \leq n$, while variables $p_0, \ldots, p_m$ and $q_0, \ldots, q_m$ correctly indicate the next instruction of M to be simulated. When the program counter of M reaches the value m, the corresponding variables p_m and q_m assume value 1. Since no program contains these variables either in the production function or among the enzymes, the simulation reaches a final configuration; the result of the computation is contained in variables $r_1, \ldots, r_\beta$. □

By taking $\beta = 0$ and $\alpha \geq 1$ in the previous proof, we obtain the following result concerning the accepting variant of EN P systems working in the one-parallel mode.

Corollary 2. *For any $L \in \mathbf{Ps}(\alpha)\mathbf{RE}$ there exists a one-membrane EN P system, having linear production functions each depending upon at most two variables, that accepts L by working in the one-parallel mode.*

On the other hand, by taking $\alpha = 0$ and $\beta \geq 1$ we get the following characterization of $\mathbf{Ps}(\beta)\mathbf{RE}$ by the generating variant of EN P systems working in the one-parallel mode.

Corollary 3. *For any $L \in \mathbf{Ps}(\beta)\mathbf{RE}$ there exists a one-membrane (nondeterministic) EN P system, having linear production functions each depending upon at most two variables, that generates L by working in the one-parallel mode.*

By putting $\alpha = 1$ and $\beta = 0$ in Corollary 2, and $\alpha = 0$ and $\beta = 1$ in Corollary 3, we obtain the following characterization:

$$\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(2), oneP)$$

Another consequence of Theorem 4 is that there exists the small universal deterministic one-parallel EN P system mentioned in the following theorem.

Theorem 5. *There exists a universal one-parallel deterministic EN P system of degree* 1*, having* 146 *variables and* 105 *programs.*

Proof. The system mentioned in the statement simulates the small universal deterministic register machine M_u reported in Figure 1, and is built according to the description given in the proof of Theorem 4, as we have done in the proof of Theorem 3. The number of increment and decrement instructions of M_u are 9 and 13, respectively. Each increment and each decrement instruction is translated to 3 and 6 programs of the small universal EN P system, respectively, for a total of 105 programs. As for variables, 8 are used to simulate the registers of M_u, and 46 are used to denote the value of its program counter; moreover, there are 3 and 5 auxiliary variables for each increment and each decrement instruction, respectively, for a total of 146 variables. □

Let us note that, since the EN P system mentioned in the statement of Theorem 5 is deterministic, it also works in the all-parallel mode, albeit in this case the system described in Theorem 3 is smaller.

By looking at the operation of the EN P system described in the proof of Theorem 4, we can see that the only programs whose production functions depend upon two variables are programs (12) and (13). Further, if we remove variables $z_{j,2}$ and $z_{j,3}$ from these programs the simulation of register machine M continues to work correctly, except in the case when $r_i = 1$ and $p_j = q_j = 1$. Hence if r_i could only assume even values (so that the value $2v$ denotes the fact that the contents of the i-th register of M is v) we could get rid of variables $z_{j,2}$ and $z_{j,3}$ in programs (12) and (13), thus obtaining a one-parallel EN P system whose linear production functions each depend on just one variable. This is exactly what we do in the next theorem, where $2\mathbb{N}$ denotes the set of even natural numbers.

Theorem 6. *Each partial recursive function $f : (2\mathbb{N})^\alpha \to (2\mathbb{N})^\beta$ $(\alpha, \beta \geq 0)$ can be computed by a one-membrane EN P system working in the one-parallel mode, having linear production functions that use each at most one variable.*

Proof. The proof is similar to the one given for Theorem 4. The one-parallel EN P system Π_M that simulates the nondeterministic register machine $M = (n, P, m)$ is now defined as follows:

$$\Pi_M = (1, H, \mu, (Var_1, Pr_1, Var_1(0)))$$

where:

- $H = \{s\}$ is the label of the only membrane (the skin) of Π_M;
- $\mu = [\,]_s$ is the membrane structure;
- $Var_1 = \{r_1, \ldots, r_n\} \cup \{p_0, \ldots, p_m\} \cup \{q_0, \ldots, q_m\} \cup \{z_{j,1}, z_{j,2}, z_{j,3}$ for all $0 \leq j < m\}$;
- $Pr_1 = \{z_{j,1}+4|_{p_j} \to 2|r_i+1|p_k+1|q_k,\ z_{j,1}+4|_{p_j} \to 2|r_i+1|p_l+1|q_l,\ z_{j,2}-1|_{p_j} \to 1|q_j,\ z_{j,3}-1|_{q_j} \to 1|p_j$ for all instructions $j : (\text{INC}(i), k, l) \in P\} \cup \{z_{j,1}-2|_{p_j} \to 1|r_i,\ r_i+4|_{p_j} \to 2|r_i+1|p_l+1|q_l,\ 2p_j|_{r_i} \to 1|p_j+1|p_k,\ 2q_j|_{r_i} \to 1|q_j+1|q_k,\ z_{j,2}-1|_{p_j} \to 1|q_j,\ z_{j,3}-1|_{q_j} \to 1|p_j\}$ for all instructions $j : (\text{DEC}(i), k, l) \in P\}$;
- $Var_1(0)$ is the vector of initial values of the variables of Var_1, obtained by putting:
  - $r_i = 2x_i$ for all $1 \leq i \leq \alpha$;
  - $r_i = 0$ for all $\alpha + 1 \leq i \leq n$;
  - $p_0 = q_0 = 1$;
  - $p_j = q_j = 0$ for all $1 \leq j \leq m$;
  - $z_{j,1} = z_{j,2} = z_{j,3} = 0$ for all $0 \leq j < m$.

As stated above, now the value of r_i is the double of the value of register i, for $1 \leq i \leq n$. So, in particular, the double of the input values $x_1, \ldots, x_\alpha$ are introduced into the P system as the initial values of variables $r_1, \ldots, r_\alpha$. Once again, like in the proof of Theorem 4, the system uses both variables $p_0, \ldots, p_m$ and $q_0, \ldots, q_m$ to indicate the value of the program counter of M, so that when simulating the j-th instruction of P variables p_j and q_j are both equal to 1, while all the others are zero. The value of variables $z_{j,1}, z_{j,2}, z_{j,3}$ is always zero during the entire computation.

Each increment instruction $j : (\text{INC}(i), k, l)$ of M is simulated in one step by the execution of the following programs:

$$z_{j,1} + 4|_{p_j} \to 2|r_i + 1|p_k + 1|q_k \tag{16}$$

$$z_{j,1} + 4|_{p_j} \to 2|r_i + 1|p_l + 1|q_l \tag{17}$$

$$z_{j,2} - 1|_{p_j} \to 1|q_j \tag{18}$$

$$z_{j,3} - 1|_{q_j} \to 1|p_j \tag{19}$$

The simulation is analogous to the one described in the proof of Theorem 4, with the difference that instead of incrementing r_i the system now adds 2 to it; to do so, the production value computed by the first two programs must be 4 instead of 3. Nondeterminism is given by the fact that, when $p_j = q_j = 1$, variable $z_{j,1}$ makes programs (16) and (17) compete in the one-parallel mode. If the machine M to be simulated is deterministic, then program (17) disappears, and so the simulation becomes deterministic.

Each decrement instruction $j : (\text{DEC}(i), k, l)$ is simulated in one step by the execution of the following programs:

$$z_{j,1} - 2|_{p_j} \rightarrow 1|r_i \tag{20}$$
$$r_i + 4|_{p_j} \rightarrow 2|r_i + 1|p_l + 1|q_l \tag{21}$$
$$2p_j|_{r_i} \rightarrow 1|p_j + 1|p_k \tag{22}$$
$$2q_j|_{r_i} \rightarrow 1|q_j + 1|q_k \tag{23}$$
$$z_{j,2} - 1|_{p_j} \rightarrow 1|q_j \tag{24}$$
$$z_{j,3} - 1|_{q_j} \rightarrow 1|p_j \tag{25}$$

The simulation is analogous to the one described in the proof of Theorem 4, with small differences.

The case when $p_j = q_j = 0$ operates just like in the proof of Theorem 4: programs (20), (21), (24) and (25) are not active, while programs (22) and (23) are executed only if $r_i > 0$; however, in such a case, a contribution of 0 is distributed to variables p_j, q_j, p_k, q_k after setting p_j and q_j to zero.

Now assume that $p_j = q_j = 1$ and $r_i > 0$. Program (20) correctly decrements r_i (subtracting 2 from its value), whereas program (21) is not executed since $r_i > p_j$. Programs (22) and (23) set to 1 variables p_k and q_k (thus pointing at the next instruction of M to be simulated), and send a contribution of 1 to variables p_j and q_j, after setting their value to zero. On the other hand, programs (24) and (25) send a contribution of -1 to p_j and q_j, so that their final value will be zero.

Now assume that $p_j = q_j = 1$ and $r_i = 0$. In this case, the value of r_i should be kept equal to zero, and the computation should continue with instruction l. Program (20) sends a contribution of -2 to r_i. This time, however, program (21) is also executed; its effect is sending a contribution of 2 to r_i, after setting it to zero (so that its final value will be zero), and setting to 1 the value of variables p_l and q_l. Programs (22) and (23) are inactive, and hence are not executed. Finally, programs (24) and (25) send a contribution of -1 to p_j and q_j, so that their final value will be zero.

It follows from the description given above that the simulation is correct, and that after the simulation of each instruction the value of variable r_i is exactly the double of the contents of register i, for $1 \leq i \leq n$. If and when the program counter of M reaches the value m, the corresponding variables p_m and q_m assume value 1 and the computation reaches a final configuration; the result of the computation is then contained in variables $r_1, \ldots, r_\beta$. □

Let $2\mathbb{N}\mathbf{RE}$ denote the family of recursively enumerable sets of even natural numbers: $2\mathbb{N}\mathbf{RE} = \{\{2x \mid x \in X\} \mid X \in \mathbb{N}\mathbf{RE}\}$. By taking $\beta = 0$ and $\alpha \geq 1$ (resp., $\alpha = 0$ and $\beta \geq 1$) in the previous proof one obtains a characterization of the recursively enumerable sets of vectors of even natural numbers by accepting (resp., generating) one-parallel EN P systems. In particular, by putting $\beta = 0$ and $\alpha = 1$ or $\alpha = 0$ and $\beta = 1$, we obtain:

$$2\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(1), oneP)$$

As a byproduct of Theorem 6 we also obtain a small universal deterministic EN P system that computes any partial recursive function $f : 2\mathbb{N} \rightarrow 2\mathbb{N}$, by simulating

the universal deterministic register machine illustrated in Figure 1. With respect to the small EN P system described in the proof of Theorem 5 we have removed two auxiliary variables from the programs that simulate each decrement instruction, hence the new system consists of 105 programs and 120 variables. As discussed after the proof of Theorem 5, this small EN P system is deterministic too and hence it also works in the all-parallel mode; however, it works only with even natural numbers as inputs and outputs.

Of course one would desire a characterization of $\mathbb{N}\mathbf{RE}$ (instead of $2\mathbb{N}\mathbf{RE}$) by one-parallel EN P systems having linear production functions, each depending upon just one variable. We can actually obtain such a characterization by using the EN P system Π_M described in the proof of the previous theorem as a subroutine. The idea is to produce a new one-parallel EN P system Π'_M that, given a vector from $\mathbb{N}^\alpha$ as input, prepares a corresponding input vector for Π_M by doubling its components. Then Π_M is used to compute the output vector from $\mathbb{N}^\beta$, if it exists. At this point Π'_M should take this output and halve each component, to produce its output. To avoid this further step, we proceed as follows: while preparing the input for Π_M, Π'_M also makes a copy of its input into additional variables s_i, for $1 \leq i \leq n$. Then we modify the programs of Π_M in such a way that, while simulating a (possibly nondeterministic) register machine M, it keeps in s_i the contents of the registers, and in r_i the doubles of such contents. So the programs use variables r_i to correctly perform the simulation, while at the end of the computation the result will be immediately available in variables s_i. The details are given in the proof of the following theorem, where the systems Π_M and Π'_M are combined together.

Theorem 7. *Each partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$ ($\alpha \geq 0,\ \beta \geq 0$) can be computed by a one-membrane EN P system working in the one-parallel mode, having linear production functions that use each at most one variable.*

Proof. Like in the proofs of Theorems 4 and 6, we build a one-parallel EN P system $\Pi_M = (1, H, \mu, (Var_1, Pr_1, Var_1(0)))$ that simulates a nondeterministic register machine $M = (n, P, m)$ that computes f, as follows:

- $H = \{s\}$ is the label of the only membrane (the skin) of Π_M;
- $\mu = [\,]_s$ is the membrane structure;
- $Var_1 = \{r_1, \ldots, r_n\} \cup \{s_1, \ldots, s_n\} \cup \{t_1, \ldots, t_n\} \cup \{p\} \cup \{p_0, \ldots, p_m\} \cup \{q_0, \ldots, q_m\} \cup \{z_{j,1}, z_{j,2}, z_{j,3}$ for all $0 \leq j < m\}$;
- $Pr_1 = \{3t_i \to 2|r_i + 1|s_i$ for all $1 \leq i \leq \alpha\} \cup \{2p \to 1|p_0 + 1|q_0\} \cup \{z_{j,1} + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_k + 1|q_k,\ z_{j,1} + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_l + 1|q_l,\ z_{j,2} - 1|_{p_j} \to 1|q_j,\ z_{j,3} - 1|_{q_j} \to 1|p_j$ for all instructions $j : (\text{INC}(i), k, l) \in P\} \cup \{z_{j,1} - 3|_{p_j} \to 2|r_i + 1|s_i,\ r_i + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_l + 1|q_l,\ 2p_j|_{r_i} \to 1|p_j + 1|p_k,\ 2q_j|_{r_i} \to 1|q_j + 1|q_k,\ z_{j,2} - 1|_{p_j} \to 1|q_j,\ z_{j,3} - 1|_{q_j} \to 1|p_j\}$ for all instructions $j : (\text{DEC}(i), k, l) \in P\}$;
- $Var_1(0)$ is the vector of initial values of the variables of Var_1, obtained by putting:
  - $t_i = x_i$ (the input values of f) for all $1 \leq i \leq \alpha$;
  - $t_i = 0$ for all $\alpha + 1 \leq i \leq n$;

- $r_i = s_i = 0$ for all $1 \leq i \leq n$;
- $p = 1$;
- $p_j = q_j = 0$ for all $0 \leq j \leq m$;
- $z_{j,1} = z_{j,2} = z_{j,3} = 0$ for all $0 \leq j < m$.

The input values $x_1, \ldots, x_\alpha$ of f are introduced into the P system as the initial values of variables $t_1, \ldots, t_\alpha$. Moreover, the value of variable p is set to 1. In the first step of its computation, the P system will copy the values of $t_1, \ldots, t_\alpha$ to $s_1, \ldots, s_\alpha$, and the double of these values to variables $r_1, \ldots, r_\alpha$. So doing, after the simulation of each instruction of M variables $s_1, \ldots, s_n$ will contain the values of the registers of M, while $r_1, \ldots, r_n$ will contain their doubles. While making these copies, the value of variable p is copied to both p_0 and q_0, in order to start the simulation of M. The simulation proceeds much like in the way described in the proof of Theorem 6; the programs there illustrated are here modified in order to deal with the new variables. If and when the simulation reaches a final configuration, variables $s_1, \ldots, s_\beta$ contain the result of the computation.

The initialization step is performed by executing the following programs:

$$3t_i \to 2|r_i + 1|s_i \quad \text{for all } 1 \leq i \leq \alpha$$
$$2p \to 1|p_0 + 1|q_0$$

Each increment instruction $j : (\text{INC}(i), k, l)$ of M is simulated in one step by the execution of the following programs:

$$z_{j,1} + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_k + 1|q_k \tag{26}$$
$$z_{j,1} + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_l + 1|q_l \tag{27}$$
$$z_{j,2} - 1|_{p_j} \to 1|q_j \tag{28}$$
$$z_{j,3} - 1|_{q_j} \to 1|p_j \tag{29}$$

The simulation is analogous to the one described in the proof of Theorem 6, with the difference that when adding 2 to r_i the system now also increments s_i; to do so, the production value computed by the first two programs must be 5 instead of 4. Once again, if the machine M to be simulated is deterministic then program (27) disappears and the simulation itself becomes deterministic.

Each decrement instruction $j : (\text{DEC}(i), k, l)$ is simulated in one step by the execution of the following programs:

$$z_{j,1} - 3|_{p_j} \to 2|r_i + 1|s_i \tag{30}$$
$$r_i + 5|_{p_j} \to 2|r_i + 1|s_i + 1|p_l + 1|q_l \tag{31}$$
$$2p_j|_{r_i} \to 1|p_j + 1|p_k \tag{32}$$
$$2q_j|_{r_i} \to 1|q_j + 1|q_k \tag{33}$$
$$z_{j,2} - 1|_{p_j} \to 1|q_j \tag{34}$$
$$z_{j,3} - 1|_{q_j} \to 1|p_j \tag{35}$$

The simulation is analogous to the one described in the proof of Theorem 6, with the only difference that when subtracting or adding 2 to r_i by programs (30) and (31), respectively, the system now also decrements or increments s_i, respectively.

It can be easily checked that the simulation is correct, and that after simulating each instruction of M the values of variable s_i (resp., r_i) is equal to the contents (resp., the double of the contents) of register i, for $1 \leq i \leq n$. If and when the program counter of M reaches the value m, the corresponding variables p_m and q_m assume value 1 and the computation reaches a final configuration; the result of the computation can then be recovered from variables $s_1, \ldots, s_\beta$. □

By taking $\beta = 0$ and $\alpha \geq 1$ in the previous proof, we obtain the following result concerning the accepting variant of EN P systems working in the one-parallel mode.

Corollary 4. *For any $L \in \mathbf{Ps}(\alpha)\mathbf{RE}$ there exists a one-membrane EN P system, having linear production functions each depending upon at most one variable, that accepts L by working in the one-parallel mode.*

On the other hand, by taking $\alpha = 0$ and $\beta \geq 1$ we get the following characterization of $\mathbf{Ps}(\beta)\mathbf{RE}$ by the generating variant of EN P systems working in the one-parallel mode.

Corollary 5. *For any $L \in \mathbf{Ps}(\beta)\mathbf{RE}$ there exists a one-membrane (nondeterministic) EN P system, having linear production functions each depending upon at most one variable, that generates L by working in the one-parallel mode.*

By putting $\alpha = 1$ and $\beta = 0$ in Corollary 4, and $\alpha = 0$ and $\beta = 1$ in Corollary 5, we obtain the following characterization:

$$\mathbb{N}\mathbf{RE} = \mathbf{ENP}_1(poly^1(1), oneP)$$

Moreover, it can be easily checked that when the register machine M simulated in Theorem 7 and in Corollary 4 is deterministic, the simulating EN P system Π_M works in the all-parallel mode. This means that the above construction leads to a further characterization of $\mathbb{N}\mathbf{RE}$ by all-parallel recognizing EN P systems having linear production functions of one variable, alternative to the one obtained by Theorem 2.

Another consequence of Theorem 7 is that there exists a further small universal one-parallel deterministic EN P system, as stated in the following theorem.

Theorem 8. *There exists a universal one-parallel deterministic EN P system of degree* 1*, having* 137 *variables and* 108 *programs.*

Proof. The system mentioned in the statement simulates the small universal deterministic register machine M_u reported in Figure 1, and is built according to the description given in the proof of Theorem 7, as we have done in the proofs of Theorems 3 and 5. The number of increment and decrement instructions of M_u are 9 and 13, respectively, and each of them is translated to 3 and 6 programs

of the small universal EN P system, respectively. The initialization step requires further $\alpha+1=3$ programs, since M_u is fed with two input values: the "code" of f and its input. We thus obtain a total of 108 programs. As for variables, $8 \cdot 3 = 24$ are used to simulate the registers of M_u, and 46 are used to denote the value of its program counter; moreover, there are 3 auxiliary variables for each instruction of M, and one variable (p) which used to trigger the start of the simulation, for a total of 137 variables. □

Since the universal register machine M_u simulated in Theorem 8 is deterministic, the simulating small EN P system is deterministic too, and works both in the all-parallel as well as in the one-parallel mode. By comparing the number of variables and programs in all "small" EN P systems described in this paper, we see that the smallest is the one described in Theorem 3, containing only 31 variables and 61 programs. However such a small EN P system is not able to work in the one-parallel mode, hence in case we are forced to do so we must resort to one of the others described in this paper; the choice will depend upon the parameter (number of variables or number of programs) we want to minimize, as well as whether we are willing to work with even inputs and outputs. It is left as an open problem to prove that these are the smallest possible universal EN P systems, or finding instead smaller ones. Designing sets of programs that simulate consecutive INC and DEC instructions of M_u, as it has already been done in [9] and several other times in the literature, could be a hint for finding smaller systems.

4 Producing Output in Separate Variables

In all EN P systems described above, the output is considered to be the value of some specified variables in the final configuration, if and when this is reached. This is different from how EN P systems produce their output in most existing papers: usually, some separate output variables are considered, and the output of the system is defined as the set of all values assumed by these variables during the entire computation. In this section we prove that each of our EN P systems can be easily modified in order to produce its output according to this latter way.

Theorem 9. *The EN P systems used in Theorems 2, 4, 6 and 7 can be modified so that their output is produced into separate variables.*

Proof. Let $\Pi_M = (1, H, \mu, (Var_1, Pr_1, Var_1(0))$ be one of the EN P systems mentioned in the statement, simulating a register machine M computing the partial recursive function $f : \mathbb{N}^\alpha \to \mathbb{N}^\beta$. Let $x_1, \ldots, x_\beta$ denote the output variables of Π_M, that is, variables $r_1, \ldots, r_\beta$ for Theorems 2, 4, 6 and variables $s_1, \ldots, s_\beta$ in Theorem 7. Note that, by construction, these variables contain the value of f if and when a final configuration is reached, and this happens if and only if p_m (the variable indicating label m of the program of M) assumes value 1.

We modify Π_M by introducing the following new variables:

- $\{y_1, \ldots, y_\beta\}$, whose values are kept identical to $x_1, \ldots, x_\beta$ until p_m becomes 1 (if this happens);
- $\{z_1, \ldots, z_\beta\}$, as the new output variables;
- $\{u_1, \ldots, u_\beta\}$, as flags;

and programs:

$$\beta p_m \to 1|u_1 + \ldots + 1|u_\beta \tag{36}$$

$$u_i \to 1|y_i \qquad \text{for all } 1 \leq i \leq \beta \tag{37}$$

$$x_i|_{y_i} \to z_i \qquad \text{for all } 1 \leq i \leq \beta \tag{38}$$

Moreover, each program already present in Π_M that changes the value of an output variable x_i is modified in order to also apply the same change to the new variable y_i, as done in the proof of Theorem 7. So doing, after simulating each instruction of M the values of variables x_i and y_i will be the same for all $1 \leq i \leq \beta$. Since $y_1, \ldots, y_\beta$ never appear in the production functions of these modified programs, no change is caused to the behavior of Π_M.

All new variables are initialized to zero before the computation starts. During the first computation step variables $y_1, \ldots, y_\alpha$ are initialized to the values of $x_1, \ldots, x_\alpha$, as in the initialization step of Theorem 7. The computation then proceeds as prescribed by the programs of Π_M. If and when the computation reaches a final configuration then program (36) is executed, with the effect of zeroing p_m and setting $u_1, \ldots, u_\beta$ to 1. When this happens, by programs (37) the values of $y_1, \ldots, y_\beta$ are incremented, thus becoming larger than the values of $x_1, \ldots, x_\beta$. This means that programs (38) can now be applied, with the effect of copying the values of the original output variables $x_1, \ldots, x_\beta$ to the new output variables $z_1, \ldots, z_\beta$.

On the other hand, note that before and after reaching a final configuration of Π_M the value of variables $z_1, \ldots, z_\beta$ is never affected. In fact, when $p_m = 0$ program (36) has no effect, since it distributes a contribution of 0 to $u_1, \ldots, u_\beta$, leaving their value unaltered. This happens both before p_m becomes 1, and after executing program (36). Programs (37) increment the values of $y_1, \ldots, y_\beta$ only once, when $u_1 = \ldots = u_\beta = 1$, otherwise they produce no effect. Finally, programs (38) are first executed as soon as the values of $y_1, \ldots, y_\beta$ become larger than that of $x_1, \ldots, x_\beta$, after which they distribute a contribution of zero to $z_1, \ldots, z_\beta$.

So the only value assumed by the new output variables $z_1, \ldots, z_\beta$, besides zero, is the output value of M. □

5 Conclusions and Directions for Further Work

In this paper we have studied the computational power of enzymatic numerical P systems working in the all-parallel and one-parallel modes.

We have improved some previously known universality results, in terms of number of membranes and number of variables used in the production functions.

So, by using a flattening technique, we have first shown that every EN P system working either in the all-parallel or in the one-parallel mode can be simulated by an equivalent one-membrane EN P system working in the same mode. Then we have shown that linear production functions, each depending upon at most one variable, suffice to reach universality for both computing modes. As a byproduct we have obtained several small universal deterministic EN P systems, the smallest one having only 31 variables and 61 programs.

It is left open whether smaller universal EN P systems exist. It is also left open whether the known universality result on *sequential* EN P systems contained in [17] — a characterization of $\mathbb{N}\mathbf{RE}$ by sequential EN P systems of degree 7, whose production functions are polynomials of degree at most 5, each depending upon at most 5 variables — can be improved.

Acknowledgements

The ideas exposed in this paper emerged during the *Eleventh Brainstorming Week on Membrane Computing (BWMC 2013)*, held in Seville from February 4$^{\text{th}}$ to February 8$^{\text{th}}$, 2013.

This research was partially funded by Lombardy Region under project NEDD.

References

1. P. Bottoni, C. Martín-Vide, Gh. Păun, G. Rozenberg: Membrane systems with promoters/inhibitors. *Acta Informatica* 38(10):695–720, 2002.
2. C. Buiu, C.I. Vasile, O. Arsene: Development of membrane controllers for mobile robots. *Information Sciences* 187:33–51, 2012.
3. R. Freund, M. Oswald: GP systems with forbidding context. *Fundamenta Informaticae* 49(1-3):81–102, 2002.
4. R. Freund, Gh. Păun: On the number of non-terminals in graph-controlled, programmed, and matrix grammars. In: M. Margenstern, Y. Rogozhin (Eds.), *Universal Machines and Computations*, Chişinău, 2001, LNCS 2055, Springer-Verlag, 2001, pp. 214–225.
5. R. Freund, Gh. Păun: From regulated rewriting to computing with membranes: collapsing hierarchies. *Theoretical Computer Science* 312:143–189, 2004.
6. I. Korec: Small universal register machines. *Theoretical Computer Science* 168:267–301, 1996.
7. Y. Matijasevitch: *Hilbert's tenth problem*. MIT Press, Cambridge, London, 1993.
8. M.L. Minsky: *Computation. finite and infinite machines*. Prentice Hall, Englewood Cliffs, New Jersey, 1967.
9. A. Păun, Gh. Păun: Small universal spiking neural P systems. *Biosystems* 90(1):48–60, 2007.
10. Gh. Păun, R. Păun: Membrane computing and economics: numerical P systems. *Fundamenta Informaticae* 73:213–227, 2006.
11. Gh. Păun, G. Rozenberg, A. Salomaa (eds.): *The Oxford handbook of membrane computing*. Oxford University Press, 2010.

12. A.B. Pavel: *Membrane controllers for cognitive robots*. Master's thesis, Department of Automatic Control and System Engineering, Politechnica University of Bucharest, Romania, 2011.
13. A.B. Pavel, O. Arsene, C. Buiu: Enzymatic numerical P systems – A new class of membrane computing systems. *IEEE Fifth International Conference on Bio-Inspired Computing: Theory and Applications (BIC-TA)*, IEEE, 2010, pp. 1331–1336.
14. A.B. Pavel, C. Buiu: Using enzymatic numerical P systems for modeling mobile robot controllers. *Natural Computing* 11(3):387–393, 2012.
15. A.B Pavel, C.I. Vasile, I. Dumitrache: Robot localization implemented with enzymatic numerical P systems. In: T.J. Prescott et al. (Eds.), Proceedings of *Living Machines 2012*, Barcelona, Spain, July 9–12, 2012, LNAI 7375, Springer-Verlag, 2012, pp. 204–215.
16. C.I. Vasile, A.B. Pavel, I. Dumitrache: Universality of enzymatic numerical P systems. *International Journal of Computer Mathematics*, 2013. DOI: 10.1080/00207160.2012.748897
17. C.I. Vasile, A.B. Pavel, I. Dumitrache, Gh. Păun: On the power of enzymatic numerical P systems. *Acta Informatica* 49(6):395–412, 2012.

Simulating a Family of Tissue P Systems Solving SAT on the GPU

Miguel A. Martínez-del-Amor, Jesús Pérez-Carrasco, Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
University of Seville
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: mdelamor@us.es, jesusperezcarrasco@gmail.com, marper@us.es

Summary. In order to provide efficient software tools to deal with large membrane systems, high-throughput simulators are required. Parallel computing platforms are good candidates, since they are capable of partially implementing the inherently parallel nature of the model. In this concern, today GPUs (Graphics Processing Unit) are considered as highly parallel processors, and they are being consolidated as accelerators for scientific applications. In fact, previous attempts to design P systems simulators on GPUs have shown that a parallel architecture is better suited in performance than traditional single CPUs.

In 2010, a GPU-based simulator was introduced for a family of P systems with active membranes solving `SAT` in linear time. This is the starting point of this paper, which presents a new GPU simulator for another polynomial-time solution to `SAT` by means of tissue P systems with cell division, trading space for time. The aim of this simulator is to further study which ingredients of different P systems models are well suited to be managed by the GPU.

Keywords: Membrane Computing, tissue P systems, SAT, GPU Computing

1 Introduction

Membrane Computing [15] is a recent branch of *Natural Computing*, which defines massively parallel and non-deterministic computing devices abstracted from living *eukaryotic cells*. These devices are called membrane systems, or simply, *P systems* (named after its creator, Gheorghe Păun) [12]. Today, researchers have only one method to work with P systems, which is the usage of simulators running on conventional electronic computers. However, these simulators are normally unfit for very large P systems models. The main reason is that they are not throughput-oriented, so they consume large amounts of time and memory resources on a computer. Therefore, the necessity of efficient simulators arises [15].

In the last years, the trend has been oriented to implement P systems parallelism on parallel platforms, such as *accelerators* (special parallel devices). In fact, *the advent of the accelerators in High Performance Computing offers fresh avenues for developing new and efficient simulators* [2]. One of the most important accelerators nowadays is the *GPU (Graphics Processing Unit)*. It is the core of graphics cards and, thanks to the fast growth of video and game market, typically contains hundreds of slight processors. Their evolution has also led to a new programming model based on data parallelism. This permits to use GPUs for general purpose applications (*GPGPU* or *GPU computing*) [8].

So far, many GPU-based simulators have been developed for several P systems models: *active membranes* [2], *PDP systems* [9], *Spiking Neural P systems* [1], among others. These simulators are flexible for the corresponding P system model, supporting a wide variety of P systems. However, this feature causes a negative effect on performance [10]. An alternative line was initiated in 2010 with the introduction of a specific (*ad-hoc*) simulator for a P system based efficient solution to `SAT` by using GPUs [3, 4]. The solution is based on P systems with active membranes, and the simulator achieves speedups of up to 90x, compared to the CPU counterpart. The obtained results lead to a new open question, related with the efficiency of P system simulators: fixed a problem (e.g. `SAT`), which is the fastest P system based solution simulated on the GPU? In order to answer this question, we first need to analyze which elements of P systems are better suited to be handled by the GPU. In fact, this can help to define new methods to design more efficient simulators.

In this paper, we consider another efficient solution to `SAT` based on *tissue P systems with cell division*. A simulator based on GPUs for this solution is presented. We provide an analysis of performance of the new simulator, together with a performance comparison between the cell-like and the tissue-like simulators.

The paper is structured as follows: Section 2 introduces the model of tissue P systems with cell division and the solution to `SAT`; Section 3 surveys the typical GPU architecture and the peculiarities of GPU computing; Section 4 depicts the design of the new simulator; Section 5 provides the performance analysis of the developed simulator and the mentioned comparisons; and finally, Section 6 ends the paper with conclusions and open research lines.

2 Tissue-like P systems

In this paper, we work on computational devices inspired by the cell inter–communication in tissues, and adding the ingredient of cell division rules. Cell division is an elegant process that enables organisms to grow and reproduce by means of the production of two daughter cells from a single parent cell.

Tissue P systems with cell division are inspired by the cell-like model of P systems with active membranes [13]. In these models, the cells are not polarized; cells obtained by division have the same labels as the original cell, and if a cell is

divided, its interaction with other cells or with the environment is locked during the division process.

First, we recall some preliminaries. An *alphabet* Γ is a non–empty set whose elements are called *symbols*. A *multiset* m over an alphabet Γ is a pair $m = (\Gamma, f)$ where f is a mapping from Γ into $\mathbb{N}$. If $m = (\Gamma, f)$ is a multiset then its *support* is defined as $supp(m) = \{x \in \Gamma \mid f(x) > 0\}$. A multiset is finite if its support is a finite set. If $supp(m) = \{a_1, \ldots, a_k\}$ then the multiset m will be denoted as $m = a_1^{f(a_1)} \ldots a_k^{f(a_k)}$ (here the order is irrelevant), and we say that $f(a_1) + \cdots + f(a_k)$ is the cardinal of m, denoted by $|m|$. The empty multiset is denoted by λ.

Let $m_1 = (\Gamma, f_1)$ and $m_2 = (\Gamma, f_2)$ multisets over Γ. The *union* of m_1 and m_2, denoted by $m_1 + m_2$, is the multiset (Γ, g), where $g = f_1 + f_2$, that is, $g(x) = f_1(x) + f_2(x)$ for each $x \in \Gamma$. The *relative complement* of m_2 in m_1, denoted by $m_1 \setminus m_2$ is the multiset (Γ, g), where $g(x) = f_1(x) - f_2(x)$ if $f_1(x) \geq f_2(x)$ and $g(x) = 0$ otherwise.

Definition 1. *A tissue P system with cell division of degree $q \geq 1$ is a tuple $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$, where:*

1. *Γ, Σ and $\mathcal{E}$ are finite alphabets such that $\Sigma \subsetneq \Gamma$ and $\mathcal{E} \subseteq \Gamma$.*
2. *Γ has two distinguished objects* `yes` *and* `no`*, and $\{\texttt{yes}, \texttt{no}\} \cap \mathcal{E} = \emptyset$.*
3. *$\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over $\Gamma \setminus \Sigma$.*
4. *At least one copy of objects* `yes` *and* `no` *are present in some initial multisets $\mathcal{M}_1, \ldots, \mathcal{M}_q$, but none of them are present in $\mathcal{E}$.*
5. *$\mathcal{R}$ is a finite set of rules of the following forms:*
   (a) Communication rules*: $(i, u/v, j)$, for $i, j \in \{0, 1, 2, \ldots, q\}, i \neq j$, u, v finite multisets over Γ, and $|u + v| \neq 0$;*
   (b) Division rules*: $[a]_i \rightarrow [b]_i[c]_i$, where $i \in \{1, 2, \ldots, q\}$, $i \neq i_{out}$ and $a, b, c \in \Gamma$.*
6. *$i_{in} \in \{1, 2, \ldots, q\}$, and $i_{out} \in \{0, 1, \ldots, q\}$.*

A *tissue P system with cell division* $\Pi = (\Gamma, \Sigma, \mathcal{E}, \mathcal{M}_1, \ldots, \mathcal{M}_q, \mathcal{R}, i_{in}, i_{out})$ of degree $q \geq 1$ can be viewed as a set of q cells, labeled by $1, \ldots, q$, with an environment labeled by 0 such that: (a) $\mathcal{M}_1, \ldots, \mathcal{M}_q$ are finite multisets over Γ representing the objects (elements in Γ) initially placed in the q cells of the system; (b) $\mathcal{E}$ is the set of objects located initially in the environment of the system, all of them appearing in an *arbitrary number of copies*; and (c) i_{in} represents the input cell, and $i_{out} \in \{0, 1, \ldots, q\}$ represents the *region* (a distinguished cell when $i_{out} \in \{1, \ldots, q\}$, or the environment when $i_{out} = 0$) which will encode the output of the system.

When applying a rule $(i, u/v, j)$, the objects of the multiset u are sent from region i to region j and, simultaneously, the objects of multiset v are sent from region j to region i. When applying a division rule $[a]_i \rightarrow [b]_i[c]_i$, under the influence of object a, the cell with label i is divided into two cells with the same label. In the first copy, object a is replaced by object b, and in the second one,

object a is replaced by object c; all the other objects are replicated and copies of them are placed in the two new cells. The output cell i_{out} cannot be divided.

The rules of a tissue P system with cell division are applied in a non-deterministic maximally parallel manner. At each step, all the cells which can evolve must evolve in a maximally parallel way (at each step, we apply a multiset of rules which is maximal, no further rule can be added), with the following important remark: if a cell divides, only the division rule is applied to that cell at that step; the objects inside that cell do not evolve by means of communication rules.

A *configuration* at any instant of Π is described by all multisets of objects over Γ associated with all the cells present in the system, and the multiset of objects over $\Gamma \setminus \mathcal{E}$ associated with the environment at that moment. Given a finite multiset m over Σ, the *initial configuration* with input m is $(\mathcal{M}_1, \ldots, \mathcal{M}_{i_{in}} + m, \ldots, \mathcal{M}_q; \emptyset)$. A configuration is a *halting configuration* if no rule of the system is applicable to it.

We say that configuration $\mathcal{C}_1$ yields configuration $\mathcal{C}_2$ in one *transition step* if we can pass from $\mathcal{C}_1$ to $\mathcal{C}_2$ by applying the rules from $\mathcal{R}$ following the previous remarks. A *computation* of Π is a sequence of configurations such that: (a) the first term of the sequence is an initial configuration of the system; (b) each remaining term of the sequence is obtained from the previous one by applying the rules of the system in a maximally parallel manner with the restrictions previously mentioned; and (c) if the sequence is finite (called *halting computation*), then the last term of the sequence is a halting configuration.

A tissue P system with cell division is a *recognizer system* if all computations halt, and if $\mathcal{C}$ is a computation of Π, then either object `yes` or object `no` (but not both) must have been released into the environment, and only at the last step of the computation. We say that $\mathcal{C}$ is an *accepting* (respectively, *rejecting*) *computation* if object `yes` (respectively, object `no`) appears in the environment associated with the corresponding halting configuration of $\mathcal{C}$.

2.1 An efficient solution to SAT by means of tissue P systems with cell division

This section presents an efficient solution to the `SAT` problem by means of family of recognizer tissue P systems with cell division (see [14] for details).

For each pair of natural numbers $m, n \in \mathbf{N}$, we will consider the recognizer tissue P system with cellular division $\Pi(\langle m, n \rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, R, 2)$ of degree 2, defined as follows:

- The input alphabet is $\Sigma = \{x_{i,j}, \overline{x}_{i,j} : \ 1 \leq i \leq n, \ 1 \leq j \leq m\}$
- The working alphabet is

$$\begin{aligned}\Gamma = \Sigma \cup \{a_i, t_i, f_i \mid 1 \leq i \leq n\} \cup \{r_i \mid 1 \leq i \leq m\} \cup \\ \cup \{T_i, F_i \mid 1 \leq i \leq n\} \cup \{T_{i,j}, F_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq m+1\} \cup \\ \cup \{b_i \mid 1 \leq i \leq 2n+m+1\} \cup \{c_i \mid 1 \leq i \leq n+1\} \cup \\ \cup \{d_i \mid 1 \leq i \leq 2n+2m+nm+1\} \cup \\ \cup \{e_i \mid 1 \leq i \leq 2n+2m+nm+3\} \ \cup \ \{f, g, \texttt{yes}, \texttt{no}\}\end{aligned}$$

- The environment alphabet is $\mathcal{E} = \Gamma - \{\texttt{yes}, \texttt{no}\}$.
- The set of labels is $\{1, 2\}$.
- The initial multisets associated with the cells are $\mathcal{M}_1 = \{\texttt{yes}, \texttt{no}, b_1, c_1, d_1, e_1\}$ and $\mathcal{M}_2 = \{f, g, a_1, a_2, \ldots, a_n\}$.
- The input cell is the one labeled by 2, and the output region is the environment.
- The set $\mathcal{R}$ is formed by the following rules:

1. **Division rule:**

   (a) $[\ a_i\ ]_2 \rightarrow [\ T_i\ ]_2[\ F_i\ ]_2$, for $i = 1, 2, \ldots, n$.

2. **Communication rules:**

   (b) $(1, b_i/b_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (c) $(1, c_i/c_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (d) $(1, d_i/d_{i+1}^2, 0)$, for $i = 1, \ldots, n$.
   (e) $(1, e_i/e_{i+1}, 0)$, for $i = 1, \ldots, 2n + 2m + nm + 2$.
   (f) $(1, b_{n+1}c_{n+1}/f, 2)$.
   (g) $(1, d_{n+1}/g, 2)$.
   (h^*) $(1, f^2/f, 0)$.
   (h) $(2, c_{n+1}T_i/c_{n+1}\ T_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (i) $(2, c_{n+1}F_i/c_{n+1}\ F_{i,1}, 0)$, for $i = 1, \ldots, n$.
   (j) $(2, T_{i,j}/t_i\ T_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (k) $(2, F_{i,j}/f_i\ F_{i,j+1}, 0)$, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.
   (l) $(2, b_i/b_{i+1}, 0)$.
   (m) $(2, d_i/d_{i+1}, 0)$, for $i = n + 1, \ldots, 2n + m$.
   (n) $(2, b_{2n+m+1}\ t_i\ x_{i,j}/b_{2n+m+1}\ r_j, 0)$.
   (o) $(2, b_{2n+m+1}\ f_i\ \overline{x}_{i,j}/b_{2n+m+1}\ r_j, 0)$, for $1 \leq i \leq n$ and $1 \leq j \leq m$.
   (p) $(2, d_i/d_{i+1}, 0)$, for $i = 2n + m + 1, \ldots, 2n + m + nm$.
   (q) $(2, d_{2n+m+nm+j}\ r_j/d_{2n+m+nm+j+1}, 0)$, for $j = 1, \ldots, m$.
   (r) $(2, d_{2n+2m+nm+1}/f\ \texttt{yes}, 1)$.
   (s) $(2, \texttt{yes}/\lambda, 0)$.
   (t) $(1, e_{2n+2m+nm+3}\ f\ \texttt{no}/\lambda, 0)$.

Let $\varphi = C_1 \wedge \cdots \wedge C_m$ be a propositional formula in **CNF**[1] such that the set of variables of the formula is $Var(\varphi) = \{x_1, \ldots, x_n\}$, and consists of m *clauses* $C_j = y_{j,1} \vee \cdots \vee y_{j,k_j}$, $1 \leq i \leq m$, where $y_{j,j'} \in \{x_i, \neg x_i : 1 \leq i \leq n\}$ are the literals of φ. Without loss of generality, we can assume that the formula is in simplified expression.

Next, we consider a polynomial encoding (cod, s) of the SAT problem in the family $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$. The function cod associates to the previously described propositional formula φ, that is an instance of SAT with parameters n (number of variables) and m (number of clauses), the following multiset of objects

[1] Conjunctive Normal Form

$$cod(\varphi) = \bigcup_{i=1}^{m} \{x_{i,j} : x_i \in C_j\} \cup \{\overline{x}_{i,j} : \neg x_i \in C_j\}$$

In this case, object $x_{i,j}$ represents that variable x_i belongs to clause C_j.

The *size* function, s, is defined as follows $s(\varphi) = \langle m, n \rangle = \frac{(m+n)\cdot(m+n+1)}{2} + m$. The system of the family $\mathbf{\Pi}$ to process the instance φ will be the tissue P system $\Pi(s(\varphi))$ with input multiset $cod(\varphi)$.

The execution of the system $\Pi(s(\varphi))$ with input $cod(\varphi)$ is structured in six phases:

- *Valuations generation phase*: in this phase all the possible relevant truth valuations are generated for the set of variables of the formula $\{x_1, \ldots, x_n\}$. It is implemented by using division rules (a), whereby each object x_i produces two new cells, one having the object T_i, that codifies the true value of the variable x_i, y and the other having the object T_i, that codifies the false value of the variable x_i. Thus, 2^n cells are obtained in n computation steps. These cells are labeled by 2, and each one codifies each possible truth valuation of the set of variables $\{x_1, \ldots, x_n\}$. Meanwhile, the objects f, g are replicated in each created cell. This phase spends n computation steps.
- *Counters generation phase*: simultaneously, and using the rules (b), (c), (d) and (e), the counters b_i, c_i, d_i, e_i of the cell labeled by 1, are evolving such that in each computation step the number of objects in each one are doubling. Thereby, through this process and after n steps, we get 2^n copies of the objects b_{n+1}, c_{n+1}, and d_{n+1}. Objects $b's$ will be used to check which clauses are satisfied for each truth valuation. Objects $c's$ are used to obtain a sufficient number of copies of t_i, f_i (namely, m). Objects $d's$ will be used to check if there is at least one valuation satisfying all clauses. Finally, objects $e's$ will be used to produced, in its case, the object `no` at the end of the computation.
- *Checking preparation phase*: this phase aims at preparing the system for checking clauses. For this, at step $n + 1$ of the computation, and by the application of the rules (f) and (g), the counters b_{n+1}, c_{n+1}, d_{n+1} of the cell 1 is exchanged for the objects f and g of the 2^n cells 2. Thus, after this step, each cell labeled by two has a copy of the objects b_{n+1}, c_{n+1}, d_{n+1}, while the cell 1 has 2 copies of the objects f and g.

  Subsequently, the presence of an object c_{n+1} in each one of the 2^n cells labeled by 2 allows to generate the objects $T_{i,1}$ and $F_{i,1}$. By the application of rules (j) and (k), these objects allows the emergence of m copies of t_i and m copies of f_i, according to the values of truth or falsity that a cell 2 assigns to a variable x_i. This process spends $n + m$ steps since there is only one object c_{n+1} in each cell 2 and, moreover, for each $i = 1, \ldots, n$, the rules (j) and (k) are applied exactly m consecutively times. Simultaneously, in the first steps of this process, the application of the rule (h^*) makes the cell labeled by 1 to appear only one copy of the object `yes`.

  Simultaneously in this phase, the counters b_i, d_i and e_i are evolving by the applications of the corresponding rules.

- *Checking clauses phase*: in this phase, the clauses that are true for every truth valuation are determined, and encoded by a cell labeled by 2. This phase starts at the computation step $(n+1)+(n+m)+1 = 2n+m+2$. Using the rules (n) and (o), the true clauses are checked for each valuation encoded by a cell., so that the appearance of an object r_j in a cell 2 means that the corresponding valuation makes true the clause C_j. Bearing in mind that a single copy of the object b_{2n+m+1} is in each cell, the phase takes nm computation steps.
  Thus, the configuration $\mathcal{C}_{2n+m+nm+1}$ is characterized by the following:
  - It contains exactly 2^n cells labeled by 2. Each one contains the object $d_{2n+m+nm+1}$, and copies of objects r_j for each clause C_j made true by the encoded valuation in the cell.
  - It contains a unique cell labeled by 1, containing a copy of objects `yes`, `no`, f, g and the counter $e_{2n+m+nm+2}$.

  This phase consumes m computation steps.
- *Formula checking phase*: in this phase it is determined if there exists any valuation making true the m clauses of the formula. For this, the rules of type (q) are used, analyzing in an ordered way (first the clause C_1, after that clause C_2, and so on) if the clauses of the formula are being satisfied by the represented valuation in the corresponding cell labeled by 2. For example, from counter $d_{2n+m+nm+1}$ appearing in every cell 2, the appearance of the object r_1 (the valuation makes true clause C_1) permits to generate in that cell the object $d_{2n+m+nm+2}$. This object, in turn, permits to evolve object $d_{2n+m+nm+3}$ if in that cell appears the object r_2. In this manner, a valuation represented by a cell labeled by 2 makes true the formula φ if and only if the object $d_{2n+m+nm+m+1}$ appears in the content of that cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$.
- *Output phase*: in this phase the system will provide the corresponding output, depending on the analysis in the formula checking phase.
  If the formula φ is satisfiable, then there is some cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ that contains an object $d_{2n+m+nm+m+1}$. In this case, the application of rule (r) sends an object f and the object `yes` to the cell 1. The object `yes` therefore disappears from cell 1, and consequently, rule (t) can not be applied. In the next computation step, the application of the rule (s) produces an object `yes` in the environment (for the first time during the whole computation) and the process ends.
  If the formula φ is not satisfiable, then there no exist any cell in the configuration $\mathcal{C}_{2n+m+nm+m+1}$ containing an object $d_{2n+m+nm+m+1}$. In this case, the rule (r) is not applicable, and in the next computation step, the counter e_i evolves, providing an object $e_{2n+m+nm+m+3}$ in the cell 1. This object permits the application of rule (t), since the objects `no` and f remains in the cell 1. In this way, the object `no` is sent in the next computation step, and the computation finalizes.

It can be easily proved that the family $\mathbf{\Pi} = \{\Pi(\langle m,n\rangle) : \ n,m \in \mathbf{N}\}$, defined above, is polynomially uniform by deterministic Turing machines. For this, it is enough to keep in mind that the systems of the family have benn defined through

recursive expressions, and the amount of resources needed to describe the system $\Pi(\langle m, n\rangle)$ is quadratic in $\max\{m, n\}$. Indeed:

1. Size of the alphabet: $6nm + 12n + 7m + 12 \in \Theta(nm)$.
2. Number of initial cells: $2 \in \Theta(1)$.
3. Number of initial objects: $n + 8 \in \Theta(n)$.
4. Number of rules: $4nm + 10n + 3m + 16 \in \Theta(nm)$.
5. Upper limit of rule length: $5 \in \Theta(1)$

3 GPU computing

The *GPU (Graphics Processor Unit)* is a specialized chip designed to manipulate computer graphics efficiently. In fact, it is an essential part of most current computers. Their highly parallel structure is based on hundreds of simple computing cores, making them more effective than common CPUs for processing large blocks of data in parallel [8]. Thus, the GPU is being consolidated as a device suitable for *High Performance Computing*, as it was foreseen by Elster [5] and other authors [8].

3.1 CUDA programming model

In 2007, NVIDIA announced *CUDA (Compute Unified Device Architecture)* [17], a programming model totally abstracted from the hardware of the GPU. Based on C, the programmer only has to think on threads and arrays, together with some performance aspects. This easy way to build large applications has led to a rapidly evolution of GPU computing [11]. As a result, CUDA has been successfully utilized for developing P systems simulators [1, 2, 3, 4, 9].

CUDA provides an heterogeneous computing system, consisting of a host (the CPU) and several devices (GPUs) [7]. The idea is to execute on device program sections with large amount of data parallelism. These sections are written in separated C functions called *kernel*. Each kernel is executed on the GPU by a grid of threads. As shown in Figure 1, threads are grouped in *blocks*. Threads belonging to the same block are easily synchronized by barrier operations (when a thread reaches the barrier, wait for the rest to continue).

The memory model is also an aspect to consider in the CUDA programming model. This memory hierarchy is explicitly and manually managed. The *global memory* is the largest (but slowest) in the GPU. It is accessed by the host, and also by any thread, as it is the communication channel between the host and the device. The smallest (but fastest) memory is the *shared memory*. It is local to each block, but the content is volatile through kernels calls, and the CPU cannot read it. Finally, there is a variety of atomic operations to update single data elements in any memory in a concurrently and synchronously way.

An efficient way to structure an algorithm in CUDA is by maximizing the usage of the memory hierarchy, as follows:

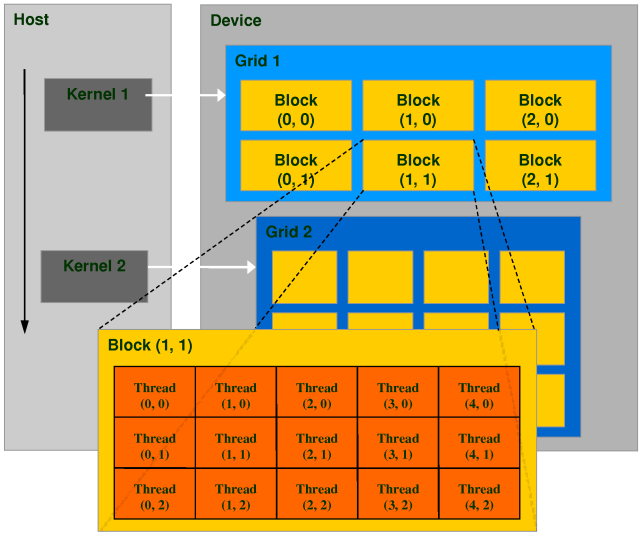


Fig. 1: Overview of CUDA programming model.

1. The threads of each block read its corresponding data portion from global memory to shared memory (which is inevitable because the host only can put the data in global memory).
2. Threads work with the data directly on shared memory.
3. Threads copy these data back to global memory (so the host can retrieve the result).

3.2 A modern GPU architecture

The GPU used in our work is the NVIDIA Tesla C1060. We use this model since it was used in our previous work, and the aim is to compare results. The Tesla C1060 is based on a scalable processor array which has 240 *SPs* (streaming-processor) cores organized as 30 *SMs* (streaming multiprocessor) and 4 GB of off-chip GDDR3 memory called device memory. The applications start at the host side which communicates with the device side through a bus, which is a PCI Express x16 bus standard.

The SM is the processing unit and an unified graphics and computing multiprocessor. Every SM contains the following units: eight SPs arithmetic cores, one double precision unit, an instruction cache, a read only constant cache, 16-Kbyte on-chip read/write shared memory, a set of 16384 32-bit registers, and access to the off-chip memory (device/local memory). The SM also has two SFUs that execute more complex floating point operations such as reciprocal square root,

sine or cosine with low latency. The arithmetic units are capable to execute three instructions per clock cycle, and they are fully pipelined, running at 1,296 GHz, yielding a peak theoretical 933 GFLOPS[2] (240 SP * 3 instructions * 1,296 GHz).

The local and global (device) memory spaces are not cached, which means that every memory access to global memory (or local memory) generates an explicit memory access. A multiprocessor takes 4 clock cycles to issue one memory instruction. Accessing local or global memory incurs an additional 400 to 600 clock cycles of memory latency [7], that is more expensive than accessing share memory and registers (only the mentioned 4 cycles).

A SM is a hardware device specifically designed with multithreaded capabilities. Each SM manages and executes up to 1024 threads in hardware with zero scheduling overhead. Each thread has its own thread execution state and can execute an independent code path. The SMs execute threads in a *SIMT (Single-Instruction Multiple-Thread)* fashion [7]. Basically, in the SIMT model all the threads execute the same instruction on different piece of data. SMs create, manage, schedule and execute threads in groups of 32 threads (which is the branching granularity of NVIDIA GPUs). This set of 32 threads is called *warp*. Each SM can handle up to 32 warps. Individual threads of the same warp must be of the same type and start together at the same program address, but they are free to branch and execute independently at cost of performance.

4 Parallel simulator on the GPU

In this section we describe the developed CUDA simulator. We first explain the data structures and the phases that compound the simulation algorithm. Secondly, the parallel simulator based on CUDA is depicted.

This simulation framework is named *TSPCUDASAT* and published under GNU GPLv3 license. It is enclosed to the software project *PMCGPU (Parallel simulators for Membrane Computing on the GPU)* [18], where the source code of the simulators is available for download.

4.1 Sequential simulation and data structures

For an easier implementation, the simulation algorithm has been divided into five (simulation) phases. Note that they are different in number than the denoted phases of the theoretical model (Section 4 and [14]). This is done to unify phases in the software design. Each of these simulation phases are implemented in code as separated functions whenever is possible. They correspond to the application of certain rules, as explained below:

- *Generation phase*: it performs the application of rules from (a) to (e) of the systems (Section 2.1). Therefore, it comprises the two first phases of the theoretical model: valuations generation phase and counters generation phase.

[2] FLOPS stands for *FLoating-point Operations Per Second*. GFLOPS are giga FLOPS.

- *Exchange phase*: it simulates the application of rules (f) and (g). It comprises the first part of the checking preparation phase.
- *Synchronization phase*: it applies the rules from (h) to (m), so comprising the second part of the checking preparation phase.
- *Checking phase*: it performs the application of rules from (n) to (p). Thus, it is the checking clauses phase we identified in the theoretical model.
- *Output phase*: it applies rules from (q) to (t). It then performs both the formula checking phase and the output phase identified in the theoretical model.

The sequential simulator implements these five simulation phases directly in source code, which is in C++. Each one works directly with the data structures depicted below. The input of the simulator is the same than the one used in the simulator for the cell-like solution [3, 4]. A DIMACS CNF file[3] is provided, and the simulator outputs the response of the computation. Therefore, it acts merely as a `SAT` solver, but the implementation follows the computation of the systems from the family of tissue P systems. Recall that the aim is not to provide a `SAT` solver, but to study P system simulations on the GPU by comparing different solutions to the same problem.

Furthermore, we have adopted a set of enhancements to improve the performance of the sequential simulator. After several tests, we have shown that the best optimizations are:

- As the Exchange phase is very simple, it is then implemented after the Generation phase loop, within the same function.
- We apply the full Synchronization phase to one cell before going to the next one. This allows us to exploit data locality in cache memories.
- In the Checking phase, we orderly insert the objects r_j, for $1 \leq j \leq m$, in the corresponding array whenever they are created. Thus, the Output phase can be easily performed, in such a way that it is not necessary to loop all the objects coming from the input multiset (literals). Now it is enough to check if there exists the m objects r_j.

4.2 Data structures

For this solution, the representation of a tissue P system $\Pi(\langle m, n\rangle)$ is twofold. As the model differentiates between cells labeled by 1 and 2, the design decision was to also have a different data structure representing each cell type in the system. The elements of the cells are encoded within 32-bit integers.

First, cell 1 is represented as an array having a constant dimension of 5 elements. That is, the multiset for cell 1 has the maximum amount of 5 objects: the three counters, b, c and d (which are initially in this cell), and the two objects *yes* and *no* (the final answer to the problem).

Second, the cells labeled by 2 are also represented by a one-dimensional array. All of them are stored inside this large array, since it is initially allocated to store

[3] One of the most adopted input formats by `SAT` solvers.

the maximum amount of cells (2^n). By studying the computation, we conclude that the maximum number of objects appearing in a cell 2 is $(2n) + 4 + |cod(\varphi)|$, where:

- $|cod(\varphi)|$ elements for the initial multiset,
- n elements for objects $T_{i,j}$ and $F_{i,j}$, for $1 \leq i \leq n$ and $1 \leq j \leq m$.
- n elements for objects t_i and f_i, for $1 \leq i \leq n$.
- 4 elements for counter objects a, b, c and d. They will be replaced for counter objects f and g.

The objects are represented similarly to the previous simulator for the cell-like based solution [3]. They are encoded at bit-level within integers of 32 bits, that store the following (8 bits for each field): the name of the object (x or $\overline{x}$), multiplicity of the object (as the multiplicity can exceed 2^8, this field can eventually be joined to the next one), variable (index i) and clause (index j).

1. The name of the object (x or $\overline{x}$)
2. Multiplicity of the object. As there are objects whose multiplicity can exceed 2^8, this field can eventually be joined to the next one (variable).
3. Variable (index i).
4. Clause (index j).

4.3 Design of the parallel simulator

The parallel simulator is designed to also fully reproduce a computation of the systems from the family of tissue P systems. That is, there is no a hybrid[4] solution providing simulation shortcuts to the computation as in [3]. The design of this parallel simulator is driven by the structure of phases explained above, using separate CUDA kernels to speedup the execution of each one.

A similar CUDA work distribution used in other simulators for cell-like solutions [2, 3] is applied. This general assignment is summarized in Figure 2. Each thread block corresponds to each cell labeled by 2 created in the system (up to 2^n cells). However, unlike the previous simulator for the cell-like solution, we do not assign a thread per literal. The assignment of each thread, this time, is different for each simulation phase. The work mapping per phase is therefore as follows:

- *Generation phase*: the number of thread blocks is iteratively increased together with the amount of cells created in each computation step. We distribute cells along the two-dimensional grid through successive kernel calls. Each thread block contains $(2n) + 4 + |cod(\varphi)|$ threads. That is, the amount of elements assigned to each cell in the global array storing multisets. Threads are then used to copy each individual elements of the corresponding cell when it is divided.

[4] A *hybrid simulator* does not perform exactly the same computational steps as the theoretical P system, but achieves the same answer.

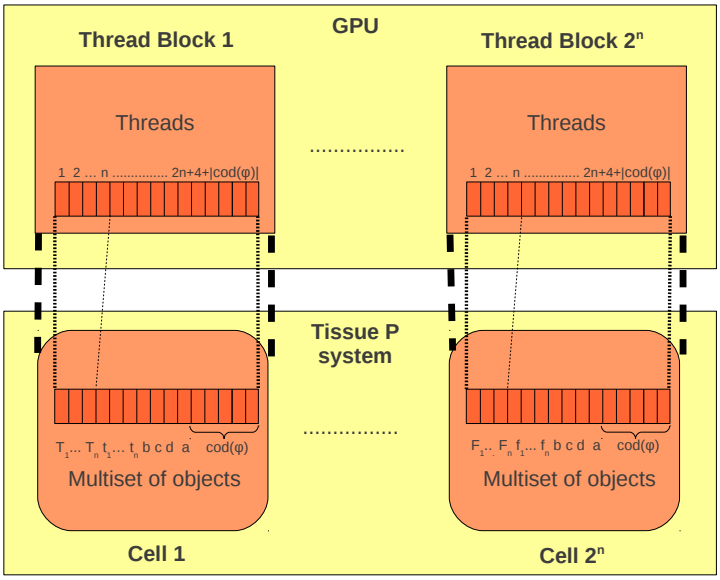


Fig. 2: General design of the parallel simulator.

- *Exchange phase*: it is executed at the kernel for Generation phase, using the same amount of thread blocks, but only the corresponding threads perform the exchange.
- *Synchronization phase*: the thread blocks are assigned to the cells labeled by 2, and the number of threads is n (number of variables). If we use the same amount of threads than in Generation phase, most of them will be idle: it is preferred to launch less threads, but performing effective work. We have experimentally corroborated this fact.
- *Checking phase*: the number of thread blocks is again assigned to be the number of cells labeled by 2. However, for this phase we use a block size of $|cod(\varphi)|$. That is, each thread is used to execute, in parallel, rules of type (n) and (o). The result at the `SAT` problem resolution level, each thread checks if the corresponding literal makes true its clause, depending on the truth assignment encoded by the cell assigned to the thread block.
- *Output phase*: rules of type (q) are sequentially executed in a separate kernel, again using $|cod(\varphi)|$ threads per block, and 2^n thread blocks.

For this solution, we have applied a small set of improvements, focused on the GPU implementation, to improve the performance of the parallel simulator. We have identified that the simulator runs twice faster than the non-enhanced simulator. We will use the enhanced version of the parallel simulator to perform the comparisons. These improvements are oriented to two performance aspects of GPU computing [11]:

1. The first enhancement type is to *emphasize the parallelism*, which aims to increase the number of threads per block (to the recommended amount from 64 to 1024).
2. The second enhancement type is to *exploit streaming bandwidth*. To do this, the data is first loaded into shared memory and operated there, avoiding global memory (expensive) accesses.

Next, we show the specific enhancement we have carried out for each phase:

- *Generation phase*: no enhancement were implemented here, since the implementation already satisfies the first optimization type. The second type will require a more sophisticated implementation, like the one presented in [4].
- *Exchange phase*: this phase is joined with the generation phase, but has no further enhancements.
- *Synchronization phase*: the two enhancement types are implemented here. The second enhancement type is carried out by using shared memory to avoid global memory accesses. The first type is performed by increasing the number of threads per block. For our simulator, we can assume that n (number of variables, and the number of threads per block) is a small number, since the number of cells grows exponentially with respect to it. For example, let be $n = 32$. Then, 2^{32} cells will be created, what require $2^{32}(68 + |cod(\varphi)|)$ bytes (in gigabytes: $272 + 4|cod(\varphi)|$). This number obviously exceeds the amount of available device memory. We therefore need to increase the number of threads per block, since $n < 32$ means to not fulfilling a CUDA warp. A solution here is to assign more than one cell to each thread block. This amount is $\frac{256}{n}$, being 256 the optimum number of threads per block. It allows us to reach a number of threads close to the optimum one. However, we have to take care also of having enough shared memory to load the data of every assigned cell.
- *Checking phase*: since $|cod(\varphi)|$ can be greater than 32, we then keep this number as the number of threads per block. However, we use shared memory to speedup the accesses to the elements of the array.
- *Output phase*: as in the previous phase, we also use shared memory, and the number of threads per block is kept to $|cod(\varphi)|$.

5 Performance analysis

In this section we show the performance tests carried out for the introduced simulator and for the cell-like based simulator [3]. All experiments are run on a Linux 64-bit server, with a 4-core (2 GHz) dual socket Intel i5 Xeon Nehalem processor, 12 GBytes of RAM, and two NVIDIA Tesla C1060 (240 cores at 1.30 GHz, 4 GBytes of memory).

We have developed two benchmarks (called *test 1* and *test 2*, respectively) to analyze the performance behavior of our simulators in two ways: increasing the number of threads per thread block, and increasing the number of thread blocks

per grid. They are the same than the two used for the cell-like simulators [3]. Both benchmarks have been generated by WinSAT program [16]. WinSAT is able to generate random `SAT` instances in DIMACS CNF format file by configuring several parameters: the number of variables (n), the number of clauses (m) and the number of literals per clause (we fix k for our experiments).

5.1 Tissue-like simulator

In this subsection, we will see the comparisons of performance between the two simulators developed for the family of tissue-like P systems under study: the sequential simulator (from now on, *tsp-sat-seq*), and the parallel simulator on the GPU (*tsp-sat-gpu*). For this analysis we will use one of the two tests mentioned above: the first one increasing the number of objects (fixing membranes to 2048), and the second increasing the number of variables (and so number of cells) and fixing the number of literals (and so input objects) to 256.

In the first case we can see that, even for small number of objects per membrane, *tsp-sat-gpu* runs faster than *tsp-sat-seq*. A different number of objects does not produce a great impact into the performance of the parallel simulator.

In the second case, we can observe that the kernels of *tsp-sat-gpu* runs faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cells 2 created by the system. For 64 membranes, the speedup is of 2x, but for 2 M cells it is of 8.3x.

Figure 3 shows the performance behavior of the tissue-like simulators for test 1. Only the time employed by kernels are considered for *tsp-sat-gpu*. We can see that, even for small number of objects per membrane, *tsp-sat-gpu* runs faster than *tsp-sat-seq*. A different number of objects does not produce a great impact into the performance of the parallel simulator. Note that in Section 4, we have introduced a different CUDA design for each phase. In this sense, the synchronization phase has been optimized to assign more cells to a thread block in order to increase the number of threads. However, the speedup is increased together with the number of objects per membrane. This means that the resources of the GPU are better utilized (e.g. 4 objects/threads does not fulfill a warp). We report the maximum speedup for 32 objects (a warp), which is of 11.6x. For 2 objects is 4x, and for 256, 6.1x.

Figure 4 shows the results for test 2, considering only kernel runtime for *tsp-sat-gpu*. For this case, we can observe that again, the kernels of *tsp-sat-gpu* runs faster than *tsp-sat-seq*. However, the performance gain is increased with the amount of cells 2 created by the system. For 64 membranes, the speedup is of 2x, but for 2 M cells it is of 8.3x.

Finally, Figure 5 shows the speedup achieved by the simulator *tsp-sat-gpu*, taking into account also the amount of time consumed by the data management (allocation and transfer). It is observed that, since the data management time is fixed for all the sizes, the speedup exceeds 1 only after 128 K cells. Systems with smaller number of cells are executed slower in the GPU because of the data

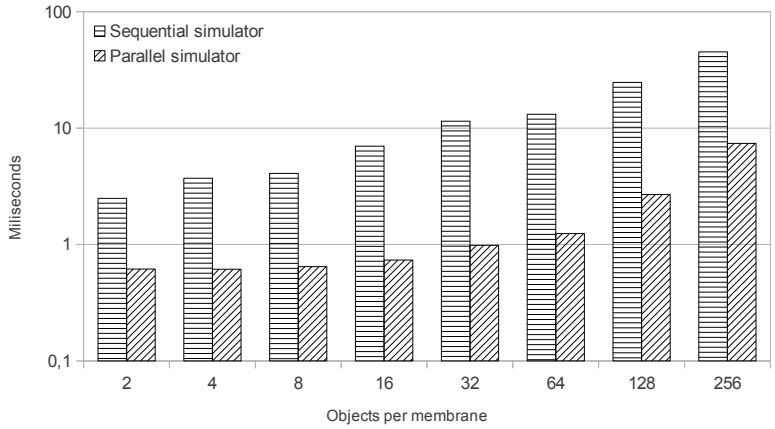


Fig. 3: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 1 (2048 membranes)

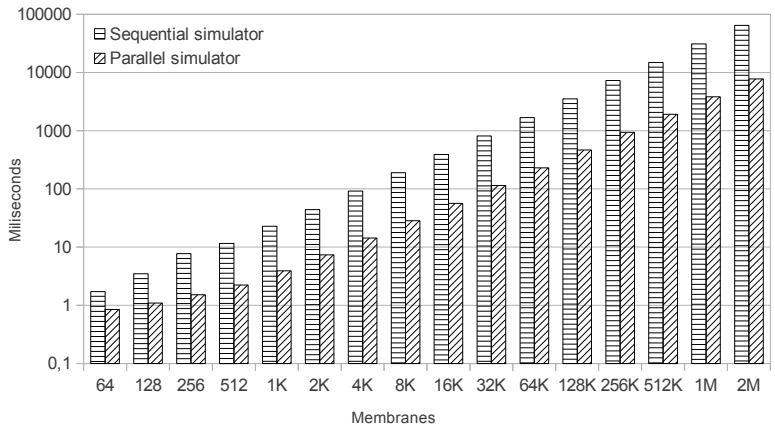


Fig. 4: Simulation performance for *tsp-sat-seq* and *tsp-sat-gpu*: Test 2 (256 Objects/Membrane)

management. However, for very large systems, the speedup is as large as only considering kernels. The maximum speedup is given for 4 M cells, up to 10x.

5.2 Cell-like vs tissue-like

Next, we compare the two simulators developed for the two solutions to `SAT` using P systems with active membranes (let call it *am-sat-gpu*) and tissue P systems with cell division (*tsp-sat-gpu*). Here we study which model is better suited to be simulated on the GPU.

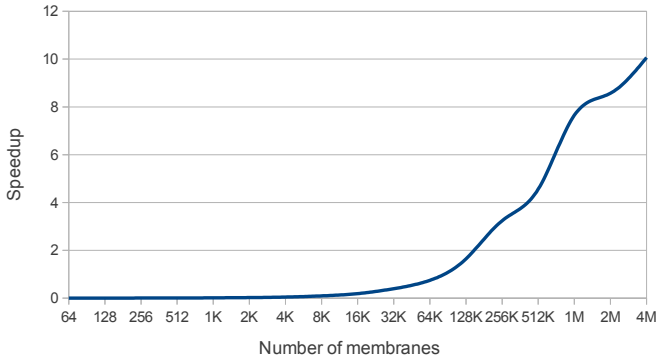


Fig. 5: Speedup achieved running test (256 Objects/Cell) for *tsp-sat-gpu* and *tsp-sat-seq*. GPU data management is also considered.

First of all, we should analyze the differences between them to better understand the different behaviors. We highlight the following:

- Computational steps: given $m, n \in \mathbb{N}$, representing the number of clauses and variables respectively, the cell-like P systems take $5n + 2m + 3$ steps, and the tissue P systems require $2n + 2m + nm + 1$. Thus, the computation of the tissue-like solution is longer (in number of steps), if $m > 3 + \frac{2}{n} \simeq 3$.
- Phases: *am-sat-gpu* is based on 4 phases (implemented in 3 kernels), whereas *tsp-sat-gpu* uses 5 phases (implemented in 4 kernels).
- Memory requirements: each membrane in *am-sat-gpu* is represented by a number of 32-bit integers equals to $|cod(\varphi)|$, but the tissue-like simulators use for them $2n + 4 + |cod(\varphi)|$. Thus, *tsp-sat-gpu* uses, in total, $(2n + 4)2^n$ bytes more.

Figure 6 compares both solutions using Test 2. It can be observed that the kernels of *am-sat-gpu* outperforms *tsp-sat-gpu*, even using optimizations for the last one. This improvement implies a speedup of 2.9x. However, if we take into account the data management in the GPU, we can see that the behavior of them is almost similar. The simulator *am-sat-gpu* runs just a bit faster, but for 2 M membranes, the speedup is almost 2x. This makes us to think that the data implementation of *am-sat-gpu* can be improved, since it requires an inferior amount of data. Recall that *am-sat-gpu* has not any GPU oriented enhancements, as in *tsp-sat-gpu*.

We finish this comparison by reporting their corresponding maximum speed-up with their sequential counterparts, which is of 63x and 10x for the cell-like and tissue-like simulators, respectively. Therefore, using the GPU for the cell-like solutions allows to get better performance gain.

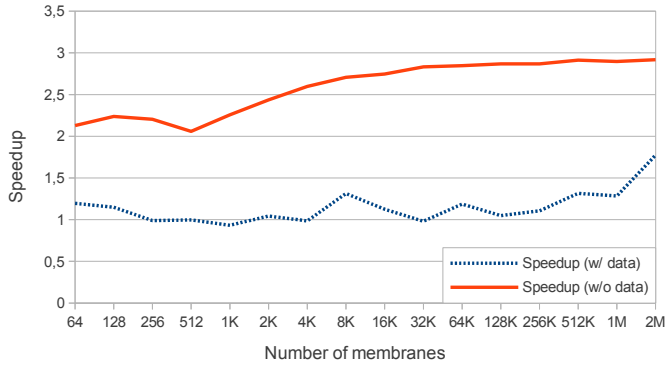


Fig. 6: Achieved speedup for both *tsp-sat-gpu* and *am-sat-gpu* simulators, considering (*w/ data*) or not considering (*w/o data*) the time for data management.

5.3 Characterizing the simulation on the GPU

Next, we characterize the simulations carried out in this work. From the comparison of the simulators for the cell-like and the tissue-like solutions, we have observed that the cell-like simulations are better carried out by the GPU. Thus, we have identified two properties that have helped to improve the performance of these GPU simulators:

- *Charges*: the model of P systems with active membranes associates charges to the membranes. They can be used to store information over the computation as well. If they are considered (and effectively used) for a given solution (e.g. to encode the truth assignment for `SAT`), less memory would be required (remind that the tissue-like simulator requires $2^n(2n+4)$ bytes more). In fact, the information encoded by charges can save objects that may or not may appear simultaneously in membranes, what saves also memory, and so, the number of threads to launch, working with much less objects.
- *Rules with no cooperation*: the model of P systems with active membranes defines rules with no cooperation, that is, the number of objects appearing in their left-hand sides is always 1. This property helps threads to be assigned to each rule, what also means to work with each object in parallel. Rules permitting cooperation (as in tissue P systems) require to take care of which objects are accessed by rules (and threads). It would be also interesting to study each type of rule (i.e., division, communication for tissue-like, and division, dissolution, send-in, send-out and evolution for cell-like) separately. Recall that a more flexible and general simulator for active membranes [2], the constraints of send-in, send-out, division and dissolution rules have to be considered for each membrane, what degrades parallelism on the GPU (it implies using local locks). However, in the model of tissue P systems these restrictions are not presented. A flexible simulator for tissue models can be implemented in a future

to study what is better: usage of charges but restricting types of rules, or not using charges (i.e. more objects per membrane) and more (but less restrictive) parallel rules.

6 Conclusions

In this paper we have presented a recent result on the parallel simulation with GPUs of an efficient solution to `SAT` by tissue P systems with cell division. The CUDA simulator design is similar to the one used in the previous simulator for a solution based on P systems with active membranes. Each thread block is assigned to each cell labeled by 2. However, the number of objects to be placed inside each cell in the memory representation is increased.

Experiments show that the CUDA simulator outperforms the sequential one by 10x. It can be seen that solving the same problem (`SAT`) under different P system variants leads to different speedups on the GPU (up to 2.9x for the cell-like simulator against the tissue-like). Indeed, we show that the usage of charges can help to save space devoted to objects, and rules without cooperation to increase thread parallelism.

Future work will be focused on developing new GPU-based simulators for other P systems models, and on improving the existing ones. In addition, further research can be carried out concerning the parallel simulation of particular P systems features, identifying which of them can be easily combined and efficiently simulated by the GPU. In this way, novel approximations for parallel simulators development can be performed also at the P systems area. An approach is to define a P system model combining all the good features for GPU simulators (let call it *GP systems*, or GPU oriented P systems). Then, the creation of a GPU based simulator for GP systems would be straightforward, considering the corresponding GPU oriented optimizations. However, it would be important to define a translation protocol from other P systems models to GP systems.

Acknowledgments

The authors acknowledge the support of "Proyecto de Excelencia con Investigador de Reconocida Valía" of the "Junta de Andalucía" under grant P08-TIC04200, and the support of the project TIN2012-37434 of the "Ministerio de Economía y Competitividad" of Spain, both co-financed by FEDER funds.

References

1. F. Cabarle, H. Adorna, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez. Improving GPU Simulations of Spiking Neural P Systems, *Romanian Journal of Information Science and Technology*, **15**, 1 (2012), 5–20.

2. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulation of P systems with Active Membranes on CUDA, *Briefings in Bioinformatics*, **11**, 3 (2010), 313–322.
3. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Simulating a P system based efficient solution to SAT by using GPUs, *Journal of Logic and Algebraic Programming*, **79**, 6 (2010), 317–325.
4. J.M. Cecilia, J.M. García, G.D. Guerrero, M.A. Martínez-del-Amor, M.J. Pérez-Jiménez, M. Ujaldón. The GPU on the simulation of cellular computing models, *Soft Computing*, **16**, 2 (2012), 231–246.
5. A. C. Elster. High-performance computing: Past, present and future, *L*NCS, **2367** (2002), 433–444.
6. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, Agustín Riscos-Núñez. An overview of P-Lingua 2.0, *LNCS*, **5957** (2010), 264–288.
7. D. Kirk, W. Hwu. *Programming Massively Parallel Processors: A Hands On Approach*, MA (USA), 2010.
8. M. Harris. Mapping computational concepts to GPUs, *ACM SIGGRAPH 2005 Courses*, NY (USA), 2005.
9. M.A. Martínez-del-Amor, I. Pérez-Hurtado, A. Gastalver-Rubio, A.C. Elster, M.J. Pérez-Jiménez. Population Dynamics P systems on CUDA, *LNCS*, **7605** (2012), 247-266.
10. V. Nguyen, D. Kearney, G. Gioiosa. Balancing performance, flexibility, and scalability in a parallel computing platform for Membrane Computing applications, *LNCS*, **4860** (2007), 385-413.
11. J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, J.C. Phillips. GPU Computing, *Proceedings of the IEEE*, **96**, 5 (2008), 879-899.
12. G. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and TUCS Report No 208.
13. G. Păun. Attacking **NP**-complete problems, In *Unconventional Models of Computation, UMC'2K*, 2000, 94-115.
14. G. Păun, M.J. Pérez-Jiménez, A. Riscos-Núñez. Tissue P System with cell division. *International Journal of Computers, Communications & Control*, **3**, 3 (2008), 295-303.
15. G. Păun, G. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, 2010.
16. M. Qasem. WinSAT website, 2009,`http://www.mqasem.net/sat/winsat`.
17. *Official NVIDIA CUDA website*. `http://www.nvidia.com/cuda`
18. *The PMCGPU project website*. `http://sourceforge.net/p/pmcgpu`

Continuous Versus Discrete: Some Topics with a Regard to Membrane Computing

Adam Obtułowicz

Institute of Mathematics, Polish Academy of Sciences
Śniadeckich 8, P.O.B. 21, 00-956 Warsaw, Poland
e-mail: A.Obtulowicz@impan.gov.pl

Summary. Some questions and open problems are formulated in the context of a dilemma continuous approach versus discrete approach to the investigations of dynamics of complex biological and physical systems with a regard to membrane computing [11].

1 A question about an extent of discretization programs of physics

Fredkin–Sorkin–Wolfram discretization programs of physics via E. Fredkin's digitalization [5], R. D. Sorkin's causal sets [15], and S. Wolfram's cellular automata approach [18] give rise to a question:

Does the discretization mean a lost (or eventually how to find or establish counterparts) of classical qualitative properties of continuously (with respect to time among others) treated processes like the properties:

- a property of reaching equilibrium and its stability [16],
- asymptotic behaviour (i.e. tending of process trajectories—the solutions of some differential equations to some possibly regular curves like limiting cycles [16]),
- irregular behaviour:
  - chaos [6], [7], [16], [13], [14], [1], [2],
  - perturbations and noise approached by stochastic treatment of system dynamics.

One should notice that the status of the concepts of an equilibrium and its stability varies from biomedical physist's critique that these concepts are not adequate to capture creative forces of nature—"a system that reached equilibrium is 'dead'", cf. [7], to their importance, for instance, for the methods (due to G. Grossberg and J. J. Hopfield) of modelling (associative) memory and learning (training) in neural networks, reviewed, e.g., in [9].

The varying status of the concepts of an equilibrium and its stability accompanied the emergence of a new research area, called *nonlinear science* (cf. [13], [14]), comprising nonlinear dynamics (cf. Box1 in [17] and the book [16]), where chaos is an important issue.

Nonlinear science requires new mathematical tools beyond calculus and the discretization mentioned above provided some of these new tools, e.g. cellular automata.

The discretization does not diminish the role of continuous-time approach to system dynamics. The review [4] and the papers [1], [2] confirm that the continuous-time approach is still alive.

2 Answer

Some (partial) answer to the main question of Section 1 is contained in:

- characterization of irregular behaviour of processes represented by large graphs (like causal sets and their Hasse diagrams) and networks in terms of dimensions [10], in particular fractal dimension [12], like chaos in continuous dynamics is approached in terms of fractals [16];
- the attempts of making the discrete constructs continuous one, like K. Martin and P. Panangaden work [8] of building back space-time manifold from Sorkin like causal order;
- the embeddings of discrete-time system behaviour in continuous-time dynamics, cf. [4], where an embedding of a Turing machine behaviour in continuous-time dynamics is presented.

Concerning membrane computing [11] one could:

- represent processes generated by P systems by causal sets like T. Bolognesi [3] represents computational processes of various mechanisms,

then

- approach the causal sets representing processes generated by P systems like in the answer to the main question given above.

One could also investigate P system behaviour by its embedding in continuous-time dynamics, like in [4], to approach the irregularities, like chaos, of the resulting continuous-time dynamics of P systems in the manner of [1], [2].

References

1. Banasiak J., Lachowicz M., Moszyński M., *Topological chaos: when topology meets medicine*, Applied Mathematics Letters **16** (2003), pp. 303–308.

2. Banasiak J., Lachowicz M., Moszyński M., *Chaotic behavior of semigroups related to the processes of gene amplification-deamplification with cell proliferation*, Mathematical Biosciences **206** (2007), pp. 200–215.
3. Bolognesi T., *Causal sets from simple models of computation*, Int. Journal of Unconventional Computing **6** (2010), pp. 489–524.
4. Bournez O., Campagnolo M.L., *A survey of continuous time computations*, in: New Computational Paradigms. Changing Conception of what is Computable, Springer, 2008, pp. 383–423.
5. Fredkin E., *An introduction to digital philosophy*, Int. Journal of Theoretical Physics **42** (2003), pp. 189–247.
6. Hill A., *Chaotic chaos*, Math. Intelligencer **22**:3 (2000), p. 5.
7. Klonowski W., *The metaphor of "Chaos"*, in: System Biology: Principles, Methods and Concepts, ed. A. K. Konopka, CRC Press, 2006, pp. 115–138.
8. Martin K., Panangaden P.A., *Domain of spacetime intervals for General Relativity*, Comm. Math. Phys. **267** (2006), pp. 563–586.
9. McEliece R., et al., *The capacity of the Hopfield associative memory*, IEEE Transactions on Information Theory **33** (1987), pp. 461–482.
10. Nowotny T., Requardt M., *Dimension theory of graphs and networks*, J. Phys. A Math. Gen. **31** (1998), pp. 2447–2463.
11. Păun G., Rozenberg G., Salomaa A. (eds.), *The Oxford Handbook of Membrane Computing*, Oxford, 2009.
12. Rozenfeld H.D., Gallos L.K., Song Ch., Makse H.A., *Fractal and transfractal scale-free networks*, Encyclopedia of Complexity and System Science, Springer, 2009, pp. 3924–3943.
13. Scott A.C. (ed.), *Encyclopedia of Nonlinear Science*, Routledge, New York, 2004.
14. Scott A.C., *The Nonlinear Universe: Chaos, Emergence, Life*, Springer, 2007.
15. Sorkin R.D., *Causal sets: Discrete gravity*, in: Lectures on Quantum Gravity (Valdivia, 2002), ed. A. Gomberoff and D. Marlof, Springer, 2005, pp. 305–327.
16. Strogatz S.H., *Nonlinear Dynamics and Chaos*, Perseus Books Publ., LLC, 1994.
17. Strogatz S. H., *Exploring complex networks*, Nature **410**, 8 March 2001, pp. 268–276.
18. Wolfram S., *A New Kind of Science*, Wolfram Media Inc., 2002.

The "Catalytic Borderline" Between Universality and Non-Universality of P Systems

Gheorghe Păun

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

Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, ghpaun@gmail.com

Summary. *P systems* are computing models inspired by the structure and the functioning of the living cells; they are the basic computing devices of *membrane computing*, a branch of *natural computing*. The present note is an overview of results and open problems related to the borderline between the computationally universal and non-universal catalytic P systems. A short introduction to membrane computing is provided, to the use of the reader not familiar with this research area.

1 A Glimpse to Membrane Computing

Along its evolution, computer science has continuously looked to biology in order to find ideas (data structures, operations with them, ways to control these operations, architectures for computing devices, etc.) useful for improving the use of the existing electronic computers and for developing new computing tools. In the last decades, this tendency became a well defined branch of computer science, called *natural computing*. Besides the goal sketched above, a complementary one is to understand and investigate the processes taking place in nature – especially in biology – as computations.

Membrane computing is one of the research areas of natural computing, having as the starting point the living cell, considered alone or as a part of more complex structures, such as tissues and organs, including the brain. This direction of research was initiated in 1998 ([12]) and it developed rapidly: already in 2003, the Thompson Institute for Scientific Research, ISI, mentioned membrane computing as a fast emerging research front in computer science, with [12] considered a "fast breaking paper" (see http://esi-topics.com). The literature of the domain is now very large, including monographs, collective volumes, PhD theses, research projects. An introduction to membrane computing can be found in [14],

with a comprehensive presentation (at the level of year 2009) in [15]. Up-to-date information (including information about the two yearly meetings in this area, the February Brainstorming Week on Membrane Computing and the summer Conference on Membrane Computing) can be found at the domain website [16].

Very generally speaking, membrane computing deals with processing *objects*, by means of bio-inspired *operations*, in the compartments of a cell-like or tissue-like arrangement of *membranes*. Basically, (i) the objects are symbols from a given alphabet (but they can also be strings or can have a more complex structure), (ii) like in biochemistry, the objects are present in the *multiset* sense (each object has a precise multiplicity in a given compartment), (iii) the operations are abstractions of biochemical *reactions* or other types of operations inspired from the biology of the cell (e.g., symport and antiport, for passing objects across membranes, or operations with membranes – division, separation, phagocytosis, and so on); (iv) the arrangement of membranes is either hierarchical, like in a cell, or "horizontal", like in tissues and other populations of cells (e.g., of bacteria). The operations (reactions) are also called *evolution rules*, or, shortly, *rules*. Like in biochemistry, in the basic model, the operations take place in a *non-deterministic* way (the rules to apply and the objects to react are chosen non-deterministically), in parallel (simultaneously in all compartments, with the objects in each compartment evolving in parallel, according to the local rules). Many variants were investigated, with respect to the types of rules, the ways to use them, the arrangement of membranes. Using the rules, we can pass from a configuration of the system to the next configuration – and in this way we can define *computations*. What is obtained, called a *P system*, was not initially meant as a model of the biological cell, to the use of biologists, but a computing model, of interest for computability.

There are two basic theoretical issues to be addressed for any new computing model, including the P systems: the computing power (competence), and the computing efficiency (performance). Accordingly, two are the reference frameworks: the Turing machines and their restrictions in what concerns the power, and the complexity classes (in particular, the theoretical borderline between tractability and intractability, between polynomial complexity and exponential complexity) in what concerns the efficiency.

From both these two points of view, membrane computing proved to be successful: many classes of P systems are equivalent in power with Turing machines (hence, according to Turing-Church thesis, they can compute whatever an algorithm can compute; we also call this property *computational completeness* or *Turing universality*), while many classes of P systems (especially those equipped with the possibility of producing an exponential working space in polynomial time, e.g., by means of membrane division or string replication) can solve computationally hard problems (typically, **NP**-complete problems, but also harder problems) in a feasible time (typically, polynomial, but often even linear).

A basic question in both these research directions (power and efficiency) is to find the borderline between universality and non-universality, in what concerns the power, and between efficiency and non-efficiency. In this paper we recall some

results about the first issue, namely, considering borderline results concerning the universality of *catalytic P systems* – definitions will be given in the next section.

In parallel with the theoretical investigations, mainly dealing with the previously mentioned questions, power and efficiency, P systems proved to be useful tools for several *applications*, starting with the very field where they originated – biology and biomedicine. This is now one of the main trends of research in this area. For the biologist, membrane computing has several attractive features: the models are directly inspired from biology, they are easy to be understood, P systems deal with discrete mathematical structures (as encountered in many situations in biology, where traditional differential equations are not appropriate), they are easily scalable and programmable, and have an emergent behavior (the evolution cannot be predicted by examining the initial configuration and the evolution rules). For other areas of application (computer graphics, approximate optimization, robot control, etc.) the inherent parallelism, hence computational efficiency, is the central attractive feature. We here do not give details about applications; the reader is referred to the Handbook [15] and to the website mentioned above.

2 Catalytic P Systems

We now introduce the model we consider in this paper, the cell-like P systems, in the catalytic form, stressing once again that, from the biological reality, we abstract a mathematical model suitable for computability investigations, thus ignoring many biological details.

The basic ingredients of a (cell-like) P system are the following ones:

1. The *membrane structure* is a hierarchical arrangement of membranes, understood as 3D vesicles; a membrane without any other membrane inside is said to be *elementary*; each membrane defines a *region/compartment*, the space between the membrane and the immediately inner membranes, if any; the space outside the "skin" membrane is called the *environment*. Each membrane can be labeled, and the label will identify both the membrane and its region. The membrane structure can be represented by a rooted tree (with a membrane in each node and the skin in the root), hence also by an expression of correctly nested labeled parentheses. Sometimes we also use Euler-Venn diagrams (disjoint sets included in a unique external set, the skin one).
2. The *objects* are placed in the compartments of the membrane structure, in the form of multisets (sets with multiplicities associated with the elements). In membrane computing, the multisets are usually represented by strings, like in formal language theory, with the multiplicity of a symbol corresponding to the number of occurrences of that symbol in the string; thus, a string and all its permutations represent the same multiset. For instance, a^2bc^4ab represents the multiset which contains 3 copies of a, 2 copies of b, and 4 copies of c.
3. The *evolution rules* are multiset rewriting rules similar to reactions in chemistry/biochemistry. The basic form is $u \to v$, where u and v are multisets

of objects from a given set O. The use of such a rule means "consuming" the objects of u and "producing" the objects of v. (Note that we do not pay attention to conservation lows, we work with arbitrary multisets, the only restriction is that u is not empty.) In order to link the regions of a system, the objects produced by a rule $u \to v$ can have associated *target indications*, of the forms *here, out, in*, with the meaning that an object with the target *here* remains in the same region where the rule is applied, an object with the target *out* is sent out of the respective membrane (in this way, objects can also be sent to the environment, when the rule is applied in the skin region), while an object with the target *in* is sent to one of the immediately inner membranes, non-deterministically chosen (if there is no such membrane, i.e., if the rule is associated with an elementary membrane, then the rule $u \to v$ with v containing an object (a, in) cannot be applied). The indication *here* in general is omitted when writing the rules.

Both the objects and the rules are associated with compartments of the system; the rules in a given region ("reactor") can be applied only to the objects from the same region.

The way of using the rules which we consider here is the *non-deterministic maximally parallel* one: the rules to be applied are chosen non-deterministically, but in such a way that the choice is maximal, i.e., we apply a multiset of rules (each rule can be applied several times) which is maximal, no further rule can be added to it so that the obtained multiset is still applicable to the existing objects.

The membranes and the objects present in the compartments of a system at a given time form a *configuration*; starting from a given *initial configuration* and using the rules as explained above, we get *transitions* among configurations; a sequence of transitions forms a *computation*. A computation is *halting* if it reaches a configuration where no rule can be applied. With a halting computation we associate a *result*, in the form of the number of objects present in a specified elementary membrane in the halting configuration.

Thus, a (cell-like) P system can be formalized as a construct

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

where O is the alphabet of objects, μ is the membrane structure (with m membranes), $w_1, \ldots, w_m$ 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 evolution rules, associated with the regions of μ, and i_o is the label of an elementary membrane, used as the output membrane.

There are many variations of this basic model. For instance, if a rule $u \to v$ has at least two objects in u, then it is called *cooperative*; if u is a single object, then the rule is *non-cooperative*; an intermediate case is that of *catalytic* P systems, whose rules are of the form $ca \to cv$, where c is a special object which never evolves and never passes through a membrane (both these restrictions can be relaxed), but it just assists object a to become the multiset v. A catalytic P system is written in

the form $\Pi = (O, C, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, i_o)$, where all components are as above and $C \subset O$ is the set of catalysts.

We end this section with a simple example, illustrating the architecture and the functioning of a (cell-like) P system, as well as the usual way of graphically representing a P system. Figure 1 indicates the initial configuration (including the rules) of a system which computes the function $n \longrightarrow n^2$, for any natural number $n \geq 1$: we introduce the number n in the initial configuration, in the form of n copies of the object a present in the skin region, and we get the result as the number of copies of object f present in membrane 2 when the computation halts. Besides catalytic and non-cooperating rules, the system also contains a rule with promoters, $e \to e(f, in)|_b$: the object e evolves to ef only if at least one copy of object b is present in the same region.

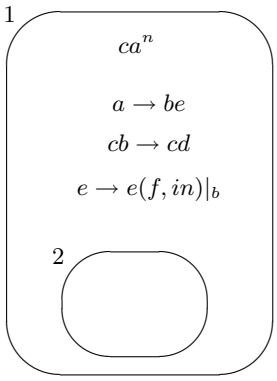


Fig. 1. A P system with catalysts and promoters

Formally, the system is given as follows:

$$
\begin{aligned}
\Pi &= (O, C, \mu, w_1, w_2, R_1, R_2, i_o) \text{ where} \\
O &= \{a, b, d, e, c, f\} \text{ (the set of objects)}, \\
C &= \{c\} \text{ (the set of catalysts)}, \\
\mu &= [\ [\ \ ]_2\ ]_1 \text{ (membrane structure)}, \\
w_1 &= ca^n \text{ (initial objects in region 1)}, \\
w_2 &= \emptyset \text{ (initial objects in region 2)}, \\
R_1 &= \{a \to be,\ cb \to cd,\ e \to e(f, in)|_b\} \text{ (rules in region 1)}, \\
R_2 &= \emptyset \text{ (rules in region 2)}, \\
i_o &= 2 \text{ (the output region)}.
\end{aligned}
$$

The system starts working by using the rule $a \to be$, which has to be applied in parallel to all copies of a; hence, in one step, all objects a are replaced by n

copies of b and n copies of e. From now on, the other two rules from region 1 can be used. The catalytic rule $cb \to cd$ can be used only once in each step, because the catalyst is present in only one copy. This means that in each step one copy of b is replaced by d. Simultaneously (because of the maximal parallelism), the rule $e \to e(f, in)|_b$ should be applied as many times as possible and this means n times, because we have n copies of e. Note the important difference between the promoter b, which allows using the rule $e \to e(f, in)|_b$, and the catalyst c: the catalyst is involved in the rule, it is counted when applying the rule, while the promoter makes possible the use of the rule, but it is not counted; the same (copy of one) object can promote any number of rules. Moreover, the promoter can evolve at the same time by means of another rule (the catalyst is never changed).

In this way, in each step we change one b to d and we produce n copies of f (one for each copy of e); the copies of f are sent to membrane 2 (the indication in from the rule $e \to e(f, in)|_b$). The computation should continue as long as there are applicable rules. This means exactly n steps: in n steps, the rule $cb \to cd$ will exhaust the objects b and in this way neither this rule can be applied, nor $e \to e(f, in)|_b$, because its promoter does no longer exist. The computation halts and in membrane 2, considered as the output membrane, we get n^2 copies of object f.

3 The Power of Catalytic P Systems

We start now to recall results about the computing power of catalytic P systems.

Let us denote by $NP_m(cat_r)$ the family of sets of numbers computed (generated, in the above sense) by P systems with at most m membranes, using catalytic or non-cooperative rules, containing at most r catalysts. We also denote by NRE the family of Turing computable sets of natural numbers ("recursively enumerable", hence the abbreviation), and by $NREG$ the family of semilinear sets of numbers (recognized by finite automata). When all the rules of a system are catalytic, we say that the system is *purely catalytic*, and the corresponding families of sets of numbers are denoted by $NP_m(pcat_r)$. When the number of membranes is not bounded by a specified m (it can be arbitrarily large), then the subscript m is replaced with $*$.

The following fundamental results are known:

Theorem 1. (i) $NP_2(cat_2) = NRE$, [5];
(ii) $NREG = NP_*(pcat_1) \subseteq NP_*(pcat_2) \subseteq NP_2(pcat_3) = NRE$, [7], [8].

Two intriguing open problems appear here, related to the borderline between universality and non-universality: (1) are catalytic P systems with only one catalyst universal? (2) are purely catalytic P systems with two catalysts universal?

We here consider only the first question. In the membrane computing community, the belief is that the answer is negative, one catalyst is not enough in order to equal the power of Turing machines. Preliminary results, supporting this conjecture, can be found, e.g., in [3].

On the other hand, in the membrane computing literature there are many results which show that P systems with only one catalyst are universal if further ingredients are added. Many results of this type can be found in [4], while recent developments can be found in [6] and [10]. We now recall several of these results, without always giving the place where they were reported first; such information can be found in the bibliography of [4]. The overall impression is that "one catalyst is almost universal": features which look "innocent" at the first sight are enough to lead P systems with one catalyst to universality.

4 Universality for P Systems with One Catalyst

Inspired from biochemistry and/or from computability theory, we may add various ingredients to P systems as defined above.

For instance, we may assume that some rules are "more active" than other rules, hence they have *priority* in being applied. This corresponds to considering a partial order relation among the rules in each compartment of a P system. It was proved already in [12] that $NP_2(cat_1, pri) = NRE$, where *pri* indicates the use of priorities.

In the example considered in Section 2 we have also used another feature, the *promoters*: rules can have associated objects which act as promoters, the rule can be applied only if at least one copy of each of the associated promoters is present. The promoters can evolve at the same time, but by other rules than those they promote. Similarly, rules can have associated *inhibitors*, objects whose presence forbids the application of the rule. Catalytic P systems with only one catalyst, using either promoters or inhibitors (one object only associated with a rule, not larger multisets), are universal.

Slightly more sophisticated is the control of the evolution of a P system by means of controlling the *membrane permeability*, [13]. This is achieved by using two operations, associated with usual multiset processing rules: decreasing the thickness (hence increasing the permeability) and increasing the thickness (hence decreasing the permeability) of membranes. Specifically, rules of the forms $u \to v\delta$ and $u \to v\tau$ are used. Initially, each membrane is supposed to be of thickness one. A membrane of thickness one behaves as a usual membrane, objects can be moved across it by means of target indications *in* and *out*. A membrane of thickness 0 is *dissolved*, its objects are left free into the surrounding region and its rules are "lost" (specific biochemistry is active in each membrane, hence, by dissolving a membrane, the respective "reactor" disappears). If a membrane has thickness 2, then it is impermeable, no object can pass across it (hence the rules which ask for such a passage cannot be used). The symbols δ, τ indicate the decrease and the increase, respectively, of the thickness by one. The thickness cannot be greater than 2, a rule $u \to v\tau$ used in such a membrane will lead to a membrane with the same thickness. As already expected, the use of the operations δ, τ (the control

of membrane permeability) again leads to the universality of P systems with one catalyst.

The previous idea actually is part of a larger research area in membrane computing, dealing with the possibility to also evolve the membrane structure during a computation. There are many biologically inspired operations of this type. Here we mention only one, the *membrane creation*, [11]. Besides usual non-cooperating and catalytic rules, we also use rules of the form $ca \to c[\ v\ ]_i$, with the meaning that object a, with the help of catalyst c, produces a new membrane, with the label i, having inside the multiset v; of course, the catalyst is reproduced. Also this feature leads to universality in the case of P systems with only one catalyst.

If instead of "standard" catalysts we use catalysts with additional features, then again we obtain universality. Two such extensions were considered: *bistable* catalysts and *mobile* catalysts, [9]. In the first case, the catalyst can oscillate between two *states* (and this is the only possible transformation the catalysts can have), in the latter case the catalyst can pass through membranes like any other object, by means of target indications *in* and *out*.

Several similar results were recently obtained in [6] and [10].

One of them is the *target restriction*. This restriction has two components, one at the syntactic level (in each rule $u \to v$, all target indications which appear in v are identical), and one at the semantic level (in each step of a computation, in each membrane one uses rules with the same target indication in their right hand member; in different membranes, different target indications may be used, while the choice of rules to apply is done as in general P systems, in the non-deterministic maximally parallel way). Interesting enough, the universality of target restricted one catalyst P systems is obtained in [6] by means of P systems with 7 membranes (it is an open problem whether or not the number of membranes can be diminished).

Another restriction considered in [6] is the *time-varying*: a sequence $U_1, \ldots, U_p$ of sets of rules is given, the computation starts with a step when rules from the set U_1 are used, then we use rules from U_2, and so on; after step p, when rules from U_p are used, we return to U_1 and continue (in step $pn + i, n \geq 0$, one uses rules from set U_i). The universality of time-varying P systems is obtained for one catalyst P systems with only one membrane, having the *period* p equal to 6.

In [10], so-called *label restricted* P systems are considered: each rule has a label, which can be a symbol from a given alphabet, or it can be the empty label; a computation is label restricted if in each transition one applies only rules with the same label, possibly also rules with the empty label. Although this restriction does not look too strong, it is sufficient to get universality of P systems with only one catalyst.

5 Final Remarks

This paper is only a hint to one of the research directions in membrane computing. After a brief introduction to membrane computing, we have recalled several cases where P systems with only one catalyst are computationally equivalent with Turing machines. Various ingredients were considered: a priority relation among rules, promoters, inhibitors, the control of membrane permeability, mobile catalysts, bistable catalysts, membrane creation, label restricted transitions, selection of rules by the target indications, time varying sets of rules. Such results are of interest in view of the conjecture that P systems with only one catalyst are not universal (two catalysts are known to lead to universality).

Several problems remain open. For instance, because the conjecture is that purely catalytic P systems with two catalysts are not universal, all the results mentioned above for the one catalyst case should also be examined for the purely catalytic P systems with two catalysts.

Furthermore, other additional features remain to be considered, with the aim of increasing the power of P systems with one catalyst – for instance, inspired from the regulated rewriting area [2] or the grammar systems area [1] in formal language theory.

Acknowledgements. Work supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200. A careful reading of the first version of the text by Rudi Freund is gratefully acknowledged.

References

1. E. Csuhaj-Varjú, J. Dassow, J. Kelemen, Gh. Păun: *Grammar Systems. A Grammatical Approach to Distribution and Cooperation.* Gordon and Breach, London, 1994.
2. J. Dassow, Gh. Păun: *Regulated Rewriting in Formal Language Theory.* Springer-Verlag, Berlin, 1989.
3. R. Freund: Particular results for variants of P systems with one catalyst in one membrane. *Proc. Fourth Brainstorming Week on Membrane Computing*, Fénix Editora, Sevilla, 2006, vol. II, 41–50.
4. R. Freund, O.H. Ibarra, A. Păun, P. Sosík, H.-C. Yen: Catalytic P systems. Chapter 4 of [15].
5. R. Freund, L. Kari, M. Oswald, P. Sosík: Computationally universal P systems without priorities: two catalysts are sufficient. *Th. Computer Sci.*, 330 (2005), 251–266.
6. R. Freund, Gh. Păun: Universal P Systems: One Catalyst Can Be Sufficient. *Proc. 11th Brainstorming Week on Membrane Computing*, Sevilla, 4-8 February 2013, Fénix Editora, Sevilla, 2013, to appear.
7. O.H. Ibarra, Z. Dang, O. Egecioglu: Catalytic P systems, semilinear sets, and vector addition systems. *Th. Computer Sci.*, 312 (2004), 379–399.
8. O.H. Ibarra, Z. Dang, O. Egecioglu, G. Saxena: Characterizations of catalytic membrane computing systems. *28th Intern. Symp. Math. Found. Computer Sci.*, 2003 (B. Rovan, P. Vojtás, eds.), LNCS 2747, Springer, 2003, 480–489.

9. S.N. Krishna, A. Păun: Results on catalytic and evolution-communication P systems. *New Generation Computing*, 22 (2004), 377–394.
10. K. Krithivasan, Gh. Păun, A. Ramanujan: On controlled P systems. *Fundamenta Informaticae*, to appear.
11. M. Mutyam, K. Krithivasan: P systems with membrane creation: Universality and efficiency. *Proc. MCU 2001* (M. Margenstern, Y. Rogozhin, eds.), LNCS 2055, Springer, Berlin, 2001, 276–287.
12. Gh. Păun: Computing with membranes. *J. Comput. Syst. Sci.*, 61 (2000), 108–143 (see also TUCS Report 208, November 1998, `www.tucs.fi`).
13. Gh. Păun: Computing with membranes – A variant. *Intern. J. Found. Computer Sci.*, 11, 1 (2000), 167–182.
14. Gh. Păun: *Membrane Computing. An Introduction*. Springer, Berlin, 2002.
15. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
16. The P Systems Website: `http://ppage.psystems.eu`.

Some Open Problems about Numerical P Systems

Gheorghe Păun

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

Department of Computer Science and Artificial Intelligence
University of Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, ghpaun@gmail.com

Summary. Some open problems and research topics related to numerical P systems are formulated – also recalling the problems from the corresponding section of the "mega-paper" [2] produced for the previous BWMC.

1 Introduction

Although introduced already in 2006, [8] (see also Section 8.1 of Chapter 23 of [10]) – and being an alternative to multiset or string processing P systems, meant to compute using numerical variables – the numerical P systems have received more attention only in the last years, especially in the framework of devising and implementing controllers for mobile robots.

In short, numerical P systems are a class of computing models inspired by both the cell structure and economics: numerical variables evolve in the compartments of a cell-like structure by means of so-called *production–repartition programs*. The variables have a given initial value and the production function is usually a polynomial whose values for the current values of variables is distributed among variables in the neighboring compartments according to the "repartition protocol". In this way, the values of variables evolve; all positive values taken by a specified variable are said to be computed by the P system.

In a natural way, these systems can also be used for computing mappings: specified variables of the system are considered as being the function variables and specified variables provide the results (hence functions from vectors to vectors of numbers can be computed – this is the case also in the robot control; of course, a suitable way to define the end of the computation should be found – halting, for instance, although in the basic computing model the computation is not supposed to halt).

In what follows, in order to help the reader (however, (s)he is supposed to be familiar with basic elements of membrane computing), I will first recall the

definition of numerical P systems, as given in [10] (Section 23.8.1), with two simple examples, then I will briefly discuss, following [15], the enzymatic numerical P systems as used in robot control; finally, further research suggestions are given.

2 Definitions

We consider usual cell-like membrane structures (with the standard $1, 2, \ldots, m$ labeling of membranes). The regions delimited by these membranes contain numerical variables. The variables in region i are written in the form $x_{j,i}, j \geq 1$. The value of $x_{j,i}$ at time $t \in \mathbf{N}$ is denoted by $x_{j,i}(t)$. These values can be of various types – in what follows we consider integers as values of variables (although in many applications one would most probably use real numbers – this is the case for robot control).

In order to evolve the values of variables, we use *programs*, composed of two components, a *production function* and a *repartition protocol*. The former can be any function with variables from a given region – here we are interested in computability issues, hence we consider only polynomials with integer coefficients. Using such a function (chosen non-deterministically if there are several programs in a given region), we compute a *production value* of the region at a given step. This value is distributed to variables from the region where the program resides, and to variables in its upper and lower neighbors according to the repartition protocol associated with the used production function. For a given region i, let $v_1, \ldots, v_{n_i}$ be all these variables. Following [8], here we consider as repartition protocols expressions of the form

$$c_1|v_1 + c_2|v_2 + \ldots + c_{n_i}|v_{n_i},$$

where $c_1, \ldots, c_{n_i}$ are natural numbers. The idea is that the coefficients $c_1, \ldots, c_{n_i}$ specify the proportion of the current production distributed to each variable $v_1, \ldots, v_{n_i}$.

This is precisely defined as follows. Consider a program

$$(F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}), \quad c_{l,1}|v_1 + c_{l,2}|v_2 + \ldots + c_{l,n_i}|v_{n_i})$$

and let

$$C_{l,i} = \sum_{s=1}^{n_i} c_{l,s}.$$

At a time instant $t \geq 0$ we compute $F_{l,i}(x_{1,i}(t), \ldots, x_{k_i,i}(t))$. The value $q = F_{l,i}(x_{1,i}(t), \ldots, x_{k_i,i}(t))/C_{l,i}$ represents the "unitary portion" to be distributed according to the repartition expression to variables $v_1, \ldots, v_{n_i}$. Thus, $v_{l,s}$ will receive $q \cdot c_{l,s}, 1 \leq s \leq n_i$.

A program as above is also written in the form

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}) \rightarrow c_{l,1}|v_1 + c_{l,2}|v_2 + \ldots + c_{l,n_i}|v_{n_i}.$$

A production function may use only part of the variables from a region. Those variables "consume" their values when the production function is used (they become zero) – the other variables retain their values. To these values – zero in the case of variables contributing to the region production – one adds all "contributions" received from the neighboring regions.

Thus, a *numerical P system* is a construct of the form

$$\Pi = (\mu, (Var_1, Pr_1, Var_1(0)), \ldots, (Var_m, Pr_m, Var_m(0)), x_{j_0,i_0}),$$

where μ is a membrane structure with m membranes labeled injectively by $1, 2, \ldots, m$, Var_i is the set of variables from region i, Pr_i is the set of programs from region i (all sets Var_i, Pr_i are finite), $Var_i(0)$ is the vector of initial values for the variables in region i, and x_{j_0,i_0} is a distinguished variable (from a distinguished region i_0), which provides the result of a computation.

Each program is of the form specified above: $pr_{l,i} = (F_{l,i}(x_{1,i}, \ldots, x_{k_i,i}), c_{l,1}|v_1 + c_{l,2}|v_2 + \ldots + c_{l,n_i}|v_{n_i})$ denotes the lth program from region i, where $\{x_{1,i}, \ldots, x_{k_i,i}\} \subseteq Var_i$ (not all variables from region i should appear in $F_{l,i}$).

Such a system evolves in the way informally described before. Initially, the variables have the values specified by $Var_i(0), 1 \leq i \leq m$. A transition from a configuration at time instant t to a configuration at time instant $t+1$ is made by (i) choosing non-deterministically one program from each region, (ii) computing the value of the respective production function for the values of local variables at time t, and then (iii) computing the values of variables at time $t+1$ as directed by repartition protocols. A sequence of such transitions forms a computation, with which we associate a set of numbers, namely, the numbers which occur as positive values of the variable x_{j_0,i_0}; this set of numbers is denoted by $N^+(\Pi)$. If all numbers, positive or negative, are taken into consideration, then we write $N(\Pi)$.

3 Examples

I illustrate the previous definition with the numerical system Π_1 given in Figure 1 with the distinguished variable $x_{1,1}$. One can easily see that variable $x_{1,3}$ increases by 1 at each step, also transmitting its value to $x_{1,2}$. In turn, region 2 transmits the value $2x_{1,2} + 1$ to $x_{1,1}$, which is never consumed, hence its value increases continuously. In the initial configuration all variables are set to 0. Thus, $x_{1,1}$ starts from 0 and continuously receives $2i + 1$, for $i = 0, 1, 2, 3, \ldots$, which implies that in $n \geq 1$ steps the value of $x_{1,1}$ becomes $\sum_{i=0}^{n-1}(2i + 1) = n^2$, and consequently $N^+(\Pi_1) = \{n^2 \mid n \geq 0\}$.

The system Π_1 was deterministic; let us consider also a non-deterministic system:

$$\Pi_2 = ([\quad]_1, (\{x_{1,1}\}, \{(2x_{1,1}, 1|x_{1,1}),\ (3x_{1,1}, 1|x_{1,1})\}, 1), x_{1,1}).$$

The production is assigned to the unique variable, but in each step we can choose either the first program or the second one; in the former case $x_{1,1}$ is multiplied by

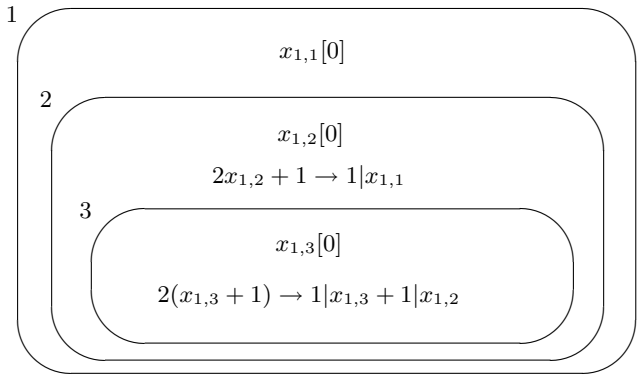


Fig. 1. The system Π_1

2, and in the latter case it is multiplied by 3. Thus, the values of $x_{1,1}$ will be of the form 2^i3^j, with $i \geq 0, j \geq 0$. Actually, *all numbers of this form are values of* $x_{1,1}$, where the value 2^i3^j is obtained in step $i+j$.

In these two examples we have chosen the programs in such a way that the production value is divisible by the total sum of coefficients c_j from each region (let us denote this case with *div*). When a current production is not divisible by the given total value of coefficients, then we can take the following decisions: (i) the remainder is lost (the production which is not immediately distributed is lost), (ii) the remainder is added to the production obtained in the next step (the non-distributed production is carried over to the next step), (iii) the system simply stops and aborts, no result is associated with that computation. We denote these three cases by *lost, carry, stop*, respectively.

4 Families of Numbers Computed

Thus, we can distinguish many types of systems, depending on the programs and their use. The family of sets of numbers $N(\Pi)$ computed by numerical P systems with at most m membranes, production functions which are polynomials of degree at most n, with at most r variables in each polynomial, with nonnegative coefficients, and the distribution of type α is denoted by $NNP_m(poly^n(r), nneg, \alpha)$, $m \geq 1, n \geq 0, r \geq 0, \alpha \in \{div, lost, carry, stop\}$. The restriction to deterministic systems is indicated by adding the letter D in front of NNP. If arbitrary coefficients are allowed, then the indication "nneg" is removed. If one of the parameters m, n, r is not bounded, then it is replaced by $*$. The set of positive numbers occurring as values of the output variable is denoted by $N^+(\Pi)$, and NN gets the superscript + when considering the family of such sets. Let NRE be the family of Turing computable sets of natural numbers.

Here are some results concerning these families – mode details can be found in [8].

Theorem 1. (i) $DNN^+P_1(poly^1(1), nneg, div) - SLIN_1^+ \neq \emptyset$.
(ii) $SLIN_1^+ \subset DNN^+P_*(poly^1(1), nneg, div)$.

The main result of [8] shows that, surprisingly enough, numerical P systems of a rather restricted type are Turing complete, even when using small numbers of membranes and polynomials of low degrees with a small number of variables:

Theorem 2. $NRE = NN^+P_8(pol^5(5), div) = NN^+P_7(poly^5(6), div)$.

The proof is based on the characterization of recursively enumerable sets of numbers as positive values of polynomials with integer values, [3]. Latter (see below) the register machines were used in universality proofs, and similar results were obtained, in certain cases, also for deterministic numerical P systems.

Many research topics are open for numerical P systems, among others: a throughout investigation of all classes of systems mentioned above, considering also vectors of numbers, looking for non-universal classes (and decidability results for those classes), hierarchies and normal forms.

5 Enzymatic Numerical P Systems

The numerical P systems were recently used in a series of papers (see references in [1], [14]) for implementing controllers for mobile robots; in this framework the P systems work in the computing mode: an input is introduced in the form of the values of some variables and an output is produced, as the values of other variables. Furthermore, in the robot control context, the so-called *enzymatic* numerical P systems were introduced and used, [4], [5], [6]. Such systems correspond to *catalytic P systems* in the "general" membrane computing: a program is applied only if the value of the associated enzyme is strictly greater than the smallest value of any variable involved in the production polynomial. Enzyme variables are not consumed or produced by the rules which they catalyze, but can be changed by the rules for which they do not act as catalysts. Therefore, their values can evolve during the computational process.

More formally (we recall the definition from [12]), enzymatic numerical P systems (in short, EN P systems) use both evolution programs as introduced above and programs of the form

$$F_{l,i}(x_{1,i}, \ldots, x_{k_i,i})|_{e_{j,i}} \rightarrow c_{l,1}|v_1 + c_{l,2}|v_2 + \ldots + c_{l,n_i}|v_{n_i},$$

where $e_{j,i}$ is a variable from Var_i different from $x_{1,i}, \ldots, x_{k_i,i}$, and from $v_1, \ldots, v_{n_i}$. Such a program can be applied at a time t only if $e_{j,i}(t) > \min(x_{1,i}(t), \ldots, x_{k_i,i}(t))$. (A slightly more complex definition is considered in [5] and [6] where: $e_{j,i}(t) > \min(|x_{1,i}(t)|, \ldots, |x_{k_i,i}(t)|)$. Considering the absolute value of the variables, instead

of their real values, simplifies the design of the membrane structures used to compute *cos* and *sin* functions as power series, but here we consider only the simpler case defined above. We also use here a notation different from that in [5], writing the enzyme in the same way as the promoters are written in multiset rewriting rules.) Note that in order to apply the program it is sufficient that *one variable* has the current value strictly smaller than the value of the enzyme variable. The enzyme cannot evolve by means of the associated program, but it can evolve by means of other programs, and can receive "contributions" from other programs and regions.

Because the enzymes are usual variables, playing a different role only "locally", in specified programs, we do not consider their set separated, hence the general writing of an enzymatic numerical P systems is the same as that of a numerical P system – only the form of programs can be different.

Using enzymes introduces a checking possibility in our systems (we compare the value of the enzyme with the values of variables from the associated program), and this suggests the possibility of choosing the positive values of the output variable "inside the system".

Tissue numerical P systems are also considered in [12], with a parallel use of programs. If in each membrane, at each step, we use a maximal set of programs (programs are selected non-deterministically, and a set of programs is applied only if it is maximal, i.e., no further program can be added to it in such a way that the new set is still applicable). Clearly, two cases are possible: (i) a variable can appear only in *one* production function, and this is the only restriction in choosing (non-deterministically) the programs to apply in a step (we denote this variant with *oneP*), and (ii) if two or more programs which are enabled at a computation step (i.e., they satisfy the condition imposed by the associated enzymes), share variables in their production functions, then they will all use the current values of those variables (we denote this with *allP*).

A large variety of classes of numerical P systems is created in this way: (1) enzymatic or non-enzymatic, (2) deterministic or non-deterministic, (3) sequential, all-parallel, one-parallel, (4) used in the generating, computing, accepting mode, etc. By combining these variants – also considering the cases *div, lost, carry, stop* from the previous sections – a large variety of classes of numerical P systems can be investigated.

In the notations of the families $NN^{\alpha}P_m(poly^n(r),\ldots)$ considered in the previous sections we add now the indication *enz* when enzymes are used, and one of *seq, oneP, allP*, depending on the way (sequential or parallel) the rules are used. When tissue systems are used, we write $NNtP_m(poly^n(r),\alpha,\beta,\gamma)$. However, in what follows we do not mention *div* and *nneg*, as they are always present.

Here are the main results from [8] (written in the new notation) and [12].

Theorem 3. $NRE = NN^+P_8(poly^5(5), seq) = NN^+P_7(poly^5(6), seq) = NNP_7(poly^5(5), enz, seq) = NNtP_*(poly^1(11), enz, oneP) = NNP_{254}(poly^2(253), enz, allP, det)$.

A considerable improvement of the last equality was proved in [13]:

Theorem 4. $NRE = NNP_4(poly^1(6), enz, allP, det)$.

Whether or not the parameters appearing in these results are optimal or not is an open problem.

6 Open Problems

Only a few of the many cases mentioned above were so far investigated, the other ones wait for further research efforts.

In particular, we have seen that enzymes improve the universality results in terms of the complexity of used polynomials, both in the cell-like case and the tissue-like case, provided that the evolution programs are used in a parallel manner. However, two different types of parallelism were used in the two cases – hence the question: can the one-parallel mode (used for tissue P systems) be used also in the cell-like case?

Various extensions of "general" notions in membrane computing to numerical P systems remain to be examined – this is a rich research topic. For instance, other ways of using the programs can be considered: minimally parallel, with bounded parallelism, asynchronously. Then, we can also consider rules for handling membranes, such as membrane division and membrane creation. These operations are the basic tools by which polynomial solutions to computationally hard problems, typically, **NP**-complete problems, are obtained in the framework of P systems with symbol objects. Is this possible also for numerical P systems? This is a particularly interesting issue, both from the point of view of applications and because in this way we can achieve "fypercomputations", [7], in terms of numerical P systems.

Of course, a natural research topic is to further explore the use of numerical P systems in controlling robots, and to look for further applications where functions from $\mathbf{R}^{k_1}$ to $\mathbf{R}^{k_2}$ should be computed. In this framework an important question is to develop a complexity theory based on numerical P systems: define specific complexity classes, compare them with existing classes, look for ways to speed-up computations (see also the previous suggestion: to bring to numerical P systems further ideas investigated for symbol object P systems, in particular, tools to create an exponential working space in polynomial time).

I end with another natural question: defining dP systems, as in [9], with the components being numerical P systems. Can this be useful from the computation efficiency ("parallelization") point of view?

References

1. C. Buiu, C.I. Vasile, O. Arsene: Development of membrane controllers for mobile robots. *Information Sciences*, 187 (2012), 33–51.

2. M. Gheorghe, Gh. Păun, M.J. Pérez-Jiménez, G. Rozenberg: Research frontiers of membrane computing: Open problems and research topics. *Intern. J. Found. Computer Sci.*, in press (a preliminary version was published in the proceedings of *Tenth Brainstorming Week on Membrane Computing*, Sevilla, 2012, vol. II, 171–250).
3. Y. Matijasevitch: *Hilbert's Tenth Problem*. MIT Press, Cambridge, London, 1993.
4. A.B. Pavel, C. Buiu: Using enzymatic numerical P systems for modeling mobile robot controllers. *Natural Computing*, in press, DOI: 10.1007/s11047-011-9286-5, 2011
5. A.B. Pavel, O. Arsene, C. Buiu: Enzymatic numerical P systems – a new class of membrane computing systems. *The IEEE Fifth Intern. Conf. on Bio-Inspired Computing. Theory and applications. BIC-TA 2010*, Liverpool, Sept. 2010, 1331–1336.
6. A.B. Pavel, C.I. Vasile, I. Dumitrache: Robot localization implemented with enzymatic numerical P systems. *Proc. Living Machines 2012*, LNCS 7375, Springer, 2012, 204–215.
7. Gh. Păun: Towards "fypercomputations" (in membrane computing). *Languages Alive. Essays Dedicated to Jurgen Dassow on the Occasion of His 65 Birthday* (H. Bordihn, M. Kutrib, B. Truthe, etcs.), LNCS 7300, Springer, Berlin, 2012, 207–221.
8. Gh. Păun, R. Păun: Membrane computing and economics: Numerical P systems. *Fundamenta Informaticae*, 73 (2006), 213–227.
9. Gh. Păun, M.J. Pérez-Jiménez: Solving problems in a distributed way in membrane computing: dP systems. *Int. J. of Computers, Communication and Control*, 5, 2 (2010), 238–252.
10. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *The Oxford Handbook of Membrane Computing*. Oxford Univ. Press, 2010.
11. The P Systems Web Page: `http://ppage.psystems.eu`.
12. C.I. Vasile, A.B. Pavel, I. Dumitrache, Gh. Păun: On the power of enzymatic numerical P systems. *Acta Informatica*, 49, 6 (2012), 395–412.
13. C.I. Vasile, A.B. Pavel, I. Dumitrache: Universality of enzymatic numerical P systems. *Intern. J. Computer Math.*, in press.
14. C.I. Vasile, A.B. Pavel, J. Kelemen: Implementing obstacle avoidance and follower behaviors on Koala robots using numerical P systems. *Tenth Brainstorming Week on Membrane Computing*, Sevilla, 2012, vol. II, 215–227.
15. C.I. Vasile, A.B. Pavel, I. Dumitrache, Gh. Păun: Numerical P systems. Section 13 in [2].

Bridging Membrane and Reaction Systems – Further Results and Research Topics

Gheorghe Păun[1,2], Mario J. Pérez-Jiménez[2], Grzegorz Rozenberg[3]

[1] Institute of Mathematics of the Romanian Academy
PO Box 1-764, 014700 Bucharest, Romania
[2] Research Group on Natural Computing
Department of Computer Science and AI, University of Sevilla
Avda Reina Mercedes s/n, 41012 Sevilla, Spain
gpaun@us.es, marper@us.es
[3] Leiden Institute for Advanced Computer Science - LIACS
Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
rozenber@liacs.nl

Summary. This paper continues an investigation into bridging two research areas concerned with natural computing: membrane computing and reaction systems. More specifically, the paper considers a transfer of two assumptions/axioms of reaction systems, non-permanency and the threshold assumption, into the framework of membrane computing. It is proved that: (1) SN P systems with non-permanency of spikes assumption characterize the semilinear sets of numbers, and (2) symport/antiport P systems with threshold assumption (translated as ω multiplicity of objects) can solve SAT in polynomial time. Also, several open research problems are stated.

1 Introduction

This paper continues research aimed at bridging two research areas concerned with processes inspired by the functioning of living cells, *membrane computing* (see, e.g., [10], [11], [14]) and reaction systems (see, e.g., [1], [3] – [6]). Membrane computing (based on *P systems*) essentially deals with *multisets*, processed in the compartments of a *membrane structure* according to rules of various types, such as, e.g., multiset rewriting and symport/antiport rules. Thus, the objects are present with specified multiplicity within the regions delimited by membranes, some of them evolve by the rules associated with membranes while the objects which are not involved in the rules used at a given step remain unchanged – thus they can be used in the subsequent processing steps.

The situation is very different in reaction systems. First of all, because of the assumed abstraction level this is a qualitative model, i.e., there is no counting: one

deals with sets rather than with multisets. Consequently, it is assumed that if an entity is present, then it is present in enough copies to be used by all reactions that use this entity as a reactant. This is referred to as a *threshold assumption*. Secondly, an entity is present in a successor state T' of a given state T only if it is produced by a reaction enabled in T or it is put into T' by the environment/context. This reflects the basic bioenergetics of the living cell, and it is referred to as the *non-permanency assumption*.

In this paper we continue an investigation of bridging membrane computing and reaction systems (see [12] and [13]) by transferring the threshold and the non-permanency assumptions to the framework of P systems. In particular, we investigate the resulting computing power and efficiency of some classes of P systems. We prove that:

(1) spiking neural (in short, SN) P systems with non-permanency of spikes characterize/compute just semilinear sets of numbers, while traditional SN P systems are Turing complete,

(2) symport/antiport P systems with the threshold assumption can solve **NP**-complete problems in polynomial time – this is illustrated with SAT, the satisfiability of propositional formulas in the conjunctive normal form.

We conclude this paper by stating a number of research problems.

2 Prerequisites

We assume the reader to be familiar with basic elements of membrane computing (e.g., from [11], [14], [17]) and of language theory (e.g., from [16]). Here we only recall some general notions and notations.

The language of all strings over an alphabet V is denoted by V^*, the empty string is denoted by λ, and $V^+ = V^* - \{\lambda\}$.

We denote by $SLIN_1$ the family of semilinear sets of numbers, and by NRE the family of recursively enumerable (Turing computable) sets of numbers. Semilinear sets are the length sets of regular languages, which are languages characterized by *regular expressions* or generated by *regular grammars*. A regular grammar is specified in the form $G = (N, T, S, P)$, where N is the nonterminal alphabet, T is the terminal alphabet, $S \in N$ is the axiom of the grammar, and P is a set of rules, each of which is of the form $A \to aB$, $A \to a$, where $A, B \in N, a \in T$.

As customary in membrane computing, we represent the multisets over an alphabet V by strings in V^* (hence a string and all its permutations represent the same multiset). Thus, we speak interchangeably about strings and multisets over V, and $|w|$ represents both the length of the string w and the cardinality of the multiset (represented by) w. We also write $w \subseteq w'$ for the inclusion between multisets (represented by the strings) w and w'. Since a set M is a multiset where all elements have multiplicity one, it is represented by a string containing each symbol from M exactly once.

2.1 Membrane Computing

We briefly recall here two classes of P systems which we investigate in this paper: the *symport/antiport P systems*, [9], and the *spiking neural* (SN) *P systems*, [8].

A membrane structure is a cell-like hierarchical arrangement of labeled membranes (understood as 3D vesicles); the external membrane is usually called the *skin* membrane, and a membrane without any membrane inside is called *elementary*. With each membrane, one associates a *region*, which is the space delimited by it and the inner membranes, if any. A membrane structure can be naturally represented by a rooted tree or by an expression of labeled parentheses (with a unique external parenthesis, associated with the skin).

A *symport* rule is either of the form (x, in) or of the form (x, out), and an *antiport* rule is of the form $(z, out; w, in)$, where x, z, and w are multisets of objects. These rules formalize the biological operations of moving several objects at a time across a membrane, either in the same direction, as is the case for symport rules or in opposite directions, as is the case for antiport rules.

A *P system with symport/antiport rules* is a construct of the form

$$\Pi = (O, \mu, w_1, \ldots, w_m, E, R_1, \ldots, R_m, i_{in}, i_{out}),$$

where O is an alphabet of objects, μ is a membrane structure with m membranes (here, labeled by $1, \ldots, m$, but any set of labels associated in a one-to-one manner to membranes can be used), $w_1, \ldots, w_m$ are the multisets present in the initial configuration in the m regions of μ (delimited by membranes labeled by $1, \ldots, m$, respectively), $E \subseteq O$, $R_1, \ldots, R_m$ are finite sets of symport/antiport rules associated with the m membranes of μ, and i_{in}, i_{out} are the input and the output regions of the system (i_{in} indicates a region of μ, while i_{out} can also be the environment of the system – we write then $i_{out} = env$). The objects of E are supposed to be present in the environment of the system with an arbitrary multiplicity.

Using an antiport rule $(z, out; w, in)$ associated with a membrane i means sending the multiset z out of region i and, simultaneously, bringing the multiset w into membrane i from the outside region adjacent to membrane i. Similarly for symport rules, where only one multiset of objects is moved across membrane i.

(Note that the symport/antiport rules do not change the number of objects, but they only displace them – that is why we need a supply of objects in the environment; this supply is inexhaustible, i.e., it does not matter how many objects are introduced into the system, still arbitrarily many objects remain in the environment.)

The rules are used in the nondeterministic maximally parallel manner. In the initial configuration, an *input* is introduced into region i_{in} in the form of a multiset, and the *result* of a computation is given in region i_{out}, most typically at the end of the computation (when no rule can be applied). If the system is used in the generative mode, then i_{in} is ignored/removed. If the system is used in the accepting mode, then i_{out} is ignored and the input is accepted if and only if the computation halts. In the computing mode both i_{in} and i_{out} are used. In particular, the system

can be used in the decidability mode: the code of an instance of a decision problem is introduced in i_{in} and the result is obtained in i_{out}: an object `yes` is placed in i_{out} at a specified step of the computation if and only if the instance of the problem has a positive answer.

More precise definitions of what it means to solve a decision problem in terms of P systems (complexity classes, uniform versus semi-uniform solutions, frontiers of efficiency etc.) can be found in many places – we mention here only [15] and the corresponding chapter from [14]. Several results in this area say that P systems able to produce an exponential workspace in a linear time (e.g., by membrane division, membrane creation, string replication) can solve computationally hard problems (typically, **NP**-problems) in polynomial time; the term *fypercomputation* was proposed in [12] for this situation, a sort of analogy to an established term *hypercomputation*, see, e.g., [2].

It is known that symport/antiport P systems (with a small number of membranes and with rules of a low complexity) used in the generative or the accepting mode characterize NRE (see, e.g., [14]).

Note that in the previous definitions multisets play a crucial role: objects appear with a finite multiplicity and the objects which do not evolve by a rule remain unchanged. However, the objects from the set E are used according to the threshold assumption (but they do not obey the non-permanency assumption).

The threshold assumption can be applied to some or to all membranes of a symport/antiport P system. In such *distinguished* membranes (where the threshold assumption applies), any object – even if it comes from a neighboring membrane with a specified multiplicity, maybe in only one copy – is present in arbitrarily many copies. A P system with such membranes is said to be an ωP *system.*

Another class of P systems investigated in this paper is that of *spiking neural P systems*, in short, SN P systems. Such a system (with extended rules, without delay, of degree $m \geq 1$) is a construct of the form

$$\Pi = (O, \sigma_1, \ldots, \sigma_m, syn, out),$$

where:

1. $O = \{a\}$ is the singleton alphabet (a is called *spike*);
2. $\sigma_1, \ldots, \sigma_m$ are *neurons* of the form

$$\sigma_i = (n_i, R_i), 1 \leq i \leq m,$$

   where:
   a) $n_i \geq 0$ is the *initial number of spikes* contained by the neuron;
   b) R_i is a finite set of *rules* each of which is in one of the following two forms:
      (1) $E/a^c \to a^p$, where E is a regular expression over $\{a\}$ and $1 \leq p \leq c$;
      (2) $a^s \to \lambda$, for some $s \geq 1$, with the restriction that $a^s \in L(E)$ for no rule $E/a^c \to a$ of type (1) from R_i;
3. $syn \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, m\}$ with $(i, i) \notin syn$ for $1 \leq i \leq m$ (*synapses* between neurons);

4. $out \in \{1, 2, \ldots, m\}$ indicates the *output neuron*.

The rules of type (1) are *firing* (we also say *spiking*) *rules*, and they are applied as follows. If the neuron σ_i contains k spikes, $a^k \in L(E)$ and $k \geq c$, then the rule $E/a^c \to a^p$ can be applied, and this means that c spikes are consumed, only $k - c$ remain in the neuron, and p spikes are produced and submitted to all neurons σ_j such that $(i, j) \in syn$ (each σ_j receives p spikes). The rules of type (2) are *forgetting* rules: if the neuron contains exactly s spikes, then the rule $a^s \to \lambda$ can be used, and this means that all s spikes are removed. The rules are used in the sequential manner within each neuron, and in parallel for all neurons of the system.

Using the rules as described above (see more detailed/precise definitions in the literature), we can define transitions among configurations. With a computation we can associate a result in several ways. The basic one associates a number to each computation (halting or not), viz., as the number of steps elapsed between the first and the second time when the output neuron spikes. The set of such numbers "generated" by Π is denoted by $N_2(\Pi)$. Another possibility is to count all spikes sent to the environment by the output neuron during halting computations. The set of numbers computed by Π in this way is denoted by $N_{out}(\Pi)$.

For both modes two types of results were obtained: Turing computability in the case of neurons without any bound on the number of spikes present inside, and a characterization of semilinear sets of numbers in the case of systems whose neurons have a bound on the number of spikes (we also call such systems *bounded*).

Note that also for SN P systems the multisets (counting the spikes in each neuron) and the permanency (spikes unused remain in the neurons) are essential.

3 The Effect of Non-Permanency

The non-permanency feature was considered in [13] for two classes of P systems: cooperative transition P systems and symport/antiport P systems, and in both cases the universality was proved. Hence there is no loss of power with respect to the traditional membrane computing case, where the objects which do not evolve survive. The case of catalytic P systems was stated as an open problem – recall that under the permanence assumption catalytic P systems are universal, even with two catalysts only, [7]. The effect of non-permanency was not investigated neither for non-cooperative transition P systems nor for the spiking neural P systems.

Let us denote by $N_{out}SNP_m(np)$ the family of sets of numbers $N_{out}(\Pi)$, for SN P systems Π with at most m neurons having the non-permanency property. When the generated numbers are taken as the number of steps between the first two steps when spikes are emitted by the output neuron, then we replace the subscript *out* by 2. The subscript m is replaced by $*$ if no bound on the number of neurons is assumed.

Lemma 1. $N_\alpha SNP_*(np) \subseteq SLIN_1$, *for* $\alpha \in \{2, out\}$.

Proof. If an SN P system Π (with m neurons) works in the non-permanency mode, then after each computation step each neuron has a number of spikes which is bounded by a constant depending on Π: each neuron emits a number of spikes, bounded by the maximum number of spikes produced by any rule of the system – denote this number by M. The spikes produced by a neuron σ_i can be replicated and submitted to at most $m-1$ other neurons (to which there is a synapse from σ_i), hence in total we have at most $m(m-1)M$ spikes. We start with a given initial number of spikes and at any moment we have in the system a bounded number of spikes, distributed to m neurons, which means a finite number of possible configurations of the system. These configurations can be taken as states of a finite automaton (or the nonterminals of a regular grammar) which simulates the work of the system. Taking as a result of a computation (of the automaton or of the grammar) either all spikes sent to the environment by the output neuron of Π or the number of steps between the first two spikes sent to the environment (e.g., we record in the configuration-nonterminal the fact that a spike was emitted, then we "count" until a second spike is emitted, and in that moment we stop the computation of the grammar), we obtain the inclusions of the lemma.

The above reasoning actually shows that an SN P system with non-permanency cannot use rules of the form $E/a^c \to a^p$ where for the regular expression E its language $L(E)$ is infinite. Thus, we can assume that each neuron contains only bounded rules, which then implies the semilinearity of the generated set of numbers (see already [8]).

Also the converse of the previous lemma holds. It was proved in [8] for bounded SN P systems, but the proof in [8] is rather complex (it starts from the characterization of semilinear sets of numbers as the union of a finite set with a finite number of arithmetical progressions), and it does not provide a bound on the number of neurons. Here we provide a direct proof (also bounding the number of neurons), starting from the characterization of semilinear sets of numbers as the length sets of regular languages. Of course, it is sufficient to consider regular languages over the one-letter alphabet.

Lemma 2. $SLIN_1 \subseteq N_{out}SNP_5(np)$.

Proof. Let us consider a regular grammar $G = (N, \{a\}, S, P)$ and assume that $N = \{A_1 = S, A_2, A_3, \ldots, A_n\}$. We construct the following SN P system (its initial configuration is given in a graphical form in Figure 1):

$$
\begin{aligned}
&\Pi = (\{a\}, \sigma_1, \ldots, \sigma_5, syn, 5), \text{ where}\\
&\sigma_1 = (n+1, \{a^{n+i} \to a^n \mid 1 \le i \le n\}),\\
&\sigma_2 = (0, \{a^{n+i} \to a^n \mid 1 \le i \le n\}),\\
&\sigma_3 = (n+1, \{a^{n+i} \to a^j \mid 1 \le i, j \le n, A_i \to aA_j \in P\}\\
&\quad \cup \{a^{n+i} \to a^{n+i} \mid 1 \le i \le n, A_i \to a \in P\}),\\
&\sigma_4 = (0, \{a^{n+i} \to a^j \mid 1 \le i, j \le n, A_i \to aA_j \in P\}
\end{aligned}
$$

$$\cup \{a^{n+i} \to a^{n+i} \mid 1 \le i \le n, A_i \to a \in P\}),$$
$$\sigma_5 = (0, \{a^i \to a \mid 1 \le i \le 2n\}),$$
$$syn = \{(1,2),(2,1),(1,4),(4,1),(2,3),(3,2),(3,4),(4,3),(3,5),(4,5)\}.$$

With each nonterminal $A_i, 1 \le i \le n$, we associate $n+i$ spikes, in neurons σ_3 and σ_4, where the rules in P are simulated. Initially, only neurons σ_1 and σ_3 spike, sending spikes to "partner neurons" σ_2 and σ_4; when these later neurons spike, they send spikes to the former neurons. The computation consists of such alternating steps. Neurons σ_3 and σ_4 also send spikes to the output neuron, σ_5, which sends a spike to the environment in each step. When a terminal rule $A_i \to a$ in P is simulated, neurons σ_3 and σ_4 produce $n+i$ spikes. The output neuron spikes once again, but all other neurons stop working: there is no rule which processes $2n+i$ spikes (these spikes are removed because of the non-permanency axiom, but this is not important since the computation halts anyway).

Consequently, the system Π produces k spikes if and only if $a^k \in L(G)$, hence $SLIN_1 \subseteq N_{out}SNP_5(np)$.

Lemma 3. $SLIN_1 \subseteq N_2SNP_5(np)$.

Proof. We consider the SN P system from the proof of Lemma 2, but now we replace the output neuron σ_5 by the following neuron:

$$\sigma_5 = (a^{2n+1}, \{a^{2n+1} \to a\} \cup \{a^{n+i} \to a \mid 1 \le i \le n\}).$$

The output neuron spikes in the first step and then it spikes only one step after the moment when a rule $A^i \to a$ was simulated in one of the neurons σ_3 or σ_4. In the steps for which $a^i, 1 \le i \le n$, spikes are sent to neuron σ_5; these spikes are removed – this is implied by the non-permanency axiom, because there is no rule to process them.

Consequently, the modified system spikes twice, at a distance of k steps if and only if $a^k \in L(G)$, hence $SLIN_1 \subseteq N_2SNP_5(np)$.

Combining the previous three lemmas we obtain:

Theorem 1. $SLIN_1 = N_\alpha SNP_\beta(np)$, *for all* $\alpha \in \{2, out\}$ *and* $\beta \in \{5, 6, \ldots\} \cup \{*\}$.

What about the SN P systems using less than five neurons? It is easy to see that computations in one-neuron systems last only one step, hence they produce only finite sets. SN P systems with two neurons can generate infinite sets – in the *out* mode. Here is an example of such a system:

$$\Pi = (\{a\}, \sigma_1, \sigma_2, \{(1,2),(2,1)\}, 2), \text{ where}$$
$$\sigma_1 = (2, \{a^2 \to a^2,\ a^2 \to a\}),$$
$$\sigma_2 = (0, \{a^2 \to a^2\}).$$

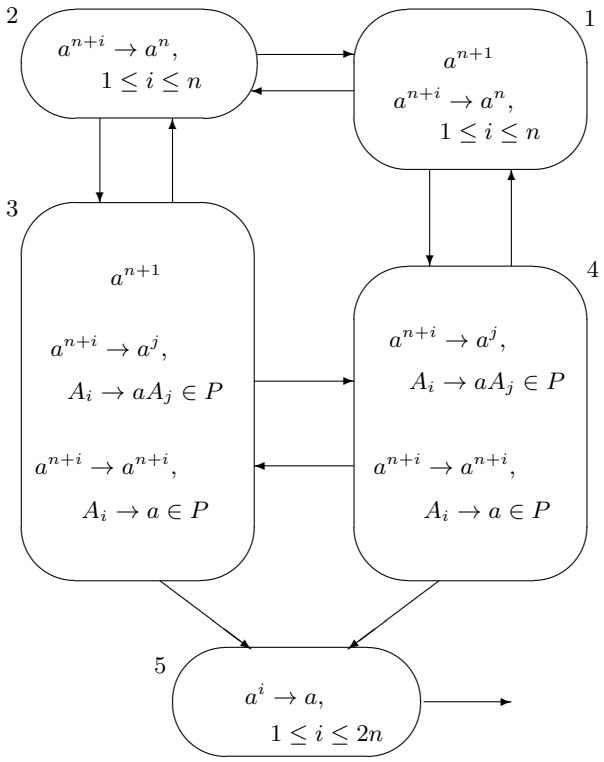


Fig. 1. The SN P system in the proof of Lemma 2

The computation can continue until using the rule $a^2 \to a$ in the first neuron – at that moment no rule can be applied in any neuron and all spikes vanishes. We have $N_{out}(\Pi) = \{2n \mid n \ge 0\}$.

Interestingly enough, when the result is the distance between the first two spikes, SN P systems with two neurons generate only singletons. If there is only one synapse between the two neurons, then each computation lasts one or two steps, hence only one of the numbers 0 and 1 can be generated. If the two neurons can communicate with each other, this can be done simultaneously or at most in alternate steps (after using a rule, no spikes remains in a neuron, because of the non-permanency assumption, hence new spikes must be obtained from the partner neuron); one of the neurons is the output neuron, hence it must spike twice in the first four steps of the computation and so only numbers 1 and 2 can be computed. One can easily see that the generated set is a singleton, containing one of the numbers 1, 2.

However, SN P systems with three neurons can generate infinite sets also as the distance between the first two spikes sent to the environment. This is the case

for the system Π in Figure 2, for which we have $N_2(\Pi) = \{n \mid n \geq 1\}$. The output neuron spikes in the first step of a computation and then only after the step when neuron σ_2 uses the rule $a?2 \to a$; as long as both σ_1 and σ_2 use their rules $a^2 \to a^2$, neuron σ_3 cannot use its rule, hence the four spikes it receives are lost, due to the non-permanency assumption.

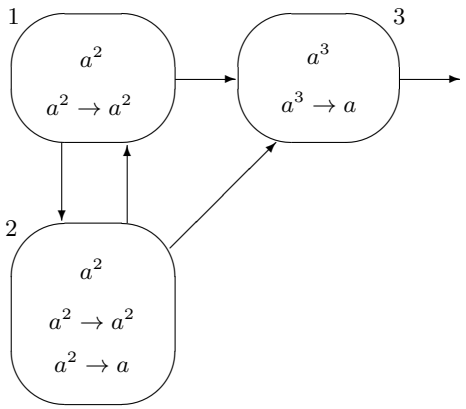


Fig. 2. An SN P system with three neurons generating an infinite set.

It remains an open problem whether SN P systems with four neurons characterize $SLIN_1$.

4 The Effect of the Threshold Assumption

It was proved in [12] that cooperative P systems without permanency, with two membranes where the inner one works under the threshold assumption (any object present here is available in arbitrarily many copies), can solve `SAT` in a polynomial time (actually, linear with respect to the number of clauses and independent of the number of variables) in a uniform way.

This result can be extended to symport/antiport P systems, with one additional feature: the system uses precomputed resources. More specifically, for a $SAT(n, m)$ problem (n variables and m clauses) we work with a number of objects of the order of n^m, i.e., all sets of at most m variables. These objects are given in advance, available in the initial configuration of the system, but they are "precomputed", provided at no cost, although there are exponentially many of them (but without containing other information than that provided by n and m). Of course, in this case we do not work in the standard complexity framework, as the systems solving a class of problems cannot be constructed in a polynomial time with respect to the size of the problems (actually, up to now there is no definition of complexity classes for the case of precomputed resources).

The proof follows the same idea as that of the result in [12], implemented for symport/antiport systems. Since in such systems the objects are only moved across membranes and they cannot be changed during the computation, we see no way to avoid using an exponential number of objects given in advance (in [12] these objects are created during the computation by means of cooperative multiset rewriting rules).

Theorem 2. SAT *can be solved (in a uniform way) in a polynomial time by symport/antiport ωP systems with precomputed resources.*

Proof. Let us consider the SAT problem for n variables, $x_1, x_2, \ldots, x_n$, and m clauses. We denote $Lit = \{x_i, \neg x_i \mid 1 \leq i \leq n\}$ and $Val = \{t_i, f_i \mid 1 \leq i \leq n\}$. Let $v(x_i) = t_i$ and $v(\neg x_i) = f_i$, for all $1 \leq i \leq n$. We have $\neg t_i = f_i$ and $\neg f_i = t_i$, for $1 \leq i \leq n$. (Note that t_i, f_i identify, by their subscripts, the variables with which they are associated.)

An instance $\gamma = C_1 \wedge C_2 \wedge \ldots \wedge C_m$ of $\mathtt{SAT}(n, m)$, with $C_i = y_{i,1} \vee y_{i,2} \vee \ldots \vee y_{i,k_i}$, for $y_{i,j} \in Lit, 1 \leq j \leq k_i$, is encoded as

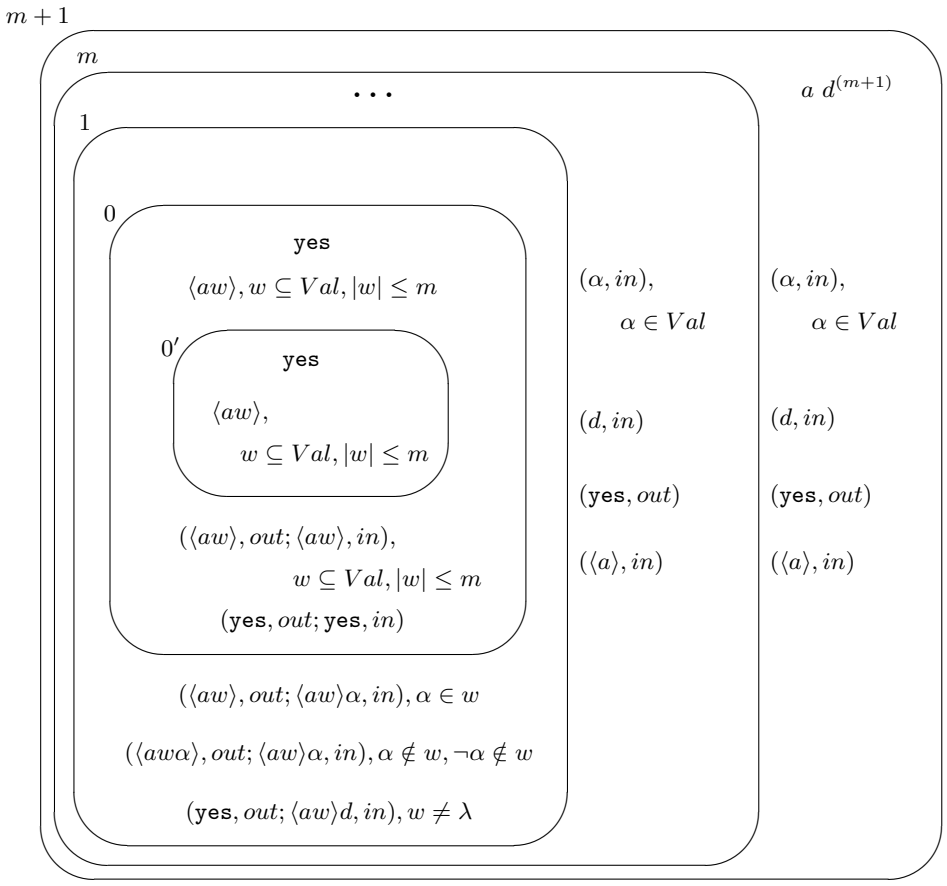
$$code(\gamma) = v(y_{1,1})^{(1)} \ldots v(y_{1,k_1})^{(1)} v(y_{2,1})^{(2)} \ldots v(y_{2,k_2})^{(2)} \ldots v(y_{m,1})^{(m)} \ldots v(y_{m,k_m})^{(m)}.$$

We now construct the following symport/antiport ωP system (the system works under the non-permanency assumption, the three inner membranes are distinguished, and the objects are present in them with ω multiplicity):

$$\begin{aligned}
\Pi &= (O, \mu, w_1, \ldots, w_{m+1}, w_0, w_{0'}, E, R_1, R_2, \ldots, R_{m+1}, R_0, R_{0'}, m+1, env),\\
O &= \{\alpha^{(j)} \mid \alpha \in Val, 1 \leq j \leq m\} \cup Val\\
&\quad \cup \{a, \langle a \rangle, \mathtt{yes}\} \cup \{d^{(i)} \mid 1 \leq i \leq m+1\} \cup \{\langle aw \rangle \mid w \subseteq Val, |w| \leq m\},\\
\mu &= [[\ \ldots [[[[\ ]_{0'}]_0]_1]_2 \cdots]_m]_{m+1},\\
w_{0'} &= w_0 = \{\mathtt{yes}\} \cup \{\langle aw \rangle \mid w \subseteq Val, |w| \leq m\},\\
w_1 &= w_2 = \ldots = w_m = \lambda,\ w_{m+1} = ad^{(m+1)},\\
E &= \{d^{(i)} \mid 1 \leq i \leq m\} \cup \{d, \langle a \rangle\} \cup Val,\\
R_{0'} &= \{(\mathtt{yes}, out; \mathtt{yes}, in\} \cup \{(\langle aw \rangle, out; \langle aw \rangle, in) \mid w \subseteq Val, 0 \leq |w| \leq m\},\\
R_0 &= \{(\langle aw \rangle, out; \langle aw \rangle \alpha, in) \mid \alpha \in Val, w \subseteq Val, 0 \leq |w| \leq m, \alpha \in w\}\\
&\quad \cup \{(\langle aw\alpha \rangle, out; \langle aw \rangle \alpha, in) \mid \alpha \in Val, w \subseteq Val, 0 \leq |w| \leq m, \alpha \notin w, \neg\alpha \notin w\}\\
&\quad \cup \{(\mathtt{yes}, out; \langle aw \rangle d, in) \mid w \subseteq Val, 1 \leq |w| \leq m\},\\
R_i &= \{(\alpha, in) \mid \alpha \in Val\} \cup \{(d, in),\ (\mathtt{yes}, out),\ (\langle a \rangle, in)\},\ 1 \leq i \leq m,\\
R_{m+1} &= \{(d^{(1)}, out; d, in),\ (\mathtt{yes}, out)\} \cup \{(d^{(i+1)}, out; d^{(i)}, in) \mid 1 \leq i \leq m\}\\
&\quad \cup \{(\alpha^{(j+1)}, out; \alpha^{(j)}, in) \mid \alpha \in Val, 1 \leq j \leq m-1\}\\
&\quad \cup \{(\alpha^{(1)}, out; \alpha, in) \mid \alpha \in Val\} \cup \{(a, out; \langle a \rangle, in)\}.
\end{aligned}$$

For an easier understanding, we also present this system in a graphical form, in Figure 3. (We follow here the standard way of representing a P system with

symport/antiport rules, i.e., indicating inside membranes the objects present in the initial configuration of the system, and on the outside side of each membrane the associated rules.)



$$(d^{(i+1)}, out; d^{(i)}, in), 1 \leq i \leq m$$
$$(d^{(1)}, out; d, in)$$
$$(\alpha^{(j+1)}, out; \alpha^{(j)}, in), \alpha \in Val, 1 \leq j \leq m-1$$
$$(\alpha^{(1)}, out; \alpha, in), \alpha \in Val$$
$$(\texttt{yes}, out),\ (a, out; \langle a \rangle, in)$$

Fig. 3. The symport/antiport ωP system from the proof of Theorem 2

The computation of Π starts after introducing the multiset $code(\gamma)$ into the skin membrane, for a given instance γ of the $\texttt{SAT}(n, m)$ problem. The truth-values

which satisfy the first clause bring from the environment the corresponding truth-values without superscripts. Simultaneously, object a is replaced by $\langle a \rangle$, and all other truth-values, corresponding to clauses $C_2, \ldots, C_m$, decrease by one their superscripts. Also the "checker" object d decreases, step by step, its superscript – but starting from $m+1$, hence by one greater than the superscripts of objects associated with C_m. The truth-values without superscripts and object $\langle a \rangle$ "migrate", step by step, towards membrane 1. First, the truth-values which satisfy C_1 (at the same time with $\langle a \rangle$) reach membrane 1, then those which satisfy C_2, and so on.

In membrane 1, objects of the form $\langle aw \rangle$ grow, starting from the "seed" $\langle a \rangle$, with w containing the truth-values which satisfy one by one the clauses. Specifically, if α satisfy clause C_i and it arrives in membrane 1, where we have the object $\langle aw \rangle$ (for $i = 1$ we have $w = \lambda$), with w containing the truth-values of the variables which satisfy all clauses $C_j, 1 \leq j \leq i-1$, then:

(1) if α is in w, then the object $\langle aw \rangle$ is not changed, and α is moved in membrane 0, where it will not survive,

(2) if neither α nor $\neg\alpha$ is in w, then this new truth-value α is added to w, by means of the rule $(\langle aw\alpha \rangle, out; \langle aw \rangle \alpha, in) \in R_0$,

(3) if none of the previous cases holds (i.e., $\neg\alpha$ appears in w), then no reaction takes place – hence both α and $\langle aw \rangle$ will disappear because of the non-permanency condition.

Note the important fact that the threshold assumption is crucial in this operation: each object, whether of the form α or $\langle aw \rangle$, appears in membrane 1 (and in membranes $0, 0'$) in the ω way, sufficient for all rules which can be applied (there is no competition for objects), hence *all* rules are applied simultaneously!

One step after the truth-values corresponding to the last clause, C_m, entered membrane 1, also d moves to membrane 1. It finds here all truth-assignments w which satisfy all clauses. If there is no such truth-assignment, then no reaction takes place in membrane 1 at that time – thus object d disappears and object `yes` is not released from membrane 0. If there is at least one non-empty truth-assignment w, then the rule $(\texttt{yes}, out; \langle aw \rangle d, in) \in R_0$ is used and `yes` is moved out of membrane 0 and from here it starts its way out of the system.

The internal membranes $0, 0'$ have the role of suppliers of objects: because of the non-permanency assumption, only objects which are moved by a rule survive.

If the formula γ is satisfiable, then object `yes` exits the system, otherwise this object remains inside. Let us count now the number of steps necessary to bring out object *yes*. Objects $d^{(i)}$ decrease their superscript from $m+1$ to 1 (hence $m+1$ steps), then $d^{(1)}$ is replaced by d (one more step). In further m steps, d crosses all membranes from region $m+1$ to region 1. Taking `yes` from membrane 0 needs one more step. Crossing all membranes $1, 2, \ldots, m+1$ requires $m+1$ steps. Thus, provided that γ is satisfiable, object `yes` exits the system in $3m+4$ steps.

Because of the rules associated with membrane $0'$, the system never halts, but the answer whether or not the formula γ is satisfiable is obtained in step $3m+4$: γ is satisfiable if and only if in this step we get `yes` out of the system.

Note that we work with precomputed resources: the total alphabet of the system as well as the contents (objects and rules) of membranes $0, 0'$ are exponential (of the order of n^m), the computation of these objects and rules is done in advance, at no cost, but there is no information in the system at the beginning of the computation related to γ different from n and m.

Finally, we notice that the construction is uniform (it starts from the problem itself, $\texttt{SAT}(n, m)$, not from a given instance of the problem), which concludes the proof.

5 Concluding Remarks. Other Cases to Consider

In this paper we continued the study of the effect of transferring to membrane computing the two basic axioms of reaction systems: the *non-permanency assumption* (an object which does not evolve disappears) and the *threshold assumption* (an object either does not appear, or it is present in arbitrarily many copies).

After recalling the results from [12] and [13], we established two new results: in the non-permanency case, SN P systems characterize the semilinear sets of numbers, and symport/antiport systems under the threshold assumption (imposed in only two membranes) can solve SAT in a polynomial time. All these results and the cases which were not yet investigated are displayed in Table 1.

	coop	cat	ncoop	S/A	SN P
Non-permanency	Univ. [13]	?	?	Univ. [13]	$SLIN_1$ Theorem 1
Threshold assumption	Fyper. [12]	?	?	Fyper. Theorem 2	?

Table 1. Cases studied – cases to be studied

Of course, there also are other open problems and research topics. Several of them were also mentioned in the previous sections. Certainly, an interesting question is whether the threshold assumption adds power to SN P systems working under the non-permanency assumption.

A natural research topic is to avoid using precomputed resources in Theorem 2 and instead to construct the exponentially many components of the initial configuration of the symport/antiport P system by using additional features of the system, e.g., using membrane division.

A "dual" research area is a transfer of ideas from membrane computing to reaction systems, but we do not address this issue here – some comments can be found in [13].

Acknowledgements. The work of the first two authors was supported by Proyecto de Excelencia con Investigador de Reconocida Valía, de la Junta de Andalucía, grant P08 – TIC 04200, co-financed by FEDER funds.

References

1. R. Brijder, A. Ehrenfeucht, M. Main, G. Rozenberg: A tour of reaction systems. *International J. Found. Computer Sci.*, 22 (2011), 1499–1518.
2. J. Copeland: Hypercomputation. *Minds and machines*, 12, 461-502 (2002)
3. A. Ehrenfeucht, J. Kleijn, M. Koutny, G. Rozenberg: Qualitative and quantitative aspects of a model for processes inspired by the functioning of the living cell. *Biomolecular Information Processing, From Logic Systems to Smart Sensors and Actuators* (E. Katz, ed.), Wiley-VCH, Weinheim, 2012, 303–322.
4. A. Ehrenfeucht, G. Rozenberg: Reaction systems. *Fundamenta Informaticae*, 75 (2007), 263–280.
5. A. Ehrenfeucht, G. Rozenberg: Events and modules in reaction systems. *Theoretical Computer Sci.*, 376 (2007), 3–16.
6. A. Ehrenfeucht, G. Rozenberg: Introducing time in reaction systems. *Theoretical Computer Sci.*, 410 (2009), 310–322.
7. R. Freund, L. Kari, P. Sosik: Computationally universal P systems without priorities: two catalysts are sufficient. *Theoretical Computer Sci.*, 330 (2005), 251–266.
8. M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 279–308 (2006)
9. A. Păun, Gh. Păun: The power of communication: P systems with symport/antiport. *New Generation Computing*, 20, 3 (2002), 295–306.
10. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143 (first circulated as Turku Center for Computer Science-TUCS Report 208, November 1998, `www.tucs.fi`).
11. Gh. Păun: *Membrane Computing. An Introduction.* Springer, Berlin, 2002.
12. Gh. Păun: Towards fypercomputations (in membrane computing). *Languages Alive. Essays Dedicated to Jurgen Dassow on the Occasion of His 65 Birthday* (H. Bordihn, M. Kutrib, B. Truthe, etcs.), LNCS 7300, Springer, Berlin, 2012, 207–221.
13. Gh. Păun, M.J. Pérez-Jiménez: Towards bridging two cell-inspired models: P systems and R systems. *Theoretical Computer Sci.*, 429 (2012), 258–264.
14. Gh. Păun, G. Rozenberg, A. Salomaa, eds.: *Handbook of Membrane Computing.* Oxford University Press, 2010.
15. M.J. Pérez-Jiménez: Computational complexity theory in membrane computing (Invited talk). *New Frontiers in Informatics. 24th Annual Conference of Academia Europaea*, Bergen, Norway, September 11, 2012.
16. A. Salomaa: *Formal Languages.* Academic Press, New York, 1973.
17. The P Systems Website: `http://ppage.psystems.eu`.

Analysing Gene Networks with PDP Systems. *Arabidopsis thaliana*, a Case Study

Luis Valencia–Cabrera[1], Manuel García–Quismondo[1], Mario J. Pérez–Jiménez[1], Yansen Su[2], Hui Yu[2], and Linqiang Pan[2]

[1] Research Group on Natural Computing
Dpt. of Computer Science and Artificial Intelligence, University of Sevilla
Avda. Reina Mercedes s/n. 41012 Sevilla, Spain
{lvalencia,mgarciaquismondo,marper}@us.es

[2] Department of Control Science and Engineering, Huazhong University of Science and Technology
1037 Luoyu Road, Wuhan, China
suyansen1985@163.com, yuhuihustac@gmail.com, lqpan@mail.hust.edu.cn

Summary. Gene Regulatory Networks (*GRNs*) are a useful tool for biologists to understand the interactions among genes in living organisms. A special kind of GRNs known as Logic Networks (*LNs*) has been recently introduced. These networks consider that the state of one or more genes can influence another one. In a previous work, we proposed a Membrane Computing model which simulates the dynamics of LNs by drawing on the improved LAPP algorithm. In this paper we provide a case study for our LN model on a network which regulates the circadian rhythms of long–term studied plant *Arabidopsis thaliana*. We outline the software tools employed and propose a methodology for analysing LNs on our Membrane Computing model. At the end of the paper, some conclusions and future work are included.

Keywords: Bioinformatics, Genetics, Gene networks, Membrane Computing, MeCoSim, Software engineering, Modelling, LAPP, Logic networks

1 Introduction

Since its very beginning, Membrane Computing [13] has been employed as a modelling framework for biochemical phenomena. Although the current landscape is more focused on metabolite–oriented dynamics, gene regulatory networks (GRNs) have also been modelled by means of P systems as part of this framework. In a previous work, we followed this line of research by proposing a Membrane Computing model for a specific type of gene networks known as Logic Networks (*LN*) [16]. This model describes a P systems family known as LN Dynamic P systems (*LN*

DP systems), within the framework of PDP systems [11]. LNs are a specific type of GRNs in which the combination of states of several genes, rather than the single state of any of them, influence another one. Bowers *et al.* [2] proposed a methodology for the construction of logic networks out of statistical data, known as Logical Analysis of Phylogenesis Profiles (LAPP). In our model, these combinations are limited to at most two genes affecting a third one. The model, in conjunction with DCBA algorithm[4], intends to capture the behaviour of the Improved LAPP Method introduced by Wang *et al.* [18]. In their work, they propose a case study on a gene network associated to *Arabidopsis thaliana*'s flowering process. We intend to reproduce this case study by using our Membrane Computing model. We also include a guide for generating custom simulators on MeCoSim for LN DP systems, depicting a step–by–step guide on MeCoSim tool [12]. Finally, the data employed in this case study is provided as an appendix, thus easing cross–checking of results. This paper is structured as follows. Section 1.1 introduces the Logic Network to be studied, a GRN associated to the flowering process of *Arabidposis thaliana*. Section 2 outlines the LN DP system model presented in [16], in order to make the current work self contained, as it is used to analyse our case study. Section 3 consists of a guide to simulate LNs from scratch on MeCoSim [14, 12]. This guide complements the simulation methodology described in [16]. Section 4 describes a case study on a real–world logic network on Arabidopsis Thaliana, in order to experimentally verify the behaviour of the model on complex gene networks. Finally, section 5 lists the conclusions obtained and proposes some open problems.

1.1 A Logic Network on *Arabidopsis thaliana* flowering processes

Arabidopsis is a long–day plant. Zhang and Zuo [19] stated that long–day conditions can promote reproductive growth and induce early flowering. However, short–day conditions can promote vegetative growth and induce late flowering or even no flowering. To understand the intrinsic mechanisms of Arabidopsis flowering in different lighting conditions, it is required to compare the relationships of related genes.

In the latest ten years, much work has been reported in the field about *A. thaliana* flowering. Imaizumi *et al.* [8] found that FKF1 is a blue light receptor which regulates flowering. Later, they also showed that FKF1 together with Flavin–Binding and Kelch Repeat degrade Cycling Dof Factor1 (CDF1) to eventually control carbon monoxyde [7]. In the same year, Abe *et al.* [1] found that Flowering Locus T (FT) together with FD activate Apetala1 (AP1) to initiate floral development and promote floral transition at the shoot apex. Previous work deal only with one or few genes related to flowering. However, the networks considered in this work focus on the relationships among a large number of genes systematically. Bowers *et al.* [2] proposed the Logic Analysis of Phylogenetic Profiles (LAPP) [2]. This method helps researchers to know biological functions of some genes or proteins on the basis of phylogenetic profiles, which has been developed both on theory and application ([3, 20, 17]). For example, Wang *et al.* [17]

developed the improved LAPP method, and reversely constructed a logic network of sixteen genes in shoot for Arabidopsis under salt stimuli.

2 Description of the model

This section summarizes both the P system family and the model (i.e., initial configuration and rule patterns) employed in this case study. For a detailed description of the model, see [16].

2.1 A family of P systems based on Logic Networks

The model depicted here is a P system of a family known as Logic Network Dynamic P systems (*LN DP systems*). An LN DP system is described within an expansion of Population Dynamics P systems (*PDP systems*) [16].

An LN DP system Π_{LN} of degree (q, m) with $q, m \geq 1$, taking $T \geq 1$ time units, is a tuple

$$\Pi_{LN} = (G, \Gamma, \Sigma, T, R_E, \mu, R, \{f_{r,j} : \ r \in R, 1 \leq j \leq m\}, \\ \{\mathcal{M}_{ij} : \ 1 \leq i \leq q, \ 1 \leq j \leq m\}, \{\mathcal{M}_j : 1 \leq j \leq m\})$$

where:

- $(G, \Gamma, \Sigma, T, R_E, \mu, R, \{f_{r,j} : \ r \in R, 1 \leq j \leq m\}, \{\mathcal{M}_{ij} : \ 1 \leq i \leq q, \ 1 \leq j \leq m\})$ is a PDP system.
- $\{f_{r,j} = 1 : \ r \in R, 1 \leq j \leq m\}$.
- For each j $(1 \leq j \leq m)$, $\mathcal{M}_j$ are multisets over Γ, describing the objects initially placed in the m environments e_j.

In this paper, in the description of an LN PDP System, functions $f_{r,j}$ are omitted. They are all equal to 1, so it is not necessary to make them explicit.

2.2 The model

Here the model for the family of Logic Network Dynamic P systems is outlined. This model covers any possible P system in this family, so the multisets, rules, etc. depend on the P system which represent each specific instance of a logic network. The definition of the general model requires the use of parameters in our constructs, as explained at the end of this subsection.

Let LN be a logic network. Let ng, nu, nb be the number of genes, unary and binary interactions of LN, respectively. Let $n = ng + nu + nb$. The model consists of the following PDP system of degree $(1, n)$,

$$\Pi_{LN} = (G, \Gamma, \Sigma, T, R_E, \mu, R, \{\mathcal{M}_{ij} : \ 0 \leq i \leq q-1, \ 1 \leq j \leq m\}, \{\mathcal{M}_j : \ 1 \leq j \leq m\})$$

where:

- G is a directed graph containing a node (environment) for each gene, unary or binary interaction, following this order.
- In the alphabet Γ, we represent gene states, interaction types, contribution weights and targets.

$$\begin{aligned}\Gamma = {} & \{a_i, b_i, c_i : 0 \leq i \leq 1\} \cup \{go, d_0\} \cup \{unop_j, binop_j : 1 \leq j \leq 4\} \cup \\ & \{auxDest_{i,g_{j,1},k} : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb + nu\} \cup \\ & \{dest_{i,g_{j,1},t_{k,1}+ng} : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb\} \cup \\ & \{dest_{i,g_{j,1},unt_{k-nb,1}+ng+nb} : 0 \leq i \leq 1, 1 \leq j \leq ng, nb+1 \leq k \leq nb+nu\} \cup \\ & \{e_{t_{k,4}*i+(1-i)*(1-t_{k,4}),t_{k,1}+ng} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{e_{t_{k,6}*i+(1-i)*(1-t_{k,6}),t_{k,1}+ng} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{e_{unt_{k-nb,4}*i+(1-i)*(1-unt_{k-nb,4}),unt_{k-nb,1}+ng+nb} : \\ & \ 0 \leq i \leq 1, nb+1 \leq k \leq nb+nu\} \cup \\ & \{eF_{t_{k,8}*i+(1-i)*(1-t_{k,8}),t_{k,1}+ng} : 0 \leq i \leq 1, 1 \leq k \leq nb\} \cup \\ & \{eF_{i,(unt_{k,1}+ng+nb)} : 0 \leq i \leq 1, 1 \leq k \leq nu\} \cup \\ & \{clock_j : 0 \leq j \leq cc+3\}\end{aligned}$$

- The environment alphabet is $\Sigma = \Gamma \setminus \{d_0\}$
- Each cycle to evolve from a real network configuration to the next one involves 15 computational steps, so $T = 15 \cdot Cycles$, where $Cycles$ is the number of cycles to simulate.
- $\mu = [\ \ ]_1$ is the membrane structure.
- The initial multisets are:
  - $\mathcal{M}_{g_{k,1}} = \{\ {a_1}^{g_{k,3}}, {a_0}^{1-g_{k,3},go} : 1 \leq k \leq ng\}$.
  - $\mathcal{M}_{ng+t_{i,1}} = \{\ binop_{t_{i,2}} : 1 \leq i \leq nb\}$.
  - $\mathcal{M}_{ng+nb+unt_{i,1}} = \{\ unop_{unt_{i,2}} : 1 \leq i \leq nu\}$.
- The rules of R and R_E to apply are showed below. They are put together to follow the sequential order of execution. Environment rules start with re and skeleton rules start with rs.
  - $rs_{1,i} \equiv go\, a_i[\,]_1 \longrightarrow c_i {b_i}^{max*i} {b_0}^{threshold} clock_0[\,]_1 : 0 \leq i \leq 1$
  - For each source gene environment:

$$\begin{aligned}re_{2,i,j,k} \equiv {} & (c_i \longrightarrow \{auxDest_{i,g_{j,1},k} : \{1 \leq k \leq nb+nu\}\})_{g_{j,1}} \\ & : 0 \leq i \leq 1, 1 \leq j \leq ng \\ re_{3,i,j,k} \equiv {} & (auxDest_{i,g_{j,1},k} \longrightarrow dest_{i,g_{j,1},t_{k,1}+ng})_{g_{j,1}} \\ & : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb \\ re_{4,i,j,k} \equiv {} & (auxDest_{i,g_{j,1},k} \longrightarrow dest_{i,g_{j,1},untk-nb,1+ng+nb})_{g_{j,1}} \\ & : 0 \leq i \leq 1, 1 \leq j \leq ng, nb+1 \leq k \leq nb+nu \\ re_{5,i,k} \equiv {} & (dest_{i,t_{k,3},t_{k,1}+ng} \longrightarrow e_{t_{k,4}*i+(1-i)*(1-t_{k,4}),t_{k,1}+ng})_{t_{k,3}} \\ & : 0 \leq i \leq 1, 1 \leq k \leq nb \\ re_{6,i,k} \equiv {} & (dest_{i,t_{k,5},t_{k,1}+ng} \longrightarrow e_{t_{k,6}*i+(1-i)*(1-t_{k,6}),t_{k,1}+ng})_{t_{k,5}} \\ & : 0 \leq i \leq 1, 1 \leq k \leq nb \\ re_{7,i,k} \equiv {} & (dest_{i,unt_{k-nb,3},unt_{k-nb,1}+ng+nb} \longrightarrow \\ & e_{unt_{k-nb,4}*i+(1-i)*(1-unt_{k-nb,4}),unt_{k-nb,1}+ng+nb})_{unt_{k-nb,3}} \\ & : 0 \leq i \leq 1, nb+1 \leq k \leq nb+nu\end{aligned}$$

$re_{8,i,k} \equiv \ (\quad)_{t_{k,1}+ng}(e_{i,t_{k,1}+ng})_{t_{k,3}} \longrightarrow (a_i)_{t_{k,1}+ng}(\quad)_{t_{k,3}}$
$: 0 \leq i \leq 1, 1 \leq k \leq nb$
$re_{9,i,k} \equiv \ (\quad)_{t_{k,1}+ng}(e_{i,t_{k,1}+ng})_{t_{k,5}} \longrightarrow (a_i)_{t_{k,1}+ng}(\quad)_{t_{k,5}}$
$: 0 \leq i \leq 1, 1 \leq k \leq nb$
$re_{10,i,k} \equiv (\quad)_{unt_{k-nb,1}+ng+nb}(e_{i,unt_{k-nb,1}+ng+nb})_{unt_{k-nb,3}} \longrightarrow$
$(a_i)_{unt_{k-nb,1}+ng+nb}(\quad)_{unt_{k-nb,3}}$
$: 0 \leq i \leq 1, nb+1 \leq k \leq nb+nu$

– Evaluation of the result of the interactions (1/2).

$rs_{11} \equiv binop_1\, {a_0}^2[]_1 \longrightarrow binop_1\, c_0[]_1$
$rs_{12} \equiv binop_1\, {a_1}^2[]_1 \longrightarrow binop_1\, c_1[]_1$
$rs_{13} \equiv binop_1\, a_1\, a_0[]_1 \longrightarrow binop_1\, c_1[]_1$
$rs_{14} \equiv binop_2\, {a_1}^2[]_1 \longrightarrow binop_2\, c_1[]_1$
$rs_{15} \equiv binop_2\, {a_0}^2[]_1 \longrightarrow binop_2\, c_0[]_1$
$rs_{16} \equiv binop_2\, a_1\, a_0[]_1 \longrightarrow binop_2\, c_0[]_1$
$rs_{17} \equiv binop_3\, {a_1}^2[]_1 \longrightarrow binop_3\, c_0[]_1$
$rs_{18} \equiv binop_3\, {a_0}^2[]_1 \longrightarrow binop_3\, c_0[]_1$
$rs_{19} \equiv binop_3\, a_1\, a_0[]_1 \longrightarrow binop_3\, c_1[]_1$
$rs_{20,i} \equiv unop_1\, a_i[]_1 \longrightarrow unop_1\, c_i[]_1 : 0 \leq i \leq 1$
$rs_{21,i} \equiv unop_2\, a_i[]_1 \longrightarrow unop_2\, c_{i-1}[]_1 : 0 \leq i \leq 1$
$rs_{22,i} \equiv unop_3\, a_i[]_1 \longrightarrow unop_3\, {c_i}^i[]_1 : 0 \leq i \leq 1$
$rs_{23,i} \equiv unop_4\, a_i[]_1 \longrightarrow unop_4\, {c_{1-i}}^i[]_1 : 0 \leq i \leq 1$

– Evaluation of the result of the interactions (2/2).

$re_{24,i,k} \equiv (c_i)_{t_{k,1}+ng}(\quad)_{t_{k,7}} \longrightarrow$
$(\quad)_{t_{k,1}+ng}(eF_{tk,8*i+(1-i)*(1-t_{k,8}),t_{k,1}+ng})_{t_{k,7}}$
$: 0 \leq i \leq 1, 1 \leq k \leq nb$
$re_{25,i,k} \equiv (c_i)_{unt_{k,1}+ng+nb}(\quad)_{unt_{k,5}} \longrightarrow$
$(\quad)_{unt_{k,1}+ng+nb}(eF_{i,(untk,1+ng+nb)})_{unt_{k,5}}$
$: 0 \leq i \leq 1, 1 \leq k \leq nu$

– Calculation of contributions.

$rs_{26,i,k} \equiv eF_{i,(t_{k,1}+ng)}[]_1 \longrightarrow {b_i}^{t_{k,9}}[]_1 : 0 \leq i \leq 1, 1 \leq k \leq nb$
$rs_{27,i,k} \equiv eF_{i,(unt_{k,1}+ng+nb)}[]_1 \longrightarrow {b_i}^{unt_{k,6}}[]_1 : 0 \leq i \leq 1, 1 \leq k \leq nu$

– Elimination of different–signed contributions.

$rs_{28} \equiv \ b_1\, b_0[]_1 \longrightarrow []_1$
$rs_{29,i} \equiv clock_{i-1}[]_1 \longrightarrow clock_i[]_1 : 1 \leq i \leq cc+3$

– Calculation of the next gene state.

$rs_{30} \equiv b_0[]_1^- \longrightarrow [d_0]_1^-$
$rs_{31} \equiv b_1[]_1^- \longrightarrow []_1^-$
$rs_{32,i,j,k} \equiv dest_{i,j,t_{k,1}+ng}[]_1^- \longrightarrow []_1^- : 0 \leq i \leq 1, 1 \leq j \leq ng, 1 \leq k \leq nb$
$rs_{33,i,j,k} \equiv dest_{i,j,unt_{k-nb,1}+ng+nb}[]_1^- \longrightarrow []_1^-$
$: 0 \leq i \leq 1, 1 \leq j \leq ng, nb+1 \leq k \leq nb+nu$
$rs_{34} \equiv [d_0]_1^- \longrightarrow []_1^+$
$rs_{35} \equiv clock_{cc+3}[]_1^+ \longrightarrow go\, a_0[]_1^0$
$rs_{36} \equiv clock_{cc+3}[]_1^- \longrightarrow go\, a_1[]_1^0$

In this section, only input parameters are described. This way, details about the model dynamics are left aside. These parameters are described in table 1.

Table 1. Parameters

Parameter	Description
	General parameters for the system
ng	Number of genes in the network
nb	Number of binary interactions
nu	Number of unary interactions
$threshold$	Maximum strength for an interaction
cc	Clock control
	Gene configuration parameters
$g_{i,1}$	Gene number (id)
$g_{i,3}$	Initial state of the gene
	Binary interactions parameters
$t_{i,1}$	Binary interaction number (id)
$t_{i,2}$	Interaction type (or: 1, and: 2, xor: 3)
$t_{i,3}$	1^{st} source gene number (id)
$t_{i,4}$	1^{st} source gene contribution (positive: 1, negative: 0)
$t_{i,5}$	2^{nd} source gene number (id)
$t_{i,6}$	2^{nd} source gene contribution (positive: 1, negative: 0)
$t_{i,7}$	Destination gene number (id)
$t_{i,8}$	Influence over destination gene (positive: 1, negative: 0)
$t_{i,9}$	Strength of the destination
	Unary interactions parameters
$unt_{i,1}$	Unary interaction number (id)
$unt_{i,2}$	Interaction type (strong promotion: 1, inhibition: 2; weak ones: 3, 4)
$unt_{i,3}$	Source gene number (id)
$unt_{i,4}$	Source gene contribution (positive, negative)
$unt_{i,5}$	Destination gene number (id)
$unt_{i,6}$	Influence over destination gene (positive, negative)

2.3 Model output

The state of the network is encoded as the multiplicity of objects a_1 and a_0 in each gene environment. The presence of objects a_1 inside a gene environment represents that its gene is active (a_0 for inactive). Due to the nature of the system, membrane genes cannot have objects a_1 and a_0 simultaneously. Therefore, to know the final state of the network, it suffices to identify which environments contain object a_1 and which ones a_0 at configuration T.

3 Modelling and simulation on MeCoSim

This section explains some relevant issues concerning the software environment, putting the focus on the needed changes in P-Lingua framework and the configuration of MeCoSim. The P–Lingua definition used to analyse the PDP model adheres to P–Lingua version 4 standard, available at [10].

3.1 Custom interface in MeCoSim

In our previous work [16], a P–Lingua model for the family of logic networks based on PDP systems has been extensively described. This model contains a number of parameters representing relevant information about each specific scenario. Thus, although a general model has been presented, a mechanism to ease the task of introducing the specific data for each scenario is needed. This task is performed through the software environment provided by MeCoSim [14, 12]. MeCoSim permits the definition of a custom visual simulator. This simulator includes an interface with the needed inputs, outputs, and a way to translate the input data into parameters for the model. The simulation engine is provided by pLinguaCore, available at [10]. The most relevant facts of this process are listed below.

Definition of a custom visual simulator for Logic Networks

Here, the process for defining a custom simulator based on MeCoSim is provided. This process is very simple, and consists of the following steps:

Configuration file: The first step is to define a spreadsheet file containing the configuration for the definition of visual tabs, input tables, output tables and charts, and the mechanism to generate both model parameters from input tables and outputs from the simulation results. The contents of the simulation parameters tab in the file is shown in figure 1. The configuration file is available by contacting the authors.

Param Name	Param Value	Index 1	Index 2
ng	<4,1,1>		
nb	<4,1,2>		
nu	<4,1,3>		
max	<4,1,4>		
cc	<4,1,5>		
g	<5,1,2>	[1..ng]	[1..2]
t	<2,1,2>	[1..nb]	[1..<@c,2>]
unt	<3,1,2>	[1..nu]	[1..<@c,3>]

Fig. 1. MeCoSim configuration file. Simulation params

Loading configuration file on MeCoSim: That file is loaded through the main window in MeCoSim by clicking the "Load config file" button, choosing the file, selecting "Update all information" option and pressing "Update config info" button. After these steps, the configuration file is loaded, so the custom simulator is ready to use. Finally, the message "The Application has been successfully initialized" is prompted in MeCoSim main display.

Running custom simulator: The newly configured simulator is ready to use by selecting "Gene network" application and pressing "Run Application" button. Then, the custom interface is visualized, enabling the user to load the model (*.pli* file) and enter the input data for a specific scenario, as shown in figure 2.

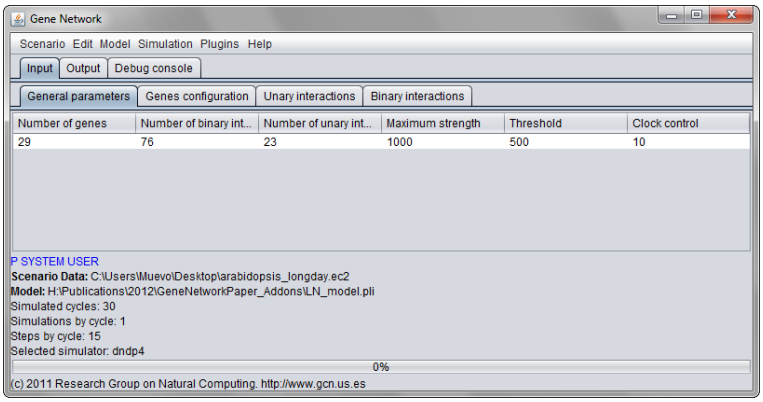


Fig. 2. MeCoSim window

3.2 Simulation methodology

In [16], we describe a methodology to simulate LN DP systems on MeCoSim. This methodology can be summarized in the following steps:

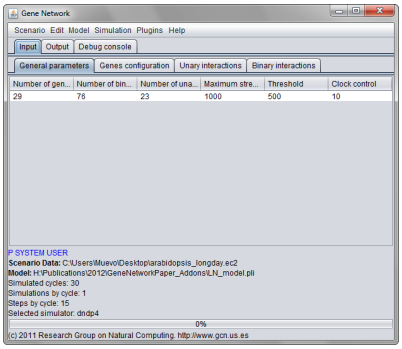
- Load the model specification by clicking on `Model > Set model`.
- Fill in the input tables in tab `Input`. Optionally, it is possible to save this data by clicking on `Scenario > Save`. This data can be loaded later by clicking on `Scenario > Open`.
- Set the number of steps on `Simulation > Number of steps`.
- Click on `Simulation > Simulate!`.
- Visualize the results in tab `Output`.

A toy example on a 3–gene logic network is provided in [16]. This network is taken from [15]. In this network, interactions have no associated weights. Hence, we presume all interactions to have the same weight (say 100). Although interaction scoring based on Pearson correlation coefficient is a rather widespread metric for measuring gene interaction strength [9], there is little literature on LNs, thus making it hard to find LN toy examples.

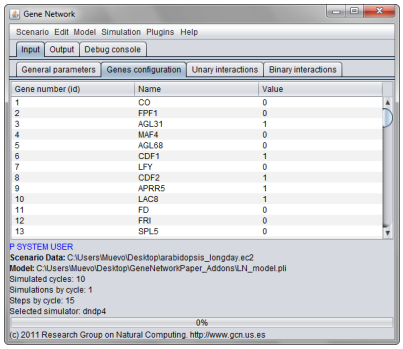
4 A case study on *Arabidopsis thaliana*

In order to experimentally verify our model, we have tested our algorithm by using a logic network which regulates flowering processes associated to *Arabidopsis thaliana* on a long day scenario. This relatively large network integrates gene interaction samples from NCBI/EBI database [5]. This logic network has been constructed according to the procedure described by Bowers *et al.* [2]. *A. thaliana* is a species widely used in genetic and protein interaction networks. The total number of genes in the network is 29, whereas the total number of interactions is 99. These interactions consist of 23 unary interactions and 76 binary interactions. We notice that only a few different types of all possible interactions are present in this network. In the case of unary interactions only strong promotions and strong inhibitions are present. When it comes to binary interactions, only *AND*–like and *OR*–like interactions are present. As regards to the distribution of the present interactions, the vast majority of them are *AND*–like interactions with both inputs in non–negated form (that is, $G'_j = G_j$ and $G'_k = G_k$), as well as non–negated result ($G'_l = G_l$).
Gene network data is provided as an appendix in section 7. Specifically, gene initial states are reflected in table 5. Unary gene interactions are reflected in figure 6. Similarly, binary gene interactions are reflected in figures 7, and 8. Figure 3 displays the MeCoSim input tables used in this case study.

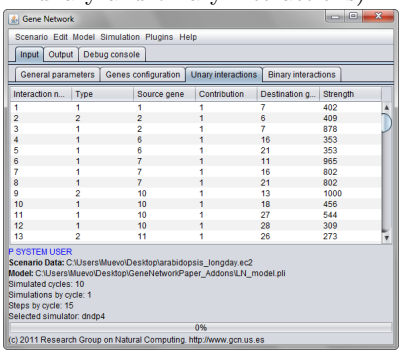
Eventually, we have simulated the corresponding P system for the *A. thaliana* network entered. The improved LAPP method (as presented in Wang et al. [18]) has been run for 30 steps on this data. Similarly, the LN DP model has been simulated for 30 cycles. As each cycle in an LN DP system consists of 15 computation steps, the total number of steps simulated in the model is $30 \times 15 = 450$. The results (see figure 4) match the ones obtained from the execution of the improved LAPP method on the same input data. Therefore, it is verified that, on this gene network and scenario, the P system model behaviour is analogous to that from the improved LAPP method.



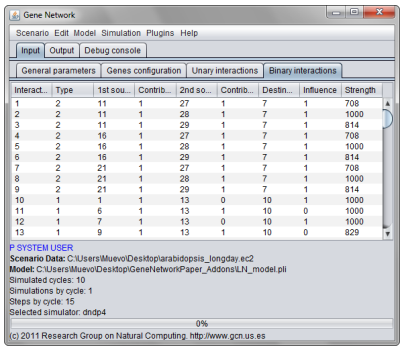
General parameters (that is, number of unary and binary interactions)



Initial state of each gene (active or inactive)



Unary interactions



Binary interactions

Fig. 3. Arabidopsis - MeCoSim Interface - Input Data

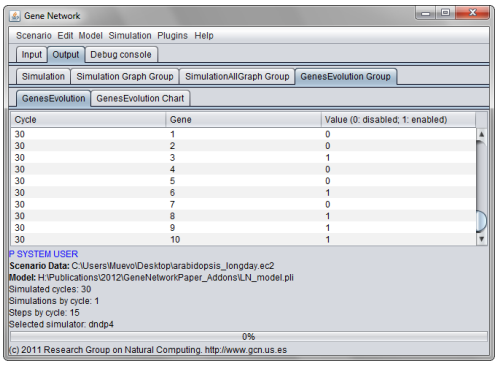


Fig. 4. Final gene states used for the simulation on MeCoSim interface

5 Conclusions

In this work, we have presented a case study on LN DP systems for a gene network which regulates the flowering process of *Arabidopsis thaliana*. We suplement this case study with a guide for generating a custom MeCoSim simulator for LN DP systems. In the case study, we validate the model against the improved LAPP method [18]. We conclude that our Membrane Computing model matches the output data obtained by the latter algorithm.
As a future work, it would be interesting to apply this model to gene networks with different biological functions, so as to test if the model matches the improved LAPP algorithm for a sufficiently representative number and variety of cases. This task can be complemented with a comparative study of the improved LAPP algorithm and different biochemical simulation methods (such as the Gillespie algorithm [6]) by means of Membrane Computing models.

6 Acknowledgements

Luis Valencia–Cabrera, Manuel García-Quismondo and Mario J. Pérez-Jiménez are supported by project TIN2012-37434 from "Ministerio de Ciencia e Innovación" of Spain, and "Proyecto de Excelencia con Investigador de Reconocida Valía P08-TIC-04200" from Junta de Andalucía, both co-financed by FEDER funds. Manuel García-Quismondo is also supported by the National FPU Grant Programme from the Spanish Ministry of Education.

References

1. Mitsutomo Abe, Yasushi Kobayashi, Sumiko Yamamoto, Yasufumi Daimon, Ayako Yamaguchi, Yoko Ikeda, Harutaka Ichinoki, Michitaka Notaguchi, Koji Goto, and Takashi Araki. (2005). Fd, a bzip protein mediating signals from the floral pathway integrator ft at the shoot apex. *Science*, 309(5737):1052–1056.
2. Peter M. Bowers, Shawn J. Cokus, Todd O. Yeates, and David Eisenberg. (2004). Use of logic relationships to decipher protein network organization. *Science*, 5705(306):2246–2249.
3. Peter M. Bowers, Brian D. O'Connor, Shawn J. Cokus, Eniat Sprinzak, Todd O. Yeates, and David Eisenberg. (2005). Utilizing logical relationships in genomic data to decipher cellular processes. *the FEBS journal*, 272(1):5110–5118.
4. Miguel A. Martínez del Amor, Ignacio Pérez-Hurtado, Manuel García-Quismondo, Luis F. Macías-Ramos, Luis Valencia-Cabrera, Álvaro Romero-Jiménez, Carmen Graciani-Díaz, Agustín Riscos-Núñez, M. Angels Colomer, and Mario J. Pérez-Jiménez. (2013). Dcba: Simulating population dynamics p systems with proportional objects distribution. *Lecture Notes in Computer Science*, pages 257–276.
5. National Center for Biotechnology Information, (March 2013). http://www.ncbi.nlm.nih.gov/.

6. Daniel T. Gillespie. (1977). Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361.
7. Takato Imaizumi, Thomas F. Schultz, Frank G. Harmon, Lindsey A. Ho, and Steve A. Kay. (2005). Fkf1 f-box protein mediates cyclic degradation of a repressor of constans in arabidopsis. *Science*, 309(5732):293–297.
8. Takato Imaizumi, Hien G. Tran, Trevor E. Swartz, Winslow R. Briggs, and Steve A. Kay. (2003). Fkf1 is essential for photoperiodic-specific light signaling in arabidopsis. *Nature*, 426:301–309. 10.1007/s11424-010-0205-0.
9. Bolan Linghu, EricA. Franzosa, and Yu Xia. (2013). Construction of functional linkage gene networks by data integration. In Hiroshi Mamitsuka, Charles DeLisi, and Minoru Kanehisa, editors, *Data Mining for Systems Biology*, volume 939 of *Methods in Molecular Biology*, pages 215–232. Humana Press.
10. P Lingua Web Page, (February 2009). http://www.p-lingua.org.
11. Miguel A. Martínez-del Amor, Ignacio Pérez-Hurtado, Adolfo Gastalver-Rubio, Anne C. Elster, and Mario J. Pérez-Jiménez. (2012). Population dynamics p systems on cuda. In David Gilbert and Monika Heiner, editors, *Computational Methods in Systems Biology*, Lecture Notes in Computer Science, pages 247–266. Springer Berlin Heidelberg.
12. MeCoSim Web Page, (July 2010). http://www.p-lingua.org/mecosim.
13. Gheorghe Paun, Grzegorz Rozenberg, and Arto Salomaa. (2010). *The Oxford Handbook of Membrane Computing*. Oxford University Press, Inc., New York, NY, USA.
14. Ignacio Pérez-Hurtado, Luis Valencia-Cabrera, Mario J. Pérez-Jiménez, M. A. Colomer, and Agustín Riscos-Núñez. (2010). MeCoSim: a general purpose software tool for simulating biological phenomena by means of P systems. *IEEE Fifth International Conference on Bio-inpired Computing: Theories and Applications (BIC-TA 2010)*, I:637–643.
15. Thomas Schlitt and Alvis Brazma. (2007). Current approaches to gene regulatory network modelling. *BMC Bioinformatics*, 8(Suppl 6):S9.
16. Luis Valencia-Cabrera, Manuel García-Quismondo, Yansen Su, Mario J. Perez-Jimenez, Hui Yu, and Linqiang Pan. (2013). Modeling logic gene networks by means of probabilistic dynamic p systems, (accepted publication). *International Journal of Unconventional Computing*.
17. Shudong Wang, Yan Chen, Qingyun Wang, Eryan Li, Yansen Su, and Dazhi Meng. (2006). Analysis for stimuli to shoot genes of arabidopsis thaliana based on logical relationships. *Optimization and Systems Biology*, 11(-):435–447.
18. Shudong Wang, Yan Chen, Qingyun Wang, Eryan Li, Yansen Su, and Dazhi Meng. (2010). Analysis for gene networks based on logic relationships. *Journal of Systems Science and Complexity*, 23:999–1011. 10.1007/s11424-010-0205-0.
19. Su-Zhil Zhang and Zuo Jian-Rhu. (2006). Advance in the flowering time control of arabidopsis. *Progress in Biochemistry and Biophysics*, 33:301–309.
20. Xin Zhang, Seungchan Kim, Tie Wang, and C. Baral. (October 2006). Joint learning of logic relationships for studying protein function using phylogenetic profiles and the rosetta stone method. *Trans. Sig. Proc.*, 54(6):2427–2435.

7 Appendix A: Gene Network Data

Gene number	Initial state
1	0
2	0
3	1
4	0
5	0
6	1
7	0
8	1
9	1
10	1
11	0
12	0
13	0
14	1
15	1

Gene number	Initial state
16	0
17	1
18	1
19	1
20	0
21	0
22	1
23	1
24	0
25	1
26	0
27	1
28	1
29	1

Fig. 5. Initial gene states in the *Arabidosis thaliana* gene network on the longday scenario taken as case study

ID	Logic	Weight
1	$g_1 \rightarrow g_7$	0.402
2	$g_2 \rightarrow \neg g_6$	0.409
3	$g_2 \rightarrow g_7$	0.878
4	$g_6 \rightarrow g_{16}$	0.353
5	$g_6 \rightarrow g_{21}$	0.353
6	$g_7 \rightarrow g_{11}$	0.965
7	$g_7 \rightarrow g_{16}$	0.802
8	$g_7 \rightarrow g_{21}$	0.802
9	$g_{10} \rightarrow \neg g_{13}$	0.1000
10	$g_{10} \rightarrow g_{18}$	0.456
11	$g_{10} \rightarrow g_{27}$	0.544
12	$g_{10} \rightarrow g_{28}$	0.309

ID	Logic	Weight
13	$g_{11} \rightarrow \neg g_{26}$	0.273
14	$g_{12} \rightarrow g_{16}$	0.282
15	$g_{12} \rightarrow g_{21}$	0.282
16	$g_{16} \rightarrow \neg g_{29}$	0.713
17	$g_{17} \rightarrow g_{24}$	0.425
18	$g_{17} \rightarrow g_{26}$	0.389
19	$g_{19} \rightarrow g_{29}$	0.551
20	$g_{20} \rightarrow \neg g_{22}$	0.303
21	$g_{21} \rightarrow \neg g_{29}$	0.713
22	$g_{22} \rightarrow g_{26}$	0.439
23	$g_{28} \rightarrow g_{29}$	0.292

Fig. 6. Unary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study

ID	Logic	Weight
1	$g_{11} \wedge g_{27} \rightarrow g_7$	0.708
2	$g_{11} \wedge g_{28} \rightarrow g_7$	1
3	$g_{11} \wedge g_{29} \rightarrow g_7$	0.814
4	$g_{16} \wedge g_{27} \rightarrow g_7$	0.708
5	$g_{16} \wedge g_{28} \rightarrow g_7$	1
6	$g_{16} \wedge g_{29} \rightarrow g_7$	0.814
7	$g_{21} \wedge g_{27} \rightarrow g_7$	0.708
8	$g_{21} \wedge g_{28} \rightarrow g_7$	1
9	$g_{21} \wedge g_{29} \rightarrow g_7$	0.814
10	$g_1 \vee \neg g_{13} \rightarrow g_{10}$	1
11	$g_6 \wedge g_{13} \rightarrow \neg g_{10}$	1
12	$g_7 \vee \neg g_{13} \rightarrow g_{10}$	1
13	$g_9 \wedge g_{13} \rightarrow \neg g_{10}$	0.829
14	$g_{11} \vee \neg g_{13} \rightarrow g_{10}$	1
15	$g_{12} \vee \neg g_{13} \rightarrow g_{10}$	0.829
16	$\neg g_{13} \vee g_{16} \rightarrow g_{10}$	1
17	$\neg g_{13} \vee g_{18} \rightarrow g_{10}$	0.728
18	$g_{13} \wedge g_{19} \rightarrow \neg g_{10}$	0.829
19	$\neg g_{13} \vee g_{21} \rightarrow g_{10}$	1
20	$\neg g_{13} \vee g_{27} \rightarrow g_{10}$	1
21	$g_{27} \vee \neg g_{28} \rightarrow g_{10}$	0.728
22	$g_{10} \wedge g_{16} \rightarrow g_{11}$	0.741
23	$g_{10} \wedge g_{21} \rightarrow g_{11}$	0.741
24	$g_{14} \wedge g_{16} \rightarrow g_{11}$	0.741
25	$g_{14} \wedge g_{21} \rightarrow g_{11}$	0.741
26	$g_{15} \wedge g_{16} \rightarrow g_{11}$	0.741
27	$g_{15} \wedge g_{21} \rightarrow g_{11}$	0.741
28	$g_{16} \wedge g_{17} \rightarrow g_{11}$	0.741
29	$g_{16} \wedge \neg g_{20} \rightarrow g_{11}$	0.741
30	$g_{16} \wedge g_{21} \rightarrow g_{11}$	0.741

ID	Logic	Weight
31	$g_{16} \wedge g_{22} \rightarrow g_{11}$	0.741
32	$g_{16} \wedge g_{23} \rightarrow g_{11}$	0.741
33	$g_{16} \wedge \neg g_{24} \rightarrow g_{11}$	0.741
34	$g_{16} \wedge g_{25} \rightarrow g_{11}$	0.741
35	$g_{16} \wedge \neg g_{26} \rightarrow g_{11}$	0.741
36	$g_{16} \vee g_{29} \rightarrow g_{11}$	0.741
37	$g_{17} \wedge g_{21} \rightarrow g_{11}$	0.741
38	$\neg g_{20} \wedge g_{21} \rightarrow g_{11}$	0.741
39	$g_{21} \wedge g_{22} \rightarrow g_{11}$	0.741
40	$g_{21} \wedge g_{23} \rightarrow g_{11}$	0.741
41	$g_{21} \wedge \neg g_{24} \rightarrow g_{11}$	0.741
42	$g_{21} \wedge g_{25} \rightarrow g_{11}$	0.741
43	$g_{21} \wedge \neg g_{26} \rightarrow g_{11}$	0.741
44	$g_{21} \vee \neg g_{29} \rightarrow g_{11}$	0.741
45	$g_8 \wedge g_{21} \rightarrow g_{16}$	0.801
46	$g_{10} \wedge g_{21} \rightarrow g_{16}$	1
47	$g_{11} \vee g_{21} \rightarrow g_{16}$	1
48	$g_{11} \vee \neg g_{29} \rightarrow g_{16}$	1
49	$g_{14} \wedge \neg g_{19} \rightarrow g_{16}$	0.801
50	$g_{14} \wedge g_{21} \rightarrow g_{16}$	1
51	$g_{15} \wedge g_{21} \rightarrow g_{16}$	1
52	$g_{17} \wedge g_{21} \rightarrow g_{16}$	1
53	$\neg g_{19} \wedge g_{21} \rightarrow g_{16}$	0.801
54	$\neg g_{20} \wedge g_{21} \rightarrow g_{16}$	1
55	$g_{21} \wedge g_{22} \rightarrow g_{16}$	1
56	$g_{21} \wedge g_{23} \rightarrow g_{16}$	1
57	$g_{21} \wedge \neg g_{24} \rightarrow g_{16}$	1
58	$g_{21} \wedge g_{25} \rightarrow g_{16}$	1
59	$g_{21} \wedge \neg g_{26} \rightarrow g_{16}$	1
60	$g_{21} \vee \neg g_{29} \rightarrow g_{16}$	1

Fig. 7. Binary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study (1/2)

ID	Logic	Weight
61	$g_8 \wedge g_{16} \rightarrow g_{21}$	0.801
62	$g_{10} \wedge g_{16} \rightarrow g_{21}$	1
63	$g_{11} \vee g_{16} \rightarrow g_{21}$	1
64	$g_{11} \vee \neg g_{29} \rightarrow g_{21}$	1
65	$g_{14} \wedge g_{16} \rightarrow g_{21}$	1
66	$g_{14} \wedge \neg g_{19} \rightarrow g_{21}$	0.801
67	$g_{15} \wedge g_{16} \rightarrow g_{21}$	1
68	$g_{16} \wedge g_{17} \rightarrow g_{21}$	1
69	$g_{16} \wedge \neg g_{19} \rightarrow g_{21}$	0.801
70	$g_{16} \wedge \neg g_{20} \rightarrow g_{21}$	1
71	$g_{16} \wedge g_{22} \rightarrow g_{21}$	1
72	$g_{16} \wedge g_{23} \rightarrow g_{21}$	1
73	$g_{16} \wedge \neg g_{24} \rightarrow g_{21}$	1
74	$g_{16} \wedge g_{25} \rightarrow g_{21}$	1
75	$g_{16} \wedge \neg g_{26} \rightarrow g_{21}$	1
76	$g_{16} \vee \neg g_{29} \rightarrow g_{21}$	1

Fig. 8. Binary gene interactions present in the logic network associated to the behaviour of *Arabidosis thaliana* taken as case study (2/2)

Gene number	Initial state
1	0
2	0
3	1
4	0
5	0
6	1
7	0
8	1
9	1
10	1
11	0
12	0
13	0
14	1
15	1
16	0
17	1
18	1
19	1
20	0
21	0
22	1
23	1
24	0
25	1
26	1
27	1
28	1
29	1

Fig. 9. Final gene states in the *Arabidosis thaliana* gene network on the longday scenario taken as case study

Author Index