Seventeenth Brainstorming Week on Membrane Computing

Sevilla, February 5 – 8, 2019

David Orellana-Martín
Gheorghe Păun
Agustín Riscos-Núñez
José A. Andreu-Guzmán
Editors

Seventeenth Brainstorming Week on Membrane Computing

Sevilla, February 5 – 8, 2019

David Orellana-Martín
Gheorghe Păun
Agustín Riscos-Núñez
José A. Andreu-Guzmán
Editors

RGNC REPORT 1/2019

Research Group on Natural Computing
Universidad de Sevilla

Sevilla, 2019

ISBN: ??????

Printed by: Artes Gráficas Moreno, S.L.
`http://www.agmoreno.net/`

Preface

The Seventeenth Brainstorming Week on Membrane Computing (BWMC) was held in Sevilla, from February 5 to 8, 2019, hosted by the Research Group on Natural Computing (RGNC) from the Department of Computer Science and Artificial Intelligence of Universidad de Sevilla. The first edition of BWMC was organized at the beginning of February 2003 in Rovira i Virgili University, Tarragona, and all the next editions have been taking place in Sevilla since then, always at the beginning of February.

In the style of previous meetings in this series, 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, announcements, conjectures waiting for proofs, or ongoing research works in general (involving both theory and applications). Joint work sessions were scheduled on the afternoons to allow for collaboration among the about 25 participants – see the list in the end of this preface.

This year was a special year for the RGNC, since the head of the research group, Mario J. Pérez-Jiménez recently turned 70 years old. This edition was dedicated to him, and a special session of videos from former PhD students and researchers who have been supervised by him but could not attend the meeting was carried out.

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.

A selection of papers from this volume will be considered for publication in the new *Journal of Membrane Computing*, published by Springer-Verlag (`www.springer.com/41965`).

Other papers elaborated during the 2019 edition of 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. Artiom Alhazov, Institute of Mathematics and Computer Science of Academy of Sciences of Moldova, Moldova
   aartiom@yahoo.com
2. José A. Andreu Guzmán, Universidad de Sevilla, Spain
   andreuguzman36@gmail.com
3. Lúdek Cienciala, Silesian University in Opava, Czech Republic
   ludek.cienciala@fpf.slu.cz
4. Lucie Ciencialová, Silesian University in Opava, Czech Republic
   lucie.ciencialova@fpf.slu.cz
5. Erzsébet Csuhaj-Varjú, Eötvös Loránd University, Hungary
   csuhaj@inf.elte.hu
6. Jan Drastik, Silesian University in Opava, Czech Republic
   honza.drastik@gmail.com
7. Rudolf Freund, Technological University of Vienna, Austria
   rudi@emcc.at
8. Zsolt Gazdag, University of Szeged, Hungary
   gazdag@inf.u-szeged.hu
9. Péter Battyányi, University of Debrecen, Hungary
   battyanti.peter@inf.unideb.hu
10. Carmen Graciani, Universidad de Sevilla, Spain
    cgdiaz@us.es
11. Sergiu Ivanov, IBISC, Université Évry, Université Paris-Saclay, France
    sergiu.ivanov@univ-evry.fr
12. Pramod Kumar Sethy, Eötvös Loránd University, Hungary
    pksethy@inf.elte.hu
13. Gábor Kolonits, Eötvös Loránd University, Hungary
    kolomax@inf.elte.hu
14. Alberto Leporati, University of Milano-Bicocca, Italy
    leporati@disco.unimib.it
15. Luca Manzoni, University of Milano-Bicocca, Italy
    luca.manzoni@disco.unimib.it
16. Miguel A. Martínez-del-Amor, Universidad de Sevilla, Spain
    mdelamor@us.es
17. David Orellana-Martín, Universidad de Sevilla, Spain
    dorellana@us.es
18. Ignacio Pérez-Hurtado, Universidad de Sevilla, Spain
    perezh@us.es

19. Mario de J. Pérez-Jiménez, Universidad de Sevilla, Spain
marper@us.es
20. Agustín Riscos-Núñez, Universidad de Sevilla, Spain
ariscosn@us.es
21. Álvaro Romero-Jiménez, Universidad de Sevilla, Spain
romero.alvaro@us.es
22. Zeyi Shang, University Southwest Jiaotong, China
zeyi.shang@lacl.fr
23. Vladimir Smolka, Silesian University in Opava, Czech Republic
v.smolka@outlook.cz
24. Ana Ţurlea, University of Bucharest, Romania
t_anacris@yahoo.com
25. Luis Valencia-Cabrera, Universidad de Sevilla, Spain
lvalencia@us.es
26. György Vaszil, University of Debrecen, Hungary
vaszil.gyorgy@inf.unideb.hu

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

The meeting was partially supported from various sources: (i) Research Project TIN2017-89842-P cofinanced by Ministerio de Economía, Industria y Competitividad (MINECO) of Spain, through the Agencia Estatal de Investigación (AEI), and by Fondo Europeo de Desarrollo Regional (FEDER) of the European Union, (ii) VI Plan Propio, *Vicerrectorado de Investigación de la Universidad de Sevilla*, and (iii) Department of Computer Science and Artificial Intelligence from *Universidad de Sevilla*.

The Editors
(Jun 2019)

Contents

Beyond Generalized Multiplicities: Register Machines over Groups

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

[1] Vladimir Andrunachievici Institute of
Mathematics and Computer Science
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md

[2] TU Wien, Institut für Logic and Computation
Favoritenstraße 9–11, 1040 Wien, Austria
E-mail: rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23, boulevard de France, 91034 Évry, France
E-mail: sergiu.ivanov@univ-evry.fr

Summary. Register machines are a classic model of computing, often seen as a canonical example of a device manipulating natural numbers. In this paper, we define register machines operating on general groups instead. This generalization follows the research direction started in multiple previous works. We study the expressive power of register machines as a function of the underlying groups, as well as of allowed ingredients (zero test, partial blindness, forbidden regions). We put forward a fundamental connection between register machines and vector addition systems. Finally, we show how registers over free groups can be used to store and manipulate strings.

1 Introduction

Register machines are traditionally seen as a model of computing manipulating non-negative numbers. However, quite some time ago integer numbers were already considered as the base set for register contents [8]. Such machines are traditionally called blind as long as they do not allow testing registers for zero, except eventually testing all registers for zero at the end. The computational power of such blind register machines is inferior to that of "conventional" register machines over natural numbers [2]. If the register machine is not allowed to go below zero, but can neither explicitly test its registers for zero, it is called partially blind.

Even further, we need not restrict the definition of the model to numbers. For example, Section 3 of [2] gives a very general definition of register machines whose registers may contain elements of any set A. However, going this far up the abstraction scale loses too much structure: almost nothing can be said about such general

constructs. In this paper, we focus on a level of abstraction which is in between the two: we consider register machines over finitely presented groups. This generalization comes in as a natural sequel to multiple previous works. For example, [9] introduced integer vector addition systems by lifting the traditional restriction on the vectors to only contain non-negative components. Subsequently, [7] generalized P systems (compartmentalized multiset rewriting systems [18]) to allow multiplicities of objects to come from Abelian groups instead of just natural numbers. Finally, papers [2, 3] come back on the less general case of integer multiplicities and show a number of new properties of integer vector addition systems and blind register machines.

As almost any work on register machines, studies on register machines over groups have multiple interesting consequences for P systems. The present paper lays the ground for further exploration of P systems with generalized multisets and raises a number of important questions, for example, about the ways in which multiplicities from non-commutative groups can be interpreted. As we will show later, registers containing elements of the free group can be used to emulate strings; what would be the meaning of string multiplicities in P systems?

In this work, we define register machines over arbitrary finite families of groups, with or without zero test, as well as partially blind register machines, and register machines with forbidden regions (Section 3). Each of these ingredients is meant to generalize individual features which appear in the classic definition of register machines. We then study the computational power of the variants we define: we compare the generating and the accepting modes, single- and multi-register machines, vector addition systems with and without states (Section 4).

As it often happens, all of the results we present in this paper originate from the fruitful discussions the team of authors had during the Brainstorming Week on Membrane Computing 2019 in Seville, Spain. Even though this work is not explicitly situated within the domain of membrane computing, we believe that it may have an important influence on the study of generalized multiplicities in P systems. We would therefore like to thank the organizing team for giving us the opportunity to work on these exciting topics.

2 Preliminaries

In this paper, we use the symbols $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{N}$ to refer to the set of real numbers, integer numbers, and the set of natural numbers including 0.

For an alphabet V, by V^* we denote the free monoid generated from the elements of V under the operation of concatenation, i.e., containing all possible strings over V. The *empty string* is denoted by λ. The family of all recursively enumerable sets of strings is denoted by RE, the corresponding family of recursively enumerable sets of Parikh sets (vectors of natural numbers) and of number sets is denoted by $PsRE$ and NRE, respectively. For an extensive introduction to the theory of formal languages, we recommend [18, 19].

Given a set A, a total function $f : A \times A \to A$ is called a *binary operation* over A. We will use the infix notation afb to refer to $f(a, b)$, for $a, b \in A$. A relation over a set A is any subset $R \subseteq A \times A$. As for binary operations, we will also use the infix notation aRb for $(a, b) \in R$.

A relation $\leq \subseteq A \times A$ is a called a *total order* if the following statements hold for every three elements $a, b, c \in A$:

- *antisymmetry*: if $a \leq b$ and $b \leq a$ then $a = b$,
- *transitivity*: if $a \leq b$ and $b \leq c$ then $a \leq c$,
- *totality*: either $a \leq b$ or $b \leq a$.

For $a, b \in A$ and a total order $\leq$ on A, we will sometimes write $b \geq a$ as equivalent to $a \leq b$, and use $a < b$ ($a > b$) to denote that $a \leq b$ and $a \neq b$ ($a \geq b$ and $a \neq b$).

2.1 Groups and Group Presentations

Groups

A group is the structure $G = (G', \circ)$ where G' is the set of elements (the underlying set) and $\circ : G' \times G' \to G'$ a binary operation over G' satisfying the following properties (*group axioms*):

- *closure*: for any $a, b \in G'$, $a \circ b \in G'$,
- *associativity*: for any $a, b, c \in G'$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- *identity*: there exists a (unique) element $e \in G'$, called the *identity*, such that $e \circ a = a \circ e = a$ for all $a \in G'$, and
- *invertibility:* for any $a \in G'$, there exists a (unique) element a^{-1}, called the *inverse* of a, such that $a \circ a^{-1} = a^{-1} \circ a = e$.

The group G is called *commutative* or *Abelian*, if for any $a, b \in G'$, $a \circ b = b \circ a$.

A *subgroup* of the group $(G, \circ)$ is any group $(H, \circ)$ with $H \subseteq G$ and the same group operation $\circ$.

For any element $b \in G'$, the order of b is the smallest number $n \in \mathbb{N}$ such that $b^n = e$ provided such n exists, and then we write $ord\,(b) = n$. If no such n exists, $\{b^n \mid n \geq 1\}$ is an infinite subset of G' and we write $ord\,(b) = \infty$.

In the following, we will often use the same symbol G to refer both to a group and to its underlying set.

Representations of groups

The definitions and examples from group theory we exhibit now follow the exposition given in [1] and [2], based on the notions in [10]. In what follows, we will use strings for representing group elements.

For any set B, the set B^{-1} is defined to contain the symbols representing the "inverses" of the elements of B, i.e., $B^{-1} = \left\{b^{-1} \mid b \in B\right\}$. B (not containing the identity) is called a *generator set* of the group G if every element a from G can be

written as a finite product/sum of elements from $B \cup B^{-1}$, i.e., $a = b_1 \circ \cdots \circ b_m$ for $b_1, \ldots, b_m \in B \cup B^{-1}$. In this paper, we restrict ourselves to finitely presented groups, i.e., having a finite presentation $\langle B \mid R \rangle$ with B being a finite generator set and moreover, R being a finite set of relations among these generators. Informally, the group $G = \langle B \mid R \rangle$ is the largest one generated by B subject only to the group axioms and the relations in R. We will restrict ourselves to relations of the form $b_1 \circ \cdots \circ b_m = e$ with $b_1, \ldots, b_m \in B$; omitting the identity e we write $b_1 \circ \cdots \circ b_m$, which then is called *relator*.

Example 1. The free group $F(B) = (I(B), \circ)$ can be written as $\langle B \mid \emptyset \rangle$ (or even simpler as $\langle B \rangle$) because it has no restricting relations.

Example 2. The *cyclic group* of order n has the presentation $\langle \{a\} \mid \{a^n\} \rangle$ (or, omitting the set brackets, as $\langle a \mid a^n \rangle$). It is also known as $\mathbb{Z}_n$ or as the quotient group $\mathbb{Z}/n\mathbb{Z}$.

Example 3. $\mathbb{Z}$ is a special case of an Abelian group generated by 1 and its inverse -1, i.e., $\mathbb{Z}$ is the free group generated by $B = \{1\}$. $\mathbb{Z}^d$ is the Abelian group generated by the unit vectors $(0, \ldots, 1, \ldots, 0)$ and their inverses $(0, \ldots, -1, \ldots, 0)$. It is well known that every finitely generated Abelian group is a direct sum of a torsion group and a free Abelian group, where the torsion group may be written as a direct sum of finitely many quotient groups of the form $\mathbb{Z}/p^k\mathbb{Z}$, with p a prime and $k \in \mathbb{N}$, and the free Abelian group is a direct sum of finitely many copies of $\mathbb{Z}$.

Example 4. A very well-known example of a non-Abelian group is the hexagonal group with the finite presentation $\langle a, b, c \mid a^2, b^2, c^2, (abc)^2 \rangle$. The relators a^2, b^2, and c^2 indicate that all three generators a, b, and c are self-inverse.

Remark 1. In this paper, we will restrict ourselves to finitely generated groups, for which the word equivalence problem $u = v$ is decidable, i.e., there exists a decision procedure telling us whether $u \circ v^{-1} = e$ for two strings u and v. In this case, we call G *recursive* or *computable*. If the set of relators R in a presentation $\langle B \mid R \rangle$ of G is computable (recursive), we call this a computable (recursive) presentation. Clearly, any finitely presented group is computable.

A group $(G, +)$ in which the group operation can be interpreted as addition is called additive. For such groups, the inverse of $b \in G$ is often written as $-b$, the neutral element e as 0, and the sum $a + (-b)$ as $a - b$, whenever no ambiguity arises. Another kind of groups are multiplicative groups, in which the group operation can be thought of as multiplication. For such groups, the inverse of $b \in G$ is usually written as b^{-1}, and the group operation as multiplication: $a \cdot b$ or ab.

For Abelian groups, further shortcut notation is introduced to capture chained applications of the operation to a single element. Consider $z \in \mathbb{Z}$ and $a \in G$. The *scalar product* of a by z is defined as follows (using either additive or multiplicative notation):

$$za = \begin{cases} a^z = \sum_{i=1}^{z} a, & z > 0, \\ a^0 = 0 \text{ (group identity)}, & z = 0, \\ (-a)^{-z} = \sum_{i=1}^{z} (-a), & z < 0. \end{cases}$$

A *linearly* or *totally ordered group* is construct $(A, +, \leq)$ where $(A, +)$ is a group, $\leq \subseteq A \times A$ is a total order on A and, for any triple $a, b, c \in A$, the fact that $a \leq b$ implies that $c + a \leq c + b$ and $a + c \leq b + c$.

2.2 Register Machines

Register machines are well-known universal devices for computing (generating or accepting) sets of vectors of natural numbers. The article [13] is one of the reference works on the universality of register machines.

Definition 1. *A* register machine *is the construct* $M = (m, B, l_0, l_h, P)$*, where*

- m *is the number of registers,*
- B *is a set of labels bijectively labeling the instructions in the set* P*,*
- $l_0 \in B$ *is the initial label,*
- $l_h \in B$ *is the final label, and*
- P *is the set of instructions.*

The labeled instructions in P *can be of the following forms:*

- $p : (ADD(r), q, s)$*, with* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq m$*.*
  Increment the value of register r *and non-deterministically jump to instruction* q *or* s*.*
- $p : (SUB(r), q, s)$*, with* $p \in B \setminus \{l_h\}$*,* $q, s \in B$*,* $1 \leq r \leq m$*.*
  If the value of register r *is not zero then decrement the value of register* r *(*decrement *case) and jump to instruction* q*, otherwise jump to instruction* s *(*zero-test *case).*
- $l_h : HALT$*.*
  Stop the execution of the register machine.

A *configuration* of a register machine is the tuple $C = (q, r_1, \ldots, r_m)$, in which $r_1, \ldots, r_m$ are the values of the registers and q is the current instruction label which indicates which is the *next* instruction to execute. This label is often called the *state* of the machine. An n-step computation of a register machine is a sequence of configurations $(C_i)_{0 \leq i \leq n}$ in which the configuration C_{i+1} is obtained from C_i by applying to C_i the instruction given by the current instruction label of C_i. The first configuration C_0 of a computation is usually referred to as the *initial* configuration and its current instruction label must be l_0. If, in the last configuration C_n, the current instruction label is be l_h, C_n is called a *halting* configuration and the whole computation is called a halting computation.

A register machine M can be seen as an accepting device, a generating device, or as a device computing functions or relations. In the *accepting* case, the first k registers of M are designated as input registers, and are initialized to a k-vector of natural numbers v (the input vector). If there exists a halting computation of M starting with this initial configuration, then v is accepted by M. Without losing generality, we may only consider computations in which all registers are empty in the halting configuration.

On the other hand, in the *generating* case, a single initial configuration is fixed for all computations of M, the first k registers are designated as the output registers, and for every halting computation of M, the k-vector contained in the output registers in the halting configuration is said to be generated by M. Without losing generality, we may only consider those computations of M in which all registers with indices greater than k are empty in the halting configuration.

Finally, we can designate input *and* output registers (these may be disjoint) and see M as establishing a binary *relation* between the contents of the input registers in the initial configurations and the output registers in the halting configurations. If this relation is functional, i.e., M associates at most one output vector to any input vector, M can be seen as defining a *function*.

In this paper, we will only consider the accepting and the generating cases. We will denote by $\mathcal{L}_{acc}(M)$ (respectively, by $\mathcal{L}_{gen}(M)$) the set of input vectors accepted (respectively, generated) by the register machine M. Similarly, for a family $\mathcal{X}$ of register machines, we will denote by $\mathcal{L}_{acc}(\mathcal{X})$ (respectively, by $\mathcal{L}_{gen}(\mathcal{X})$) the family of sets of vectors accepted (respectively, generated) by the register machines in the family $\mathcal{X}$. We will use the same notations to denote the sets of languages accepted (respectively, generated) by any other computing device M or any other family of computing devices $\mathcal{X}$. In case the operating mode is fixed by the definition of the device (e.g., vector addition systems always generate), we omit the corresponding subscript.

We use the notation RM to refer to the family of register machines defined as above. It is folklore (e.g., see [16]) that $\mathcal{L}_{acc}(RM) = PsRE$. Similarly, register machines generate any recursively enumerable set of vectors of natural vectors, $\mathcal{L}_{gen}(RM) = PsRE$. A proof sketch: consider $L \in PsRE$, then build the machine M such that it first non-deterministically generates a vector, and then runs a sequence of instructions recognizing precisely the vectors in L.

Blind and Partially Blind Machines

Several papers consider weaker kinds of register machines: blind and partially blind register machines, for example, see [2, 6, 8].

In *partially blind* register machines, the SUB instruction has the form $p : (SUB(r), q)$: if the register r is not empty, it is decremented and the register machine moves to state q, otherwise the machine crashes—the computation stops in a non-halting configuration, yielding no result. In *blind* register machines, the regis-

ters are allowed to contain negative values, meaning that the decrement instruction always succeeds. However, valid computations of a blind machine are required to have 0 in all non-output registers in halting configurations. The definitions of blind and partially blind machines may vary from source to source: notably some sources define blind register machines as partially blind, but without the zero check at the end [6].

In this paper, we will give uniform definitions of various types of register machines.

A General Model for Register Machines

For the record, we recall here a very general definition of a register-machine-like device given in [3].

Definition 2. *A* register-machine-like device over the set A *is the tuple* $M_A = (m, A, B, l_0, l_h, P)$, *where*

- $m \in \mathbb{N}$ *is the number of registers,*
- A *is the set of values the registers may contain,*
- B *is a finite set of instruction labels,*
- l_0 *is the initial label,*
- l_h *is the final label,*
- P *is a mapping associating an instruction to every label in* B.

An instruction p *is a function* $p : A^m \to A^m \times 2^Q$ *associating to every* m*-tuple of values from* A *another* m*-tuple of such values and a set of new instruction labels from* B. *A configuration* $C \in B \times A^m$ *of* M_A *is a tuple combining an instruction label and the values of the* m *registers of* M_A.

2.3 Vector Addition Systems (VAS)

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

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

An extended model lifting the restriction that the valid vectors must have non-negative components has recently been defined in [9] and studied in [3]: An *integer*

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

An *integer vector addition system with states* ($\mathbb{Z}$-VASS) is a $\mathbb{Z}$-VAS equipped with a state control and is defined as a tuple $(\mathbf{w}_0, Q, q_0, q_h, p, \delta)$, where $\mathbf{w}_0 \in \mathbb{Z}^n$ is the start vector, Q is a finite set of state labels, $q_0 \in Q$ is the starting state, $q_h \in Q$ is the halting state, $p : Q \setminus \{q_h\} \to \mathbb{Z}^n$ is a function assigning a vector to every state from $Q \setminus \{q_h\}$, and $\delta : Q \to 2^Q$ is a state transition function assigning to each state the set of possible successor states. A $\mathbb{Z}$-VASS starts in $\mathbf{w}_0$ and in state q_0, applies the addition vector $p(q_0)$, and non-deterministically moves into one of the states from $\delta(q_0)$. This process is iteratively repeated, until the halting state q_h is reached. The vector language generated by a $\mathbb{Z}$-VASS is defined as the set of all vectors which are reachable in the halting state q_h.

It was shown in [11] that VASS are equivalent in expressive power to VAS (without states): any n-dimensional VASS can be simulated by an $(n+3)$-dimensional VAS. On the other hand, in [3, Section 6], it is proved that $\mathbb{Z}$-VASS are strictly more powerful than $\mathbb{Z}$-VAS. This is one first example showing that changing the nature of the objects on which a model of computing operates can affect its expressive power in important ways.

3 Register Machines over Groups

3.1 General Definition

In this section we extend the definition of register machines to allow their registers to contain elements of arbitrary finitely presented groups.

Definition 3. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine *(or a register machine over the family $\mathcal{G}$) is the construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$, where*

- *B is the set of labels bijectively labeling the instructions in the set P,*
- *$l_0 \in B$ is the initial label,*
- *$l_h \in B$ is the final label, and*
- *P is the set of instructions.*

The labeled instructions in P can be of the following forms:

- *$p : (ADD(r, b), T)$, with $p \in B \setminus \{l_h\}$, $T \subseteq B$, $1 \leq r \leq m$, $b \in B_r$.*
  Add the generator b of the group $G_r = \langle B_r \mid R_r \rangle$ to the current contents of the register r, then non-deterministically jump to one of the instructions in T.

- $l_h : HALT$.
  Stop the execution of the register machine.

A configuration of a $\mathcal{G}$-register machine is, like in the case of a classical register machine, the tuple $C = (q, r_1, \ldots, r_m)$, in which $r_i \in G_i$, $1 \leq i \leq m$, are the values of the registers, and q is the current instruction label which indicates the next instruction to execute. This label is called the state of the machine. We define the computations, halting, generating, and accepting for register machines over the family $\mathcal{G}$ in the same way as for conventional register machines in Section 2.2. In particular, $k \leq m$ registers are designated as input registers in the accepting case, (respectively, as output registers in the generating case), meaning that the $\mathcal{G}$-register machine accepts (respectively, generates) vectors of the form $(g_1, \ldots, g_k)$, where g_j, $1 \leq j \leq k$, belongs to a group $G_i \in \mathcal{G}$, $1 \leq i \leq m$, where different indices i are assigned to different indices j.

Remark 2. Note that the ADD instructions as we define them here allow a non-deterministic choice between more than two target states, as different from the classical definition, in which only two target states are allowed. We allow a set of possible target states because it simplifies the formulations of many properties and results, without critically affecting the power of the model: indeed, multiple target states can be easily simulated by a chain of dummy branching instructions.

Example 5. Consider the following family of 3 groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$, where $\mathbb{Z}$ is the usual Abelian group of integer numbers which can be presented in this way: $\mathbb{Z} = \langle 1 \mid a{+}b{+}(-a){+}(-b) \rangle$. A $\mathcal{Z}^3$-register machine $M_{\mathbb{Z}^3}$ is *almost* a blind 3-register machine: indeed, the three registers of $M_{\mathbb{Z}^3}$ may contain any integer number, an increment of a register r, $r \in \{1, 2, 3\}$, is done by the operation $ADD(r, 1)$, and a decrement by the operation $ADD(r, -1)$. Nevertheless, $M_{\mathbb{Z}^3}$ is *not* a blind register machine, because no zero check is performed at the end of a computation: the only restriction on the halting configuration is to have l_h as the current instruction label.

Given a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-RM to refer to the family of $\mathcal{G}$-register machines. We will sometimes also use the notation $*$-$RM = \bigcup_{\mathcal{G}} \mathcal{G}$-$RM$.

Remark 3. We observe that, in contrast to the original definition of register machines, the definition of $\mathcal{G}$-register machines only introduces increment instructions $p : (ADD(r, b), T)$ and no decrement instructions $p : (SUB(r, b), q, s)$, as decrementing by an element $e \in B_i$ corresponds to incrementing by $-e$. On the other hand, there is no direct check for zero in these ADD-instructions.

Remark 4. Register machines over groups as we define them here are somewhat similar to previous works on automata operating on groups (e.g. [17]). However, in our work, we rather focus on generalizing the ingredients forming register machines and describing them in a general setting, instead of analyzing their power as language recognizers.

Remark 5. We could extend the classical model of register machines to operate on other algebraic structures than groups. In this paper, we choose to focus on groups because these objects are rather well studied and there have already been previous works on using groups as a substrate for computation (e.g., [5]).

Vector Addition Systems over Groups

Even though register machines and vector addition systems are traditionally seen as quite different models and research on one often does not discuss the other (see, for example, the classic works [13] and [4]), the connection between the two is clearly rather strong, especially when considered in a more general setting. For example, $\mathbb{Z}$-VASS are equivalent in power to blind register machines [3]. In the present paper, we explicitly enforce this connection by defining vector addition systems over groups in terms of register machines over groups.

Definition 4. *Consider a finitely generated computable group $(G, \circ)$. A* vector addition system with states over G *(a G-VASS) is a tuple (g_0, M), where $g_0 \in G$ is the start element and M is a (G)-register machine (i.e., a machine with a single register over the group G) working in generating mode and whose only register is initialized with g_0.*

Definition 5. *Consider a finitely generated computable group $(G, \circ)$. A* vector addition system over G *(a G-VAS) is a G-VASS with the following structure on the instructions of the underlying register machine:*

- *$l_0 : (ADD(1, 0), B) \in P$: the initial instruction does not modify the contents of the register, but allows non-deterministically jumping to any other instruction, including the halting instruction.*
- *all instructions labelled by $l \in B \backslash \{l_0, l_h\}$ have the form $l : (ADD(0, g), B \backslash \{l_0\})$, with $g \in G$: the underlying machine can jump from any non-initial instruction to any other non-initial instruction, including the halting instruction.*

Example 6. Integer vector addition systems (with states), as introduced in [9] and studied in [3], are vector addition systems (with states) over the product group $\mathbb{Z}^n = \mathbb{Z} \times \cdots \times \mathbb{Z}$. Indeed, the elements of $\mathbb{Z}^n$ are n-vectors of integer numbers, and the state control of the $(\mathbb{Z}^n)$-register machine corresponds to the state control of the integer VASS. On the other hand, since the register machine associated with a VAS over $\mathbb{Z}^n$ can halt at any time, any vector it reaches belongs to the generated language.

Given a finitely generated computable group G, we will use the notations $G\text{-}VASS$ and $G\text{-}VAS$ to refer to the families of G-VASS and G-VAS, respectively. Since vector addition systems are only considered as generating devices, we will use the notations $\mathcal{L}(G\text{-}VASS)$ and $\mathcal{L}(G\text{-}VAS)$ to refer to the families of sets of elements of G generated by G-VASS and G-VAS, respectively.

3.2 Blindness, Partial Blindness, and the Zero Test

A $\mathcal{G}$-register machine as defined in the previous section is quite "blind": it has no mechanism to make the choice of the new instruction depend on the values of the registers. A classical way to introduce such a dependence is by allowing an explicit zero-test instruction.

Definition 6. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine with zero test *is the construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine, and the set P is also allowed to contain instructions of the following form:*

- $p : (0TEST(r), s, z)$, *with $p \in B \backslash \{l_h\}$, $s, z \in B$, $1 \leq r \leq m$.*
  Test if the current value of register r is equal to the neutral element of the group G_r; if yes, jump to instruction z, if not, jump to instruction s.

Configurations, computations, halting, generating, and accepting for $\mathcal{G}$-register machines with zero test are defined as for $\mathcal{G}$-register machines in Section 3.

Example 7. Consider the same family of 3 copies of the group of integers as in Example 5, $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. A $\mathcal{Z}^3$-register machine with zero test is *almost* like a conventional register machine: increment and decrement are done by the instructions $ADD(r, 1)$ and $ADD(r, -1)$, and zero test by the $TEST(r)$ instructions. However, the registers of a $\mathcal{Z}^3$-register machine with zero test are allowed to contain arbitrary integers.

For a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-RM_0 to refer to the family of $\mathcal{G}$-register machines with zero test.

Allowing an explicit zero test instruction is known to strictly increase the computational power of register machines (e.g., [8]). A much weaker way of introducing a dependency between the contents of the registers and the choice of instructions is the terminal zero test.

Definition 7. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. A* $\mathcal{G}$-register machine with a terminal zero test *(a* blind $\mathcal{G}$-register machine*) is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine and, in any halting configuration, all registers which have not been explicitly designated as output must contain the neutral element of the corresponding group.*

Computations, halting, generating, and accepting for $\mathcal{G}$-register machines with a terminal zero test are defined as for $\mathcal{G}$-register machines in Section 3, with the additional requirement of emptiness of the working registers, as indicated in the previous definition.

We use the notation $\mathcal{G}$-BRM to refer to the family of $\mathcal{G}$-register machines with a terminal zero test.

Remark 6. Using the notation *BRM* refers to the original definition of blind register machines which we take over for the general case of $\mathcal{G}$-register machines.

Example 8. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. A $\mathcal{Z}^3$-register machine with a terminal zero test is a blind register machine, as usually defined in the literature (e.g., [3, 8]).

Remark 7. Sheila Greibach's original paper introducing blind and partially blind register machines [8] considers them as recognizers of *strings*: these devices read the string from the beginning to the end, using registers to store internal information. Later works (e.g., [13]) tend to discard the string recognizer aspect, and instead treat register machines as devices manipulating numbers exclusively. In general, it is quite easy to encode any string as a number (using, for example, a prime number encoding over the alphabet), therefore restricting registers machines to numbers does not critically affect their expressiveness. In Section 3.4, we show how to recover string-related behavior in register machines over groups with forbidden regions.

3.3 Partial Blindness

The types of register machines over groups we have defined up to now do not directly generalize the classical register machines, which can be seen as defined over the monoid of natural numbers $(\mathbb{N}, +)$: addition over the natural numbers is not invertible, because negative numbers do not belong to $\mathbb{N}$. To capture this restriction, we directly draw inspiration from the definitions of VAS and conventional partially blind register machines, and define partially blind register machines over totally ordered groups.

Definition 8. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Suppose one of the groups G_j, $1 \leq j \leq m$, is totally ordered with the total order $\leq_j$. A $\mathcal{G}$-*register machine with a partially blind register j *is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine whose register j is only allowed to contain values $a_j \in G_j$ with the property $0_j \leq_j a_j$.*

Configurations, computations, halting, generating, and accepting for $\mathcal{G}$-register machines with some partially blind registers are defined as for $\mathcal{G}$-register machines in Section 3, with the additional restriction on the values of the partially blind registers.

Example 9. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$. The group $(\mathbb{Z}, +)$ is totally ordered with respect to the natural order. A $\mathcal{Z}^3$-register machine partially blind in all of its 3 registers is a 3-register partially blind register machine in the conventional sense, but without the final zero test.

For a finite family of finitely generated computable groups $\mathcal{G}$, we use the notation $\mathcal{G}$-PB_ARM, with $A \subseteq \{1, \ldots, m\}$, to refer to the family of $\mathcal{G}$-register machines with partially blind registers with indices from A. This supposes that the groups of $\mathcal{G}$ with indices from A are totally ordered.

When $A = \{1, \ldots, m\}$, i.e., all the registers are partially blind, we will omit the subscript A from the notations, and we will refer to the register machine itself as being *partially blind*. We use the particular notation $\mathcal{G}$-$PBRM$ to refer to the family of register machines with all registers blind, and with the final zero test at the end of successful computations.

Note finally that $\mathcal{G}$-register machines with all registers blind and with the zero test instruction directly generalize classic register machines.

3.4 Forbidden Regions

While $\mathcal{G}$-register machines with partially blind registers are a generalization which is rather close to conventional register machines, imposing a total order on a group is a rather strong condition: for example, it entails the absence of elements of a finite order [14], thus excluding cyclic groups from consideration. Notice, however, that the total order is essentially used to define a forbidden subset of elements. We can therefore define another generalization of conventional register machines which imposes less constraints on the group.

Definition 9. *Let $m \in \mathbb{N}$ and take the finite family of finitely generated computable groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Let $\mathcal{F} = (F_i)_{1 \leq i \leq m}$ be a family of subsets of the groups in $\mathcal{G}$: $F_i \subseteq G_i$, $1 \leq i \leq m$. A $\mathcal{G}$-*register machine with forbidden regions *$\mathcal{F}$ is a construct $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ such that $M_{\mathcal{G}}$ is a $\mathcal{G}$-register machine whose register i is only allowed to contain the elements in $G_i \setminus F_i$, $1 \leq i \leq m$.*

Configurations, computations, halting, generation, and acceptance for $\mathcal{G}$-register machines with forbidden regions are defined as for $\mathcal{G}$-register machines in Section 3, with the additional restriction on the values of registers: if a forbidden value appears in a register, the computation crashes without producing any output.

Example 10. Consider the family of groups $\mathcal{Z}^3 = (\mathbb{Z}, \mathbb{Z}, \mathbb{Z})$ and the family $\bar{\mathcal{N}}^3 = (\bar{N}, \bar{N}, \bar{N})$, where $\bar{N} = \mathbb{Z} \setminus \mathbb{N}$. A $\mathcal{Z}^3$-register machine with the forbidden regions $\mathcal{N}^3$ is also a $\mathcal{Z}^3$-register machine partially blind in all of its registers.

Since groups suitably abstract a large number of objects and since forbidden regions can be used to carve particular "shapes" out of a given group, multiple connections with different domains can be traced for register machines over groups and with forbidden regions.

Example 11. The dihedral group D_n is the group of symmetries of a regular polygon with n sides and can be presented as follows: $D_n = \langle r, s \mid r^n, s^2, (sr)^2 \rangle$ [20].

The infinite dihedral group D_∞ can be seen as the group of symmetries of integers and can be presented as $D_\infty = \langle r, s \mid s^2, (sr)^2 \rangle$. The Cayley graph of this presentation can be depicted as follows [5]:

$$\begin{array}{ccccccccccccc}
\ldots & sr^2 & \xleftarrow{r} & sr & \xleftarrow{r} & s & \xleftarrow{r} & sr^{-1} & \xleftarrow{r} & sr^{-2} & \ldots \\
 & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & & s \downarrow\uparrow s & \\
\ldots & r^{-2} & \xrightarrow{r} & r^{-1} & \xrightarrow{r} & e & \xrightarrow{r} & r & \xrightarrow{r} & r^2 & \ldots
\end{array}$$

In this picture, the lower and the upper lines are going into opposite directions, which nicely fits as a representation of double-stranded DNA molecules, i.e., the lower line going from the left 5′-end to the right 3′-end, whereas the complementary upper line goes from the right 5′-end to the left 3′-end [5]. Thus, if the family $\mathcal{G}$ contains D_∞, a $\mathcal{G}$-register machine can be seen as operating on a DNA molecule. Forbidding the region $F = \{r^k \mid k \in \mathbb{Z}\} \subset D_\infty$ can be seen as restricting the register machine to operate on one of the strands of the molecule (the upper one on the figure).

Example 12. Take a finite alphabet of symbols V and consider the free group $\langle V \mid \emptyset \rangle = (I(V), \circ)$ over V. It follows from the definition of the free group that it contains two types of elements:

- strings from the syntactic monoid V^*: $a_1 \circ \cdots \circ a_n$, such that $a_1 \ldots a_n \in V^*$;
- strings which include the inverses $\{a^{-1} \mid a \in V\}$ of the elements of V.

Take now the singleton family of groups $\mathcal{G} = (\langle V \mid \emptyset \rangle)$ and the singleton family of forbidden regions $\mathcal{F} = (F)$, with F containing all the elements of G of the second type. Then the only register of a $\mathcal{G}$-register machine M with the forbidden regions $\mathcal{F}$ will contain strings from V^* in any successful computation.

Remark 8. On a historical side-note, register machines as originally introduced by Minsky in [15] came out as a consequence of reducing the tape alphabet of Turing machines to two symbols, including the empty symbol. Such a reduction imposes unary encoding of the working values and essentially transforms the tape into a series of registers [12]. By generalizing register machines from natural numbers to groups, we come back to computing devices operating on strings.

For a finite family of finitely generated computable groups $\mathcal{G}$, we will use the notation $\mathcal{G}$-$RM_{\neg\mathcal{F}}$ to refer to the family of $\mathcal{G}$-register machines with the forbidden regions $\mathcal{F}$.

Vector Addition Systems with Forbidden Regions

Since we define vector addition systems over groups as particular cases of register machines, the idea of forbidden regions can be easily transported to VAS.

Definition 10. *Consider a finitely generated computable group $(G, \circ)$ and a subset $F \subseteq G$. A* vector addition system over G (respectively, with states) with the forbidden region F *is a vector addition system (respectively, with states) whose underlying (G)-register machine belongs to (G)-$RM_{(F)}$.*

Example 13. A $\mathbb{Z}^n$-VAS with the forbidden region $F = \{(x_1, \ldots, x_n) \mid \exists i : x_i < 0\}$ is an n-component vector addition system as classically defined.

Given a finitely generated computable group G, we will use the notation G-$VASS_{\neg F}$ (respectively, G-$VAS_{\neg F}$) to refer to the family of G-VASS (respectively, G-VAS) with the forbidden region F.

4 Expressive Power of RM and VAS over Groups

In this section we will a give a series of results characterizing the power of register machines over groups with or without ingredients. We start by considering the simplest case: no ingredients and singleton group families.

4.1 Singleton Group Families

For register machines with no ingredients, there is little difference between considering non-singleton and singleton group families. In this section, we will use the notation $\mathcal{C} = (C_k)_{0 \leq k \leq n}$ to refer to an n-step computation of a $\mathcal{G}$-register machine, where C_k is a vector of elements of the groups in $\mathcal{G}$ collecting the contents of the registers at step k. Notice that this definition of computation and configurations discards the instruction label, as opposed to the more general definition given in Section 3.

Proposition 1. *Consider a $\mathcal{G}$-register machine M over a non-singleton family $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, $m > 1$. Then there exists a singleton family $\mathcal{G}^0 = (G)$, a family of projections $\pi = (p_i : G \to G_i)_{1 \leq i \leq m}$, and a $\mathcal{G}^0$-register machine M^0 such that, for any n-step computation $\mathcal{C}$ of M there exists an n-step computation $\mathcal{C}^0$ of M^0 with the following property:*

$$C_k[j] = p_j(C_k^0),\ 1 \leq j \leq m,$$

where $C_k^0 \in \mathcal{C}^0$, $C_k \in \mathcal{C}$, and $C_k[j]$ is the j-th element of the vector C_k.

Proof. It suffices to take the group G to be the direct product [10] of the groups in $\mathcal{G}$: $G = \prod_{i=1}^{m} G_i$. The $\mathcal{G}^0$-register machine M^0 will thus have a single register containing vectors of values of the groups in $\mathcal{G}$. Any $ADD(j, b)$ instruction of M will be represented in M^0 by an instruction $ADD(1, \mathbf{b})$, where $\mathbf{b} = (e_1, \ldots, e_{j-1}, b, e_{j+1}, \ldots, e_m)$ is a vector consisting of the neutral elements of the groups in $\mathcal{G}$, except for the j-th element. □

The converse statement is not true, because any vectors from the direct product of G can appear in the ADD instructions of M^0, thus affecting multiple components of the vector from G at once. However, unsurprisingly, any computing step of M^0 can still be simulated by M in *multiple* steps.

Proposition 2. *Consider the non-singleton family of groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$, $m > 1$, and take the singleton family $\mathcal{G}^0 = (G)$, where G is the direct product of the groups in $\mathcal{G}$ and $\pi = (p_i : G \to G_i)_{1\leq i\leq m}$ is the corresponding family of projections. Then, for any $\mathcal{G}^0$-register machine M^0 there exists a $\mathcal{G}$-register machine M such that, for any n-step computation $\mathcal{C}^0$ of M^0, there exists an n'-step computation $\mathcal{C}$ of M, $n' > n$, with the property:*

$$C_0[j] = p_j(C_0^0) \text{ and } C_{n'}[j] = p_j(C_n^0),\ 1 \leq j \leq m,$$

where C_0 and $C_{n'}$ are the first and the last configurations of the computation $\mathcal{C}$, and C_0^0 and C_n^0 are the first and the last configurations of the computation $\mathcal{C}^0$.

Proof (sketch). M simulates the instruction $p : (ADD(0, \mathbf{b}), T)$ of M^0 by the following sequence of instructions:

$$\begin{aligned} p_0 &: \big(ADD\big(0, p_0(\mathbf{b})\big), \{p_1\}\big), \\ p_j &: \big(ADD\big(j, p_j(\mathbf{b})\big), \{p_{j+1}\}\big), \quad 1 < j < m, \\ p_m &: \big(ADD\big(m, p_m(\mathbf{b})\big), T\big). \end{aligned}$$

This ensures that M simulates M^0 with a constant-time slowdown and proves the statement of the proposition. □

The two previous propositions imply that, in a somewhat counter-intuitive way, blind single-register machines are a little more efficient than multi-register machines, because the former may require less computational steps to achieve a given configuration than the latter. This statement, however, becomes false with the addition of some of the ingredients we considered in the previous sections. Indeed, the zero test in a single-register machine over a direct product of groups requires that all components of the combined register should be zero; it is impossible to individually test the components. Similarly, transposing the total orders on some or all of the groups of the family $\mathcal{G}$ to their direct product is not generally possible. Forbidden regions are, on the other hand, more flexible and can be directly carried over from individual groups to components of the elements of the product.

The conclusion we make from these arguments is that, when no additional ingredients are considered, the power of register machines over groups does not depend on the number of registers.

Theorem 1. *Consider the family of finitely presented computable groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$ and a $\mathcal{G}$-register machine M. Then there exists a $\mathcal{G}^0$-register machine M^0 over the singleton family $\mathcal{G}^0 = (\prod_{i=1}^m G_i)$ such that $\mathcal{L}_X(M) = \mathcal{L}_X(M^0)$, with $X \in \{acc, gen\}$.*

4.2 Generation and Acceptance: No Ingredients

As a consequence of the definition of VASS over groups, Theorem 1 implies that any $\mathcal{G}$-register machine working in the generating mode can be simulated by a VASS over the direct product of the groups in $\mathcal{G}$.

Corollary 1. *Consider the family of finitely presented computable groups $\mathcal{G} = (G_i)_{1\leq i\leq m}$ and a $\mathcal{G}$-register machine M. Then there exists a G-VASS A over the product $G = \prod_{i=1}^{m} G_i$ such that $\mathcal{L}_{gen}(M) = \mathcal{L}(A)$.*

The similar statement for register machines in accepting mode does not hold. In fact, an accepting register machine either accepts or rejects any contents of the input registers.

Proposition 3. *Consider the family of finitely generated computable groups $\mathcal{G}$, a subfamily $\mathcal{G}_{in}$ and a $\mathcal{G}$-register machine M whose input registers correspond exactly to the groups from $\mathcal{G}_{in}$. Then $\mathcal{L}_{acc}(M) \in \left\{\emptyset, \prod_{G\in\mathcal{G}_{in}} G\right\}$.*

Proof. M can accept the empty language by never reaching the halting state. Suppose now that it accepts some input vector $\mathbf{x} \in \prod_{G\in\mathcal{G}_{in}} G$. Since the state transitions of M do not depend on the values of its registers, and since no particular conditions are checked at halting, the sequence of actions applied to accept $\mathbf{x}$ can be applied to accept any other $\mathbf{x}' \in \prod_{G\in\mathcal{G}_{in}} G$, meaning that M will accept all possible vectors in $\prod_{G\in\mathcal{G}_{in}} G$. □

We therefore conclude that generation is at least as powerful as acceptance for $\mathbb{G}$-register machines without any additional ingredients.

Theorem 2. *Consider the family of finitely generated computable groups $\mathcal{G}$. Then $\mathcal{L}_{acc}(\mathcal{G}\text{-}RM) \subseteq \mathcal{L}_{gen}(\mathcal{G}\text{-}RM)$.*

Proof. According to Proposition 3, it suffices to show how to generate the empty language and the language of all vectors over the groups corresponding to the output registers. The empty language can be generated by never reaching the halting state. The language of all vectors can be generated by non-deterministically adding the corresponding generators and their inverses to the output registers. □

The inclusion from the previous theorem is not strict. The following example shows a case in which the generating and accepting power are equal.

Example 14. Consider the singleton group $\mathbf{1}$ containing the single element e and a group family $\mathcal{G}$ containing $\mathbf{1}$. Then the languages accepted by $\mathcal{G}$-register machines with the input register containing elements from $\mathbf{1}$ is equal to the languages generated by $\mathcal{G}$-register machines with the output register containing elements from $\mathbf{1}$. Indeed, the only two possible languages which can be accepted or generated are $\emptyset$ and $\{e\}$. As discussed previously, the first language is accepted/generated by never reaching the halting state, and the second language is accepted/generating by halting immediately.

On the other hand, generating $\mathcal{G}$-register machines are not restricted to generating all possible combinations of values of their output registers, as the following example shows.

Example 15. Consider the generating ($\mathbb{Z}$)-register machine M with two states and whose only non-halting state is associated with the instruction $ADD(1,1)$ adding 1 to the contents of its only register. When the register of M is initialized to 0, M only generates the set of natural numbers $\mathbb{N}$.

The difference in power between the accepting mode and the generating mode puts forward an asymmetry in the definition of the two semantics: in the generating mode, the registers have the "knowledge" about their initial values, whereas in the accepting mode, no information about the register contents whatsoever is available.

4.3 Generation and Acceptance: The Zero Test

In this subsection we exhibit that allowing the zero test instruction equalizes the power of the accepting and generating modes. In line with the usual terminology, we will use the term "increment register i by b" to refer to composing the contents of register i with the generator b, and the term "decrement i by b" to refer to composing the contents of register i with the *inverse* of the generator b.

Lemma 1. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{gen}(\mathcal{G}\text{-}RM_0) \subseteq \mathcal{L}_{acc}(\mathcal{G}'\text{-}RM_0)$.*

Proof. Let $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ be a $\mathcal{G}$-register machine with zero test. We consider $M_{\mathcal{G}}$ as a generating device, where $1 \leq j \leq k$ are the output registers. We now construct a register machine with zero test $M_{\mathcal{G}'} = (\mathcal{G}', B', l_0', l_h', P')$ with

$$\mathcal{G}' = (G_1, \ldots G_k, G_1, \ldots G_k, G_{k+1}, \ldots G_{m+k}),$$

i.e., every output register of $M_{\mathcal{G}}$ appears in two copies, and the first copy is designated as an input register of $M_{\mathcal{G}'}$, which now becomes an accepting device with the input registers $1 \leq k \leq m$. Given any input in the input registers, $\mathcal{G}'$ simulates $M_{\mathcal{G}}$ in the registers $k+1, \ldots, m+k$ representing the registers $1, \ldots, m$ using the instructions in P' with each register j in an instruction of P replaced by the corresponding register $k+j$ in the instructions of P'. With $M_{\mathcal{G}}$ reaching l_h, also $M_{\mathcal{G}'}$ reaches l_h. After that, in a final procedure, $\mathcal{G}'$ checks if the contents of register j equals the contents of register $j+k$ for every $1 \leq j \leq k$. In the success case, $\mathcal{G}'$ enters the final label l_h'.

For $j = 1, \ldots, k$, starting with p_1, sequences of instructions
$p_j : (0TEST(j), \hat{p}_j, p_j')$,
$p_j' : (0TEST(k+j), p_j', p_{j+1})$,
$\hat{p}_j : (ADD(j, -b), \{\bar{p}_j\})$,

$\bar{p}_j : (0TEST(j+r), \tilde{p}_j, \bar{p}_j)$, and
$\tilde{p}_j : (ADD(j+k, -b), \{p_j\})$
simultaneously decrement related registers j and $j+k$, $1 \leq j \leq k$, down to zero. In this construction, we define an instance of the rule $\hat{p_j}$ and an instance of the rule $\tilde{p_j}$ for every generator b of the group G_j, which allows testing registers j and $j+k$ for equality independently of the number of generators of the corresponding (finitely generated) group. In the success case, i.e., if both have been checked to be equal, the procedure continues with the next pair of registers. At the end, in the success case, we take $p_{k+1} = l'_h$. In the failure case, an infinite loop is entered. We leave the remaining details of the construction to the interested reader. For example, to allow non-deterministic branching from a label p to q and s, without modifying the registers, we can use a working register r and a generator b as well as the sequence of instructions $p : (ADD(r, b), \{p'\})$ and $p' : (ADD(r, -b), \{q, s\})$. Moreover, if there is no working register in $M_{\mathcal{G}}$, we add one in $M_{\mathcal{G}'}$.

We conclude that the set generated by $M_{\mathcal{G}}$ equals the set accepted by $M_{\mathcal{G}'}$. □

Lemma 2. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{acc}(\mathcal{G}\text{-}RM_0) \subseteq \mathcal{L}_{gen}(\mathcal{G}'\text{-}RM_0)$.*

Proof. We now start with an accepting $\mathcal{G}$-register machine with zero test $M_{\mathcal{G}} = (\mathcal{G}, B, l_0, l_h, P)$ and construct a generating register machine with zero test $M_{\mathcal{G}'} = (\mathcal{G}', B', l'_0, l'_h, P')$, again using a similar construction of additional registers as in the proof of Lemma 1. $M_{\mathcal{G}'}$ randomly generates two copies of the output in registers i and $i+k$, $1 \leq i \leq k$. Then $M_{\mathcal{G}'}$ simulates an accepting computation of $M_{\mathcal{G}}$ in the registers $k+1, \ldots, m+k$ of $M_{\mathcal{G}'}$. In case l_h is reached in that way, instead of halting $M_{\mathcal{G}'}$ finally decreases all working registers $k+1, \ldots, m+k$ to zero:

For $j = k+1, \ldots, m+k$, starting with $p_{k+1} = l_h$, sequences of instructions
$p_j : (0TEST(j), \hat{p_j}, p_{j+1})$ and
$\hat{p}_j : (ADD(j, -b), \{p_j, p_j\})$
are carried out in a deterministic way, finishing with the HALT-instruction with label $l'_h = p_{m+k}$. We conclude that the set accepted by $M_{\mathcal{G}}$ equals the set generated by $M_{\mathcal{G}'}$. □

As an immediate consequence of the two preceding lemmas, we conclude that the generating and the accepting power of register machines over groups with the zero test instruction is equal.

Theorem 3. $\mathcal{L}_{acc}(*\text{-}RM_0) = \mathcal{L}_{gen}(*\text{-}RM_0)$.

A result similar to the one stated in Lemma 2 also holds true for blind register machines:

Corollary 2. *Let $m \in \mathbb{N}$ and take the finite family of finitely presented groups $\mathcal{G} = (G_i)_{1 \leq i \leq m}$, with $G_i = \langle B_i \mid R_i \rangle$. Then there exists another family of finitely presented groups G' such that $\mathcal{L}_{acc}(\mathcal{G}\text{-}BRM) \subseteq \mathcal{L}_{gen}(\mathcal{G}'\text{-}BRM)$.*

Proof. We can use the same construction as described in the proof of Lemma 2, except for the final procedure decrementing all working registers to zero, which is not needed as by definition acceptance for blind register machines requires all working registers to be zero. □

4.4 Vector Addition Systems over Groups

One of the classical results on vector addition systems is that, in the conventional definition of the model, adding a state control does not increase the power, because the states can be simulated using 3 additional components of vectors [11, Lemma 2.1]. This result can be naturally generalized to vector addition systems over groups with forbidden regions.

Theorem 4. *Consider a finitely generated computable group G, the product $G' = G \times \mathbb{Z}^3$, its subset $F = \{(g, a, b, c) \in G' \mid a < 0 \text{ or } b < 0 \text{ or } c < 0\}$, and an arbitrary G-VASS A. Then there exists a G'-VAS with the forbidden region F whose computations modulo the natural projection $p : G' \to G$ are exactly the computations of A.*

Proof. The construction from the proof of [11, Lemma 2.1] can be directly carried over to our setting: the three components over $\mathbb{Z}$ with forbidden negative values can be used to encode the current state and to compute the next one unambiguously. We do not recall the cited construction here because it is rather technical and out of the scope of this article. □

We can immediately generalize this result by replacing $\mathbb{Z}$ in the previous statement by a different group into which one can injectively (monomorphically) embed $\mathbb{Z}$.

Theorem 5. *Consider a finitely generated computable group G, and a totally ordered group Z such that there exists an injective homomorphism of totally ordered groups $i : \mathbb{Z} \to Z$. Take the product $G' = G \times Z^3$, its subset $F = \{(g, a, b, c) \in G' \mid a <_Z i(0) \text{ or } b <_Z i(0) \text{ or } c <_Z i(0)\}$, and an arbitrary G-VASS A. Then there exists a G'-VAS with the forbidden region F whose computations modulo the natural projection $p : G' \to G$ are exactly the computations of A.*

Proof. The injective homomorphism i delimits a totally ordered subgroup of Z which is isomorphic to $\mathbb{Z}$, allowing to perform the same operations as in the proof of [11, Lemma 2.1]. □

Remark 9. The result [11, Lemma 2.1] as well as the two generalizations we give here do *not* necessarily state the equality between the families of languages generated by VAS with and without states. Indeed, the language generated by a VASS is usually taken to contain all the vectors which the VASS reaches while also being in a terminal or halting state, while the language of a VAS is often taken to be simply its reachability set. In Definition 5, this behavior is captured by allowing the underlying register machine of a G-VAS to halt at any time.

A consequence of the fact that only the elements a G-VASS produces in its halting state contribute to the generated language is that a G-VASS can generate the empty language $\emptyset$ by never reaching the halting state. On the other hand, the language of a G-VAS always includes at least the start element. This observation together with the fact that we define G-VAS as a particular case of G-VASS implies the following statement.

Proposition 4. *For any finitely generated computable group G, it holds that $\mathcal{L}(G\text{-}VAS) \subsetneq \mathcal{L}(G\text{-}VASS)$.*

Since this strict inclusion is rather trivial and does not reflect the intrinsic computing power of vector addition systems, in the rest of this section we will only consider G-VASS generating non-empty languages. In this setting, the increase in power due to the state control depends strongly on the underlying group. For example, $\mathbb{Z}^n\text{-}VAS \subsetneq \mathbb{Z}^n\text{-}VASS$, as shown in [3, Lemmas 6 and 7]. On the other hand, it follows trivially from Proposition 3 and Example 14 that adding states to vector addition systems over the singleton group $\mathbf{1}$ does not increase the power. We generalize these observations in the following statement.

Theorem 6. *If a finitely generated computable group G contains an element of order greater than 2, then $\mathcal{L}(G\text{-}VAS) \subsetneq \mathcal{L}(G\text{-}VASS) \setminus \{\emptyset\}$.*

Proof. Suppose that g is the element of G whose order is greater than 2. Suppose that there exists such a G-VAS A with the start element $g_0 \in G$ that $\mathcal{L}(A) = \{e, g\}$, where e is the neutral element of G. Then there exist two elements $h_0, h_1 \in G$ such that $g_0 h_0 = e$ and $g_0 h_1 = g$, and A executes the operations corresponding to adding h_0 to g_0 to generate e, and corresponding to adding h_1 to g_0 to generate g. Since $ord(g) > 2$, $g \neq e$, and either $h_0 \neq e$, or $h_1 \neq e$, or both. Let $h \in \{h_0, h_1\}$ such that $h \neq e$. Then, if A executes the sequence of actions corresponding to h_0, and afterwards the one corresponding to h, it will generate $g_0 h_0 h = h$. If $h \notin \{e, g\}$, then $\mathcal{L}(A) \supsetneq \{e, g\}$, which is a contradiction.

Now suppose that $h \in \{e, g\}$. By construction, $h \neq e$, so $h = g$. Suppose that A carries out the sequence of actions corresponding to h_0, then the sequence corresponding to h, and then the same sequence again. It would generate $g_0 h_0 h h = h^2 = g^2$. By hypothesis, $ord(g) > 2$, meaning that $g^2 \notin \{e, g\}$. But in this case $\mathcal{L}(A) \supsetneq \{e, g\}$, which is again a contradiction.

We conclude the proof by remarking that the language $\{e, g\}$ can be generated by a G-VASS with the starting element e and whose underlying register machine contains the single instruction $l : (ADD(1, g), \{l_h\})$. □

It follows immediately from the previous theorem that the state control already makes a difference for vector addition systems over $\mathbb{Z}_3 = \mathbb{Z}/3\mathbb{Z}$, the group of addition modulo 3. Indeed, it is impossible to construct a $\mathbb{Z}_3$-VAS generating $\{0, 1\}$: any attempt would end up putting the element 2 into the generated language.

Corollary 3. $\mathcal{L}(\mathbb{Z}_3\text{-}VAS) \subsetneq \mathcal{L}(\mathbb{Z}_3\text{-}VASS) \setminus \{\emptyset\}$.

On the other hand, the state control does not increase the power of vector addition systems over the two-element group $\mathbb{Z}_2 = \mathbb{Z}/2\mathbb{Z}$.

Proposition 5. $\mathcal{L}(\mathbb{Z}_2\text{-}VAS) = \mathcal{L}(\mathbb{Z}_2\text{-}VASS) \setminus \{\emptyset\}$.

Proof. Only the following non-empty languages over $\mathbb{Z}_2$ exist: $\{0\}$, $\{1\}$, and $\{0,1\}$. The first two ones can be generated by a $\mathbb{Z}_2$-VAS whose underlying register is initialized to 0 or 1, respectively, and whose underlying register machine always halts immediately.

The third one is generated by a $\mathbb{Z}_2$-VAS with the start element $g_0 \in \{0,1\}$ and with the underlying register machine containing only one instruction $l : (ADD(1,1), \{l, l_h\})$. □

Even though Theorem 6 gives a sufficient criterion for the state control to strictly augment the expressive power of G-VAS, we do not claim that this criterion is necessary. Establishing a necessary and sufficient criterion is left as an open problem.

We conclude this discussion about the frontier between the power of G-VAS and G-VASS by recalling that the paper [3] considers *uniform families* of VAS, $\mathbb{Z}\text{-}VAS_{\cup}$, which are essentially an extension of vector addition systems allowing a finite number of start vectors instead of only one of them. In a similar fashion, we can consider uniform families of G-VAS. We denote these by $G\text{-}VAS_{\cup}$. For a finite group H, languages generated by uniform families of H-VAS turn out to be the same as those generated by H-VASS.

Proposition 6. *For a finite group H, $\mathcal{L}(H\text{-}VAS_{\cup}) = \mathcal{L}(H\text{-}VASS)$.*

Proof. Since H is finite, H-VASS generate finite languages. Hence, any given H-VASS A can be "simulated" by a uniform family of H-VAS without any addition elements (the underlying register machine halts immediately) and whose start elements form exactly $\mathcal{L}(A)$. □

5 Generating and Accepting Strings

In this section, we will show how the idea to use the free group constrained to the syntactic monoid introduced in Example 12 can be used to turn register machines over groups into devices recognizing and generating strings. We assume that the reader is familiar with regular and context free grammars, finite-state and push-down automata, as well as Turing machines. For an extensive introduction to the domain of formal languages, we refer to [19].

For an alphabet V, we will use the symbol $\mathcal{V}$ to refer to the free group $\langle V \mid \emptyset \rangle$ whenever no ambiguity occurs. We will also use the notation $F_{\mathcal{V}}$ to refer to the elements of the free group $\mathcal{V}$ which do not appear in the syntactic monoid V^*: $F_{\mathcal{V}} = \{x \in \mathcal{V} \mid x \notin V^*\} = \mathcal{V} \setminus V^*$.

Our first result shows that a register machine with only one register containing values from $\mathcal{V}$ can simulate a regular grammar. The symbol REG_V stands for the class of all regular languages over the alphabet V.

Theorem 7. $\mathcal{L}_{gen}\big((\mathcal{V})\text{-}RM_{\neg(F_\mathcal{V})}\big) = REG_V$.

Proof. Consider an arbitrary regular language L and the regular grammar $G = (N, V, P, S)$ generating it, where N is the set of non-terminal symbols, V is the set of terminal symbols, $N \cap V = \emptyset$, P is the set of productions, and S is the starting symbol. We will construct a $(\mathcal{V})$-register machine with the forbidden region $F_\mathcal{V}$ which will generate the language L. We associate the instruction labels $l(A)$ (defined below) with every non-terminal $A \in N$ and we construct the program of M in the following way:

- for every rule $A \to aB$, $A, B \in N$, $a \in V$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, a), l(B))$ to the program of M;
- for every rule $A \to a$, $A \in N$, $a \in V$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, a), \{l_h\})$ to the program of M;
- for every rule $A \to \lambda$, $A \in N$, we add a fresh label l_A to the set $l(A)$ and the instruction $l_A : (ADD(1, \lambda), \{l_h\})$ to the program of M, where λ is the empty string and the neutral element of the group $\mathcal{V}$.

The set of instruction labels of M is therefore $B = \bigcup_{A \in N} l(A)$. Without losing generality, we may assume that $l(S)$ only contains one element, which will serve as the starting label for M.

By construction, M faithfully simulates the regular grammar G by reflecting the current non-terminal symbol in the instruction label, by adding the corresponding symbol to the only register containing the generated string, and by non-deterministically jumping to one of the instruction labels corresponding to the new non-terminal symbol if a rule $A \to aB$ is applied. □

A symmetric result can be proved for the accepting mode, except that in this case we need the terminal zero test to ensure that the input string has been read completely. In this statement, we combine the notation BRM and the subscript $\neg(F_\mathcal{V})$ to refer to register machines with both the terminal zero test and forbidden regions.

Theorem 8. $\mathcal{L}_{acc}\big((\mathcal{V})\text{-}BRM_{\neg(F_\mathcal{V})}\big) = REG_V$.

Proof. Consider a regular language L and a (non-deterministic) finite automaton $FA = (Q, V, \delta, q, F)$ recognizing it, where Q is the set of states, V is the set of input symbols, $\delta : Q \times V \to 2^Q$ is the transition function giving a set of target states based on the current state and the symbol on the tape, q is the starting state, and $F \subseteq Q$ is the set accepting states. We construct a $(\mathcal{V})$-register machine with the forbidden region $F_\mathcal{V}$ which accepts the reverse image of the language L, i.e., $\mathcal{L}_{acc}(M) = \{s^R \mid s \in L\} = L^R$, where s^R is the reverse of s.

We denote $l_h(p) = \{l^h\}$ if $p \in F$ and $l_h(p) = \emptyset$ otherwise. We define the following mapping from the set of states of FA to the set of labels of M:

$$l(p) = \{p_a \mid p \in Q, a \in V, \delta(p,a) \neq \emptyset\} \cup l_h(p).$$

We also use the natural extension $l(Q') = \bigcup_{p \in Q} l(p)$, for $Q' \subseteq Q$.

For every pair $p \in Q$ and $a \in V$ for which $\delta(p,a) \neq \emptyset$, we add the following to M:

1. the label p_a to the set of labels B;
2. the instruction $p_a : \big(ADD(1, a^{-1}),\ l(\delta(p,a)) \big)$ to the program.

Finally, we add the instruction $l_0 : (ADD(1,\lambda), l(q))$ to M, where q is the starting state of FA.

To recognize the string $s^R \in L^R$, M first non-deterministically jumps to one of the labels q_a by performing the instruction l_0 which does not modify the register. In the following step, the machine performs $ADD(1, a^{-1})$. If the string in its register has the form wb, $w \in V^*$, $b \in V \setminus \{a\}$, then this operation results in the forbidden string wba^{-1} and M aborts. Otherwise the value of the register becomes w, M non-deterministically jumps to one of the labels in $l(\delta(q,a))$, and repeats the same procedure.

Whenever the machine simulates a jump to a state $p \in F$, it may choose to jump to the instruction l^h and halt. If it does so while the register does not contain the empty string λ, the terminal zero test will fail and the computation will abort. If, on the other hand, it does not jump to l^h after updating its register to λ, the subsequent instruction $ADD(1, a^{-1})$ aborts the computation.

We conclude the proof by recalling that regular languages are closed under the reverse image. □

Registers containing elements from $\mathcal{V}$ can also be used as stacks, allowing to simulate pushdown automata. We only give sketches of the proofs of the following results, the omitted details being very similar to those appearing in the previous proof.

Theorem 9. *Consider two alphabets V and R, the corresponding free groups $\mathcal{V}$ and $\mathcal{R}$, and the regions $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$ and $F_{\mathcal{R}} = \mathcal{R} \setminus R^*$. Then $(\mathcal{V}, \mathcal{R})$-register machines with terminal zero test and with the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{R}})$ accept all context-free languages that can be accepted by a pushdown automaton with the tape alphabet V and the stack alphabet R.*

Proof (sketch). The proof idea is very close to that employed in Theorem 8: we construct a register machine M with two registers. The first register contains the input string which it non-deterministically "reads" by appending symbols a^{-1} and aborting when the symbol was not guessed correctly. The second register contains the stack. M pushes a symbol $z \in R$ on the stack by performing $ADD(2, z)$ and

pops a symbol by non-deterministically performing $ADD(2, z^{-1})$. At the end of the computation, both registers must be empty for M to accept. We conclude the sketch of the proof by recalling that context-free languages are closed under the mirror image. □

The proof of Lemma 2 can be directly generalized to register machines with forbidden regions, yielding the following corollary for the generating mode.

Theorem 10. *Consider two alphabets V and R, the corresponding free groups $\mathcal{V}$ and $\mathcal{R}$, and the regions $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$ and $F_{\mathcal{R}} = \mathcal{R} \setminus R^*$. Then $(\mathcal{V}, \mathcal{R})$-register machines with terminal zero test and with the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{R}})$ generate all context-free languages that can be accepted by a pushdown automaton with the tape alphabet V and the stack alphabet R.*

Finally, we remark that two registers containing strings over the same alphabet can be used to directly simulate the tape of a Turing machine. Our construction is very similar to an automaton with two independent stacks.

Without losing generality, we will only consider Turing machines whose input is placed entirely to the left of their head.

Theorem 11. *Consider the alphabet V, the free group $\mathcal{V}$ over V, and the region $F_{\mathcal{V}} = \mathcal{V} \setminus V^*$. $(\mathcal{V}, \mathcal{V})$-register machines with the zero test instruction and the family of forbidden regions $(F_{\mathcal{V}}, F_{\mathcal{V}})$ can directly simulate a Turing machine by starting with the initial tape contents in the first register and halting with the final tape contents in the first register.*

Proof (sketch). A register machine M simulates a given Turing machine T by keeping the string representing the tape contents to the left of the head in the first register, and the reverse of the tape contents to the right of the head in the second register. At every step, M checks if the second register is not empty, and if not, non-deterministically guesses the symbol T is reading. To write a symbol a on the tape, M simply performs $ADD(2, a)$. To simulate a move of the head of T to the left, M non-deterministically reads a symbol from the first register and adds it to the second register. To simulate a move to the right, M non-deterministically reads a symbol a from the second register and adds a to the first register. If the second register is empty, M simulates the action of T corresponding to reading an empty tape cell. If the head should move to the right, M adds the symbol representing the empty tape cell (different from the empty string λ) to the first register. Similarly, if M must simulate a move of the head to the left while its first register is empty, it adds the symbol representing the empty tape cell to the second register. □

6 Conclusion and Open Problems

In this paper we focused on generalizing the model of register machines to operate on groups instead of natural or integer numbers, thus continuing previous works

aiming at generalizing related models of computing, such as vector addition systems and P systems [2, 3, 7, 9]. Generalizing register machines to groups allowed us to put forward the fundamental connection between vector addition systems and register machines, as well as to reveal an unexpected possibility to operate on registers containing strings, without any encoding.

The definitions and basic tools exhibited in this paper illustrate some of the consequences of the way in which register machines are generalized. One interesting class of problems which is still left open is the role of the nature of the underlying group in defining the frontiers of computational power. For example, Theorem 6 approaches one such separation between vector addition systems with and without states, but does not give a crisp borderline. On the other hand, the impact of the group being commutative is still to be explored.

References

1. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: Jürgensen, H., Karhumäki, J., Okhotin, A. (eds.) Descriptional Complexity of Formal Systems – 16th International Workshop, DCFS 2014, Turku, Finland, August 5-8, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8614, pp. 65–76. Springer (2014). https://doi.org/10.1007/978-3-319-09704-6_7
2. Alhazov, A., Belingheri, O., Freund, R., Ivanov, S., Porreca, A.E., Zandron, C.: Purely catalytic P systems over integers and their generative power. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10105, pp. 67–82. Springer (2016). https://doi.org/10.1007/978-3-319-54072-6_5
3. Alhazov, A., Belingheri, O., Freund, R., Ivanov, S., Porreca, A.E., Zandron, C.: Semilinear sets, register machines, and integer vector addition (p) systems. In: Proceedings of the 17th International Conference on Membrane Computing, CMC 2016. pp. 27–42 (2016)
4. Büning, H.K., Lettmann, T., Mayr, E.W.: Projections of vector addition system reachability sets are semilinear. Theoretical Computer Science **64**(3), 343–350 (May 1989). https://doi.org/10.1016/0304-3975(89)90055-8
5. Freund, R.: Control mechanisms for array grammars on Cayley grids. In: Durand-Lose, J., Verlan, S. (eds.) Machines, Computations, and Universality – 8th International Conference, MCU 2018, Fontainebleau, France, June 28-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10881, pp. 1–33. Springer (2018). https://doi.org/10.1007/978-3-319-92402-1_1
6. Freund, R., Ibarra, O.H., Păun, Gh., Yen, H.C.: Matrix languages, register machines, vector addition systems. In: Gutiérrez-Naranjo, M., Riscos-Núñez, A., Romero Campero, F., Sburlan, D. (eds.) Proceedings of the Third Brainstorming Week on Membrane Computing. pp. 155–168. University of Sevilla (2005)
7. Freund, R., Ivanov, S., Verlan, S.: P systems with generalized multisets over totally ordered abelian groups. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing – 16th International Conference, CMC 2015, Valencia, Spain, August 17–21, 2015, Revised Selected Papers. Lecture Notes in Computer

Science, vol. 9504, pp. 117–136. Springer (2015). https://doi.org/10.1007/978-3-319-28475-0_9
8. Greibach, S.A.: Remarks on blind and partially blind one-way multicounter machines. Theoretical Computer Science **7**(3), 311–324 (1978). https://doi.org/10.1016/0304-3975(78)90020-8
9. Haase, C., Halfon, S.: Integer vector addition systems with states. In: Ouaknine, J., Potapov, I., Worrell, J. (eds.) Reachability Problems: 8th International Workshop, RP 2014, Oxford, UK, September 22–24, 2014. Proceedings, pp. 112–124. Springer (2014). https://doi.org/10.1007/978-3-319-11439-2_9
10. Holt, D.F., Eick, B., O'Brien, E.A.: Handbook of Computational Group Theory. CRC Press (2005)
11. Hopcroft, J., Pansiot, J.J.: On the reachability problem for 5-dimensional vector addition systems. Theoretical Computer Science **8**(2), 135–159 (1979). https://doi.org/10.1016/0304-3975(79)90041-0
12. Ivanov, S.: On the power and universality of biologically-inspired models of computation. (Étude de la puissance d'expression et de l'universalité des modèles de calcul inspirés par la biologie). Ph.D. thesis, University of Paris-Est, France (2015), `https://tel.archives-ouvertes.fr/tel-01272318`
13. Korec, I.: Small universal register machines. Theoretical Computer Science **168**(2), 267–301 (1996). https://doi.org/10.1016/S0304-3975(96)00080-1
14. Levi, F.W.: Ordered groups. In: Proceedings of the Indian Academy of Sciences. vol. A16, pp. 256–263 (1942)
15. Minsky, M.: Recursive unsolvability of Post's problem of tag and other topics in the theory of Turing machines. Annals of Mathematics, second series **74**, 437–455 (1961)
16. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
17. Mitrana, V., Stiebe, R.: Extended finite automata over groups. Discrete Applied Mathematics **108**(3), 287 –300 (2001). https://doi.org/10.1016/S0166-218X(00)00200-6
18. Păun, Gh., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
19. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages, 3 volumes. Springer, New York, NY, USA (1997)
20. Thomas W. Judson, R.A.B.: Abstract Algebra: Theory and Applications (2018)

(Tissue) P Systems with Anti-Membranes

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

[1] Vladimir Andrunachievici Institute of
Mathematics and Computer Science
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md

[2] TU Wien, Institut für Logic and Computation
Favoritenstraße 9–11, 1040 Wien, Austria
E-mail: rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23, boulevard de France, 91034 Évry, France
E-mail: sergiu.ivanov@univ-evry.fr

Summary. The concept of a matter object being annihilated when meeting its corresponding anti-matter object is taken over for membranes as objects and anti-membranes as the corresponding annihilation counterpart in P systems. Natural numbers can be represented by the corresponding number of membranes with a specific label. Computational completeness in this setting then can be obtained with using only elementary membrane division rules, without using objects. A similar result can be obtained for tissue P systems with cell division rules and cell / anti-cell annihilation rules. In both cases, as derivation modes we may take the standard maximally parallel derivation modes as well as any of the maximally parallel set derivation modes (non-extendable (multi)sets of rules, (multi)sets with maximal number of rules, (multi)sets of rules affecting the maximal number of objects).

1 Introduction

The basic model of *P systems* as introduced in [12] can be considered as a distributed multiset rewriting system, where all objects – if possible – evolve in parallel in the membrane regions and may be communicated through the membranes. Overviews on the field of P systems can be found in the monograph [13] and the handbook of membrane systems [14]; for actual news and results we refer to the P systems webpage [16] as well as to the Bulletin of the International Membrane Computing Society.

Computational completeness (computing any partial recursive relation on non-negative integers) can be obtained with using cooperative rules or with catalytic rules (possibly) together with non-cooperative rules. We recall that non-cooperative rules have the form $a \to w$, where a is a symbol and w is a multiset,

catalytic rules have the form $ca \to cw$, where the symbol c is called the catalyst, and cooperative rules have no restrictions on the form of the left-hand side. Without additional control mechanisms, at least two catalysts are needed, see [7]. Using specific control mechanisms, as for example, rule labels or target agreement, only one catalyst is needed, for example, see [6, 8, 9]. In [2, 1], another concept to avoid cooperative rules is investigated: for any object *a (matter)*, its anti-object *(anti-matter)* a^- is considered together with the corresponding *annihilation rule* $aa^- \to \lambda$, which is assumed to exist in all membranes; this annihilation rule is assumed to be a special non-cooperative rule having priority over all other rules in the sense of weak priority (e.g., see [3], i.e., other rules then also may be applied if objects cannot be bound by some annihilation rule any more). For spiking neural P systems, the idea of anti-matter has been introduced in [11] with *anti-spikes* as anti-matter objects. In [5] the power of anti-matter for solving NP-complete problems is exhibited.

Although, as expected (for example, compare with the Geffert normal forms, see [15]), the annihilation rules are rather powerful, it is still surprising that using matter/anti-matter annihilation rules as the only non-cooperative rules, with the annihilation rules having weak priority, computational completeness can already be obtained without using any catalyst, see [2, 1], whereas usually at least one catalyst is needed even when using other control mechanisms, for example, see [2].

Natural numbers can be represented by the corresponding number of membranes with a specific label. Hence, in this paper we take over the idea of anti-objects for membranes, i.e., for every membrane $[\ ]_h$ we take the anti-membrane $[\ ]_{h^-}$ and the membrane / anti-membrane annihilation rule $[\ ]_h[\ ]_{h^-} \to \lambda$. In the simplest case, we only use elementary membranes, but no objects, and elementary membrane division, i.e., rules of the form $[\ ]_h \to [\ ]_{h'}[\ ]_{h''}$, possibly also allowing membrane renaming rules of the form $[\ ]_h \to [\ ]_{h'}$ or membrane deletion rules of the form $[\ ]_h \to \lambda$. In this setting, computational completeness then can be obtained with using only elementary membrane division rules, without using objects, together with anti-membranes and membrane / anti-membrane annihilation rules.

Natural numbers can also be represented by the corresponding number of cells with a specific label. Hence, a similar computational completeness result can also be obtained for tissue P systems with cell division rules and cell / anti-cell annihilation rules.

In both cases, as derivation modes we may take the standard maximally parallel derivation modes as well as any of the maximally parallel set derivation modes (non-extendable (multi)sets of rules, (multi)sets with maximal number of rules, (multi)sets of rules affecting the maximal number of objects).

2 Prerequisites

The set of integers is denoted by $\mathbb{Z}$, and the set of non-negative integers by $\mathbb{N}$. Given an alphabet V, a finite non-empty set of abstract symbols, the free monoid

generated by V under the operation of concatenation is denoted by V^*. The elements of V^* are called strings, the empty string is denoted by λ, and $V^*\backslash\{\lambda\}$ is denoted by V^+. For an arbitrary alphabet $V = \{a_1, \ldots, a_n\}$, the number of occurrences of a symbol a_i in a string x is denoted by $|x|_{a_i}$, while the length of a string x is denoted by $|x| = \sum_{a_i \in V} |x|_{a_i}$. The Parikh vector associated with x with respect to $a_1, \ldots, a_n$ is $(|x|_{a_1}, \ldots, |x|_{a_n})$. The Parikh image of an arbitrary language L over $\{a_1, \ldots, a_n\}$ is the set of all Parikh vectors of strings in L, and is denoted by $Ps(L)$. For a family of languages FL, the family of Parikh images of languages in FL is denoted by $PsFL$, while for families of languages over a one-letter (d-letter) alphabet, the corresponding sets of non-negative integers (d-vectors with non-negative components) are denoted by NFL (N^dFL).

A (finite) multiset over a (finite) alphabet $V = \{a_1, \ldots, a_n\}$, is a mapping $f : V \to \mathbb{N}$ and can be represented by $\langle a_1^{f(a_1)}, \ldots, a_n^{f(a_n)} \rangle$ or by any string x for which $(|x|_{a_1}, \ldots, |x|_{a_n}) = (f(a_1), \ldots, f(a_n))$. In the following we will not distinguish between a vector $(m_1, \ldots, m_n)$, a multiset $\langle a_1^{m_1}, \ldots, a_n^{m_n} \rangle$ or a string x having $(|x|_{a_1}, \ldots, |x|_{a_n}) = (m_1, \ldots, m_n)$. Fixing the sequence of symbols $a_1, \ldots, a_n$ in an alphabet V in advance, the representation of the multiset $\langle a_1^{m_1}, \ldots, a_n^{m_n} \rangle$ by the string $a_1^{m_1} \ldots a_n^{m_n}$ 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 [4] and [15].

Register machines

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

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B\backslash\{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Increases the value of register j by one, followed by a non-deterministic jump to instruction l_2 or l_3. This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$, with $l_1 \in B\backslash\{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  If the value of register j is zero then jump to instruction l_3; otherwise, the value of register j is decreased by one, followed by a jump to instruction l_2. The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$. Stops 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 instruction l_0 of P, and terminate with reaching the HALT-instruction l_h.

For useful results on the computational power of register machines, we refer to [10].

3 P Systems with Active Membranes and Anti-Membranes

For using anti-matter as a frontier of tractability, we refer to [5], where some standard definition of P systems with active membranes can be found. We here consider a special rather restricted model, where no objects are used and inside the skin membrane only the following types of rules for elementary membranes are used:

elementary membrane division $[\,]_h \to [\,]_{h'}[\,]_{h''}$
the elementary membrane $[\,]_h$ is divided into two membranes, possibly changing the label h of the parent membrane $[\,]_h$ to two new labels h', h'' for the child membranes $[\,]_{h'}$ and $[\,]_{h''}$

changing membrane label $[\,]_h \to [\,]_{h'}$
the label h of the elementary membrane $[\,]_h$ is changed to h'

elementary membrane deletion $[\,]_h \to \lambda$
the elementary membrane $[\,]_h$ is deleted

membrane / anti-membrane annihilation $[\,]_h[\,]_{h^-} \to \lambda$
the elementary membrane $[\,]_h$ and its corresponding anti-membrane $[\,]_{h'}$ annihilate each other

Formally, a *P system with active membranes and anti-membranes* (a *PAMS* for short) is a construct $\Pi = (H \cup \{0\}, [\,]_0, w_0, R)$ where H is the set of *membrane labels* used in the membrane rules specified in R, $[\,]_0$ denotes the skin membrane enclosing the initial set of elementary membranes w_0 with labels from H, and R is the set of rules of the forms described above, with the labels of the elementary membranes taken from H.

In any computation step of Π a multiset of rules is chosen from the set R 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 membranes in the skin membrane. We emphasize that membrane / anti-membrane annihilation rules have weak priority over all other rules, i.e., as long as membrane / anti-membrane annihilation rules may bind some membranes, other rules are not allowed to yet be taken into the multiset of rules constructed to be maximal.

A *configuration* of the system can be represented by the membranes inside the skin membrane. Starting from a given *initial configuration* and applying evolution rules as described 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 any more.

In the *generative case*, a halting computation has associated a result, in the form of the number of membranes with the same labels present in the skin membrane; their numbers represents a vector of natural numbers. In the *accepting case*, all (vectors of) non-negative integers are accepted whose input, given as the corresponding numbers of membranes in the skin membrane in addition to w_0, leads

to a halting computation. The set of non-negative integers and the set of (Parikh) vectors of non-negative integers generated/accepted as results of halting computations in Π are denoted by $N_\delta(\Pi)$ and $Ps_\delta(\Pi)$, respectively, with $\delta \in \{gen, acc\}$. The corresponding families of sets of non-negative integers and the sets of vectors of non-negative integers generated/accepted by PAMSs are denoted by N_δ(PAMS) and Ps_δ(PAMS), respectively.

4 Tissue P Systems with Cell Division and Anti-Cells

Instead of considering elementary membranes inside the skin membrane, we may also consider cells floating in a common environment. Then instead of anti-membranes, we consider anti-cells, i.e., cells with the anti-label. Again, we here consider a special rather restricted model, where no objects are used and only the following types of rules for cells in the tissue P system are used:

cell division $\bigcirc_h \rightarrow \bigcirc_{h'} \bigcirc_{h''}$
the cell $\bigcirc_h$ is divided into two cells, possibly changing the label h of the parent cell $\bigcirc_h$ to two new labels h', h'' for the child cells $\bigcirc_{h'}$ and $\bigcirc_{h''}$

changing cell label $\bigcirc_h \rightarrow \bigcirc_{h'}$
the label h of cell $\bigcirc_h$ is changed to h'

cell deletion $\bigcirc_h \rightarrow \lambda$
the cell $[\ ]_h$ is deleted

cell / anti-cell annihilation $\bigcirc_h \bigcirc_{h^-} \rightarrow \lambda$
the cell $\bigcirc_h$ and its corresponding anti-cell $\bigcirc_{h^-}$ annihilate each other

Formally, a *tissue P system with anti-cells* (a *tPAMS* for short) is a construct $\Pi = (H, w_0, R)$ where H is the set of *cell labels* used in the rules specified in R, w_0 is the initial set of cells with labels from H, and R is the set of rules of the forms described above, with the labels of the cells taken from H.

In any computation step of Π a multiset of rules is chosen from the set R 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 cells. We emphasize that again we assume cell / anti-cell annihilation rules to have weak priority over all other rules, i.e., as long as cell / anti-cell annihilation rules may bind some cells, other rules are not allowed to yet be taken into the multiset of rules constructed to be maximal.

A *configuration* of the system can be represented by the currently existing cells. Starting from a given *initial configuration* and applying evolution rules as described 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 any more.

In the *generative case*, a halting computation has associated a result, in the form of the number of cells present in the system; their numbers represents a vector of natural numbers. In the *accepting case*, all (vectors of) non-negative integers are accepted whose input, given as the corresponding numbers of initial cells in addition to w_0, leads to a halting computation. The set of non-negative integers and the set of (Parikh) vectors of non-negative integers generated/accepted as results of halting computations in Π are denoted by $N_\delta(\Pi)$ and $Ps_\delta(\Pi)$, respectively, with $\delta \in \{gen, acc\}$. The corresponding families of sets of non-negative integers and the sets of vectors of non-negative integers generated/accepted by tPAMSs are denoted by N_δ(tPAMS) and Ps_δ(tPAMS), respectively.

5 Results

As a first result, we observe that rules changing membrane label, i.e., $[\,]_h \to [\,]_{h'}$, and elementary membrane deletion rules, i.e., $[\,]_h \to \lambda$, are not needed and can be replaced by using only elementary membrane division and suitable membrane / anti-membrane annihilation rules.

Lemma 1. *Rules changing membrane label, i.e.,* $[\,]_h \to [\,]_{h'}$*, and elementary membrane deletion rules, i.e.,* $[\,]_h \to \lambda$*, can be simulated by elementary membrane division and membrane / anti-membrane annihilation rules.*

Proof. A rule changing the membrane label, i.e., $[\,]_h \to [\,]_{h'}$, can be simulated by the rules $[\,]_h \to [\,]_{h'}[\,]_{h''}$, $[\,]_{h''} \to [\,]_g[\,]_{g^-}$, and $[\,]_g[\,]_{g^-} \to \lambda$, where h'', g, g^- are new labels (separately for each label h).

An elementary membrane deletion rule, i.e., $[\,]_h \to \lambda$, can be simulated by the rules $[\,]_h \to [\,]_g[\,]_{g^-}$ and $[\,]_g[\,]_{g^-} \to \lambda$, where g, g^- are new labels (separately for each label h). □

A similar result obviously also holds for tPAMS: rules changing a cell label, i.e., $\bigcirc_h \to \bigcirc_{h'}$, and cell deletion rules, i.e., $\bigcirc_h \to \lambda$, are not needed and can be replaced by using only cell division and suitable cell / anti-cell annihilation rules. The corresponding proof verbatim follows the proof of Lemma 1, just replacing the notation $[\,]_h$ by $\bigcirc_h$.

Corollary 1. *Rules changing cell label, i.e.,* $\bigcirc_h \to \bigcirc_{h'}$*, and cell deletion rules, i.e.,* $\bigcirc_h \to \lambda$*, can be simulated by cell division and cell / anti-cell annihilation rules.*

A PAMS only using elementary membrane division and membrane / anti-membrane annihilation rules is called a *PAMS in normal form*. As an immediate consequence of Lemma 1 we obtain the following normal form theorem:

Theorem 1. *For every PAMS* Π *we can construct a PAMS* Π' *in normal form such that* $N_\delta(\Pi) = N_\delta(\Pi')$ *and* $Ps_\delta(\Pi) = Ps_\delta(\Pi')$*, with* $\delta \in \{gen, acc\}$.

A similar normal form result obviously also holds for tPAMS as an immediate consequence of Corollary 1:

Corollary 2. *For every tPAMS Π we can construct a tPAMS Π' in normal form such that $N_\delta(\Pi) = N_\delta(\Pi')$ and $Ps_\delta(\Pi) = Ps_\delta(\Pi')$, with $\delta \in \{gen, acc\}$.*

5.1 Computational Completeness

We now show that PAMSs characterize the families NRE and $PsRE$, respectively. The main proof idea – as used very often in the area of P systems – is to simulate (the computations of) register machines, as carried out in a similar way in [1] for P systems with anti-matter.

Theorem 2. *For any $Y \in \{N, Ps\}$ and $\delta \in \{gen, acc\}$,*

$$Y_\delta(PAMS) = YRE.$$

Proof. Let $M = (m, B, l_0, l_h, P)$ be a register machine. We now construct a PAMS Π which simulates (the computations of) M:

- $\Pi = (H \cup \{0\}, [\,]_0, w_0, R)$;
- $H = \{r, r^- \mid 1 \leq r \leq m\} \cup \{l, l' \mid l \in B\} \cup \{\#, \#^-\}$ is the set of labels for the elementary membranes inside the skin membrane;
  the label $r, 1 \leq r \leq m$, is for the copies of membrane $[\,]_r$ representing the contents of register r; the labels r^- are for the corresponding anti-membranes;
- in the generating case, initially the skin membrane contains only the elementary membrane $[\,]_{l_0}$; in the accepting case, suitable copies of membranes for representing the input vector are to be added;
- R contains the rules described in the following.

The contents of register r is represented by the number of copies of the elementary membrane $[\,]_r$, $1 \leq r \leq m$, and for each membrane $[\,]_r$ we also consider the corresponding anti-membrane $[\,]_{r^-}$.

The instructions of M are simulated by the following rules in R_1:

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Simulated by the rules
  $$[\,]_{l_1} \rightarrow [\,]_r [\,]_{l_2} \text{ and } [\,]_{l_1} \rightarrow [\,]_r [\,]_{l_3}.$$

- $l_1 : (SUB(r), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq r \leq m$.
  As rules common for the simulations of all SUB-instructions, we have
  $$[\,]_{r^-} \rightarrow [\,]_{\#^-}, \; 1 \leq r \leq m,$$
  and the annihilation rules
  $$[\,]_r [\,]_{r^-} \rightarrow \lambda, \; 1 \leq r \leq m, \text{ and } [\,]_\# [\,]_{\#^-} \rightarrow \lambda$$

as well as the trap rules

$$[\,]_{\#^-} \to [\,]_{\#}[\,]_{\#} \text{ and } [\,]_{\#} \to [\,]_{\#}[\,]_{\#};$$

these last two rules lead the system into an infinite computation whenever a membrane with one of the trap symbols # or $\#^-$ is left without being annihilated.

The *zero test* for instruction l_1 is simulated by the rules

$$[\,]_{l_1} \to [\,]_{l_1'}\,[\,]_{r^-} \text{ and } [\,]_{l_1'} \to [\,]_{\#}[\,]_{l_3}.$$

The membrane labeled by #, generated by the second rule $[\,]_{l_1'} \to [\,]_{\#}[\,]_{l_3}$ can only be eliminated if the anti-membrane $[\,]_{r^-}$ generated by the first rule $[\,]_{l_1} \to [\,]_{l_1'}\,[\,]_{r^-}$ is not annihilated by $[\,]_r$, i.e., only if register r is empty, which allows for applying the rule $[\,]_{r^-} \to [\,]_{\#^-}$ and for using the annihilation rule $[\,]_{\#}[\,]_{\#^-} \to \lambda$ afterwards in the next derivation step.

The *decrement case* for instruction l_1 is simulated by the rule

$$[\,]_{l_1} \to [\,]_{l_2}[\,]_{r^-}.$$

The anti-membrane $[\,]_{r^-}$ either correctly annihilates one copy of membrane $[\,]_r$, thus decrementing the register r, or else traps an incorrect guess by forcing the anti-membrane $[\,]_{r^-}$ to evolve to $[\,]_{\#^-}$ and then to $[\,]_{\#}[\,]_{\#}$ in the next two steps in case register r is empty.

We finally observe that these two remaining derivation steps for trapping the decrement case as well as the remaining derivation step for correctly completing the decrement case or the zero test case do not influence the correct simulation of another SUB-instruction, even on the same register r, as the involved symbols have evolved at least one step before they could influence the symbols being generated by the new simulation sequence.

- $l_h : HALT$. Simulated by $[\,]_{l_h} \to \lambda$.

When the computation in M halts, the membrane $[\,]_{l_h}$ is removed, and no further rules can be applied provided the simulation has been carried out correctly, i.e., if no membranes labeled by trap symbols # are present in this situation. The remaining membranes in the system represent the result computed by M. □

For $\delta \in \{gen, acc\}$, let us denote the families of sets of non-negative integers and the sets of vectors of non-negative integers generated/accepted by PAMSs in normal form by N_δ(NFPAMS) and Ps_δ(NFPAMS), respectively.

Then, by combining Lemma 1 and Theorem 2, we obtain the following result:

Theorem 3. *For any $Y \in \{N, Ps\}$ and $\delta \in \{gen, acc\}$,*

$$Y_\delta(NFPAMS) = YRE.$$

Similar results obviously also hold for tissue P systems with anti-cells; the corresponding proofs again verbatim follow the proofs of Theorems 2 and 3, just replacing the notation $[\ ]_h$ by $\bigcirc_h$.

Corollary 3. *For any* $Y \in \{N, Ps\}$ *and* $\delta \in \{gen, acc\}$,

$$Y_\delta(PAMS) = YRE.$$

Corollary 4. *For any* $Y \in \{N, Ps\}$ *and* $\delta \in \{gen, acc\}$,

$$Y_\delta(NFPAMS) = YRE.$$

5.2 Derivation Modes

So far, we only have considered the maximally parallel derivation mode. Yet a thorough investigation of the proofs given so far in this section shows that in a successful derivation each rule need only be applied at most once, which means that instead of the maximally parallel derivation mode we can use any of the set derivation modes, where each rule can only be applied once, defined as follows:

$setmax$ take a non-extendable set of rules
$setmax_{rules}$ take a non-extendable set of rules with the maximal number of rules possible
$setmax_{objects}$ take a non-extendable set of rules affecting the maximal number of objects

The concept of using the maximal number of rules or objects can also be taken over for the maximally parallel derivation mode:

max take a non-extendable multiset of rules
max_{rules} take a non-extendable multiset of rules with the maximal number of rules possible
$max_{objects}$ take a non-extendable multiset of rules affecting the maximal number of objects

Let us now specify the derivation mode

$$\gamma \in \{max, max_{rules}, max_{objects}, setmax, setmax_{rules}, setmax_{objects}, \}$$

as additional subscript to δ, $\delta \in \{gen, acc\}$, for denoting the set of natural numbers and the set of vectors of natural numbers obtained by PAMS and tPAMS, i.e., we now write $N_{\gamma,\delta}$ and $Ps_{\gamma,\delta}$, respectively.

Moreover, we use the bracket notation [t]PAMS to indicate that we mean both PAMS and tPAMS, respectively, and in a similar way for PAMS and tPAMS in normal form.

With any of these derivation modes, using sets or multisets of rules, we now get the same normal form and computational completeness results as for the maximally parallel derivation mode max as established so far:

For the [t]PAMS to be transformed into normal form we observe that in the construction of the normal form given in the proof of Lemma 1, for each membrane label we used new additional labels and thus the corresponding new rules are independent from other such rules needed for simulating the change of a membrane label or the deletion of a membrane; hence, if in one of the set modes, one such rule is replaced to get the normal form, all the simulating rules are also needed only once, too, during the simulation sequences.

Hence, we can summarize the results obtained in this paper in the following form, for any of the derivation modes defined above.

We first state our normal form theorem:

Theorem 4. *For every [t]PAMS Π we can construct a [t]PAMS Π' in normal form such that*

$$Y_{\gamma,\delta}(\Pi) = Y_{\gamma,\delta}(\Pi')$$

for any $Y \in \{N, Ps\}$ and any $\delta \in \{gen, acc\}$ as well as any

$$\gamma \in \{max, max_{rules}, max_{objects}, setmax, setmax_{rules}, setmax_{objects}\}.$$

As our main result, we have shown computational completeness for PAMS and tPAMS, even in normal form, with all the derivation modes as defined above: we again emphasize that in the proofs given so far in this section, in a successful derivation each rule need only be applied at most once, hence, the simulations of the instructions of a register machine work for the set modes as well. On the other hand, whether the trap rule $[\ ]_{\#} \rightarrow [\ ]_{\#}[\ ]_{\#}$ is applied only once or as often as possible makes no difference for the desired effect to keep the system trapped in an infinite loop.

Theorem 5. *For any $Y \in \{N, Ps\}$ and $\delta \in \{gen, acc\}$,*

$$Y_{\gamma,\delta}([NF][t]PAMS) = YRE$$

for any

$$\gamma \in \{max, max_{rules}, max_{objects}, setmax, setmax_{rules}, setmax_{objects}\}.$$

Finally we mention that computational completeness can also be extended from the generating and accepting case to the computing case, i.e., PAMS and tPAMS, even in normal form, can also compute any partial recursive function or relation.

6 Conclusion

In this paper we have taken over the idea of matter and anti-matter objects in P systems to P systems with active membranes, now considering membranes and

anti-membranes as the objects interacting with each other in annihilation rules, which we assume to have weak priority over all other rules. We have investigated a restricted model of P systems with active membranes, without any objects in the whole system and instead only elementary membranes in the skin membrane. In this model, natural numbers are represented as copies of elementary membranes with a specific label. In such a variant of P systems with active membranes, computations of register machines can be simulated by using only (a special variant of) elementary membrane division rules and membrane/anti-membrane annihilation rules.

Moreover, we have established similar results for tissue P systems with cell division rules and cell / anti-cell annihilation rules. In both cases, as derivation modes we may also take the standard maximally parallel derivation mode(s) as well as any of the maximally parallel set derivation modes (non-extendable (multi)sets of rules, (multi)sets with maximal number of rules, (multi)sets of rules affecting the maximal number of objects) to obtain computational completeness.

In a more general model, we need not restrict ourselves to elementary membranes interacting with each other in membrane / anti-membrane annihilation rules. In fact, we may consider a variant where in such a reaction only the outermost membranes of two non-elementary membranes react, emitting the interior membrane structure into the skin membrane. In such a variant, non-elementary membrane division becomes relevant, as well as rules allowing for putting a new membrane around a given membrane structure, i.e., rules of the form $[\ ]_h \rightarrow [\ [\ ]_{h'}]_{h''}$. Finally, as it is common in P systems with active membranes, in addition objects may be added and guide the membrane rules (yet still evolution rules for the objects may be forbidden). Such variants remain to be investigated in some future papers based on this one.

Acknowledgements

The ideas for this paper came up in the inspiring atmosphere of the Brainstorming Week on Membrane Computing in Sevilla this year.

References

1. Alhazov, A., Aman, B., Freund, R.: P systems with anti-matter. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing - 15th International Conference, CMC 2014, Prague, Czech Republic, August 20-22, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8961, pp. 66–85. Springer (2014). https://doi.org/10.1007/978-3-319-14370-5_5
2. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: Macías-Ramos, L.F., Martínez-del-Amor, M.A., Păun, Gh., Riscos-Núñez, A., Valencia-Cabrera, L. (eds.) Proceedings of the Twelfth Brainstorming Week on Membrane Computing. pp. 1–26 (2014), http://www.gcn.us.es/files/12bwmc/001_bwmc2014AntiMatter.pdf

3. Alhazov, A., Sburlan, D.: Static sorting P systems. In: Ciobanu, G., Pérez-Jiménez, M.J., Păun, Gh. (eds.) Applications of Membrane Computing, pp. 215–252. Natural Computing Series, Springer (2006). https://doi.org/10.1007/3-540-29937-8_8
4. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer (1989), https://www.springer.com/de/book/9783642749346
5. Díaz-Pernil, D., Peña-Cantillana, F., Alhazov, A., Freund, R., Gutiérrez-Naranjo, M.A.: Antimatter as a frontier of tractability in membrane computing. Fundamenta Informaticae **134**(1-2), 83–96 (2014). https://doi.org/10.3233/FI-2014-1092
6. Freund, R.: Purely catalytic P systems: Two catalysts can be sufficient for computational completeness. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu. (eds.) CMC14 Proceedings – The 14th International Conference on Membrane Computing, Chişinău, August 20–23, 2013. pp. 153–166. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova (2013), http://www.math.md/cmc14/CMC14_Proceedings.pdf
7. Freund, R., Kari, L., Oswald, M., Sosík, P.: Computationally universal P systems without priorities: two catalysts are sufficient. Theoretical Computer Science **330**(2), 251–266 (2005). https://doi.org/10.1016/j.tcs.2004.06.029
8. Freund, R., Oswald, M.: Catalytic and purely catalytic P automata: control mechanisms for obtaining computational completeness. In: Bensch, S., Drewes, F., Freund, R., Otto, F. (eds.) Fifth Workshop on Non-Classical Models for Automata and Applications - NCMA 2013, Umeå, Sweden, August 13 - August 14, 2013, Proceedings. books@ocg.at, vol. 294, pp. 133–150. Österreichische Computer Gesellschaft (2013)
9. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9-11, 2013. EPTCS, vol. 128, pp. 47–61 (2013). https://doi.org/10.4204/EPTCS.128.13
10. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
11. Pan, L., Păun, Gh.: Spiking neural P systems with anti-matter. International Journal of Computers, Communications & Control **4**(3), 273–282 (2009). https://doi.org/10.15837/ijccc.2009.3.2435, http://univagora.ro/jour/index.php/ijccc/article/download/2435/901
12. Păun, Gh.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
13. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002). https://doi.org/10.1007/978-3-642-56196-2
14. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
15. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer (1997). https://doi.org/10.1007/978-3-642-59136-5
16. The P Systems Website. http://ppage.psystems.eu/

P Systems: from Anti-Matter to Anti-Rules

Artiom Alhazov[1], Rudolf Freund[2], Sergiu Ivanov[3], Mario J. Pérez-Jiménez[4]

[1] Vladimir Andrunachievici Institute of
Mathematics and Computer Science
Academiei 5, Chişinău, MD-2028, Moldova
E-mail: artiom@math.md

[2] TU Wien, Institut für Logic and Computation
Favoritenstraße 9–11, 1040 Wien, Austria
E-mail: rudi@emcc.at

[3] IBISC, Université Évry, Université Paris-Saclay
23, boulevard de France, 91034 Évry, France
E-mail: sergiu.ivanov@univ-evry.fr

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

Summary. The concept of a matter object being annihilated when meeting its corresponding anti-matter object is taken over for rule labels as objects and anti-rule labels as the corresponding annihilation counterpart in P systems. In the presence of a corresponding anti-rule object, annihilation of a rule object happens before the rule that the rule object represents, can be applied. Applying a rule consumes the corresponding rule object, but may also produce new rule objects as well as anti-rule objects, too. Computational completeness in this setting then can be obtained in a one-membrane P system with non-cooperative rules and rule / anti-rule annihilation rules when using one of the standard maximally parallel derivation modes as well as any of the maximally parallel set derivation modes (i.e., non-extendable (multi)sets of rules, (multi)sets with maximal number of rules, (multi)sets of rules affecting the maximal number of objects). When using the sequential derivation mode, at least the computational power of partially blind register machines is obtained.

1 Introduction

The basic model of *P systems* as introduced in [24] can be considered as a distributed multiset rewriting system, where all objects – if possible – evolve in parallel in the membrane regions and may be communicated through the membranes. Overviews on the field of P systems can be found in the monograph [25] and the handbook of membrane systems [26]; for actual news and results we refer to the

P systems webpage [28] as well as to the Bulletin of the International Membrane Computing Society.

Computational completeness (computing any partial recursive relation on non-negative integers) can be obtained with using cooperative rules or with catalytic rules (possibly) together with non-cooperative rules. We recall that non-cooperative rules have the form $a \to w$, where a is a symbol and w is a multiset, catalytic rules have the form $ca \to cw$, where the symbol c is called the catalyst, and cooperative rules have no restrictions on the form of the left-hand side. Without additional control mechanisms, at least two catalysts are needed, see [16]. Using specific control mechanisms, as for example, rule labels or target agreement, only one catalyst is needed, for example, see [14, 19, 20]. In [2, 1], another concept to avoid cooperative rules is investigated: for any object *a (matter)*, its anti-object *(anti-matter)* a^- is considered together with the corresponding *annihilation rule* $aa^- \to \lambda$, which is assumed to exist in all membranes; this annihilation rule is assumed to be a special non-cooperative rule having priority over all other rules in the sense of weak priority (e.g., see [9], i.e., other rules then also may be applied if objects cannot be bound by some annihilation rule any more). For spiking neural P systems, the idea of anti-matter has been introduced in [23] with *anti-spikes* as anti-matter objects. In [12] the power of anti-matter for solving NP-complete problems is exhibited.

Although, as expected (for example, compare with the Geffert normal forms, see [27]), the annihilation rules are rather powerful, it is still surprising that using matter/anti-matter annihilation rules as the only non-cooperative rules, with the annihilation rules having weak priority, computational completeness can already be obtained without using any catalyst, see [2, 1], whereas usually at least one catalyst is needed even when using other control mechanisms, for example, see [2].

In this paper we now consider a rule label as an object itself which cannot evolve as any other object in the system, but only has the task to make the rule applicable; in some sense this can be seen as a variant of rule activation as introduced in P systems with activation and blocking of rules, see [4]; for the concept of activation and blocking of rules also see [3, 5]. Introducing the anti-rule object then can be seen as a variant of blocking the corresponding rule. The main difference between these two concepts – activation and blocking of rules in contrast to rule objects and anti-rule objects – is that with activation and blocking of rules, rules can be activated and blocked for specific time steps in the future, whereas the activation of a rule by its rule object is immediate, and also blocking is immediate, but active until the anti-rule object annihilates with the rule object, which may be an unbounded number of steps in the future. When a rule is applied by consuming its corresponding rule object is also not fixed and may heavily depend not only on the applicability of the rule, but also on the derivation mode. For example, in the sequential mode, several rule objects may compete for the rule each of them represents to be executed; the sequence of applications may be crucial, as in between an anti-rule object may appear and annihilate the rule object. Finally, let us mention that the concepts of activation of rules and of rule objects already

have appeared in some different way in [13] embedded in an even more complex setting.

We also have to emphasize that each copy of a rule object allows for exactly one application of the corresponding rule it represents, i.e., we deal with multisets of rules only applicable if the corresponding multiset of rule objects is present, which restricts the set of applicable sets of multisets of rules in a given derivation mode, especially in the maximally parallel derivation modes. This also means that we have to deal with a subtle technical detail: We either may start with the set of all applicable sets of multisets of rules, no matter which and how many rule objects are present, then take out all the multisets of rules which conform with the given derivation mode, and then only take those for which sufficient resources of rule objects are available. On the other hand, we may also first take only those multisets of rules for which there are sufficient resources of rule objects available, take these as the set of applicable rule multisets and only afterwards apply the condition for the derivation mode. Examples to explain these differences will be given in Section 3.

After explaining some notions and definitions used in this paper in the next section, in Section 3 we will define our new model of P systems with rule and anti-rule objects as well the kind of rules, the derivation modes, and the halting conditions used afterwards; moreover, some examples are given to illustrate the new concept. In Section 4, we then establish our main results. We show that computational completeness can even be obtained with one-membrane P systems using non-cooperative rules and rule / anti-rule annihilation rules as well as one of the standard maximally parallel derivation modes or maximally parallel set derivation modes (i.e., non-extendable (multi)sets of rules, (multi)sets with maximal number of rules, (multi)sets of rules affecting the maximal number of objects). When using the sequential derivation mode, at least the computational power of partially blind register machines is obtained. A summary of the results we obtained as well as an outlook to future research topics conclude the paper.

2 Prerequisites

The set of integers is denoted by $\mathbb{Z}$, and the set of non-negative integers by $\mathbb{N}$. Given an alphabet V, a finite non-empty set of abstract symbols, the free monoid generated by V under the operation of concatenation is denoted by V^*. The elements of V^* are called strings, the empty string is denoted by λ, and $V^* \backslash \{\lambda\}$ is denoted by V^+. For an arbitrary alphabet $V = \{a_1, \ldots, a_n\}$, the number of occurrences of a symbol a_i in a string x is denoted by $|x|_{a_i}$, while the length of a string x is denoted by $|x| = \sum_{a_i \in V} |x|_{a_i}$. The Parikh vector associated with x with respect to $a_1, \ldots, a_n$ is $(|x|_{a_1}, \ldots, |x|_{a_n})$. The Parikh image of an arbitrary language L over $\{a_1, \ldots, a_n\}$ is the set of all Parikh vectors of strings in L, and is denoted by $Ps(L)$. For a family of languages FL, the family of Parikh images of languages

in FL is denoted by $PsFL$, while for families of languages over a one-letter (d-letter) alphabet, the corresponding sets of non-negative integers (d-vectors with non-negative components) are denoted by NFL (N^dFL).

A (finite) multiset over a (finite) alphabet $V = \{a_1, \ldots, a_n\}$, is a mapping $f : V \to \mathbb{N}$ and can be represented by $\langle a_1^{f(a_1)}, \ldots, a_n^{f(a_n)} \rangle$ or by any string x for which $(|x|_{a_1}, \ldots, |x|_{a_n}) = (f(a_1), \ldots, f(a_n))$. In the following we will not distinguish between a vector $(m_1, \ldots, m_n)$, a multiset $\langle a_1^{m_1}, \ldots, a_n^{m_n} \rangle$ or a string x having $(|x|_{a_1}, \ldots, |x|_{a_n}) = (m_1, \ldots, m_n)$. Fixing the sequence of symbols $a_1, \ldots, a_n$ in an alphabet V in advance, the representation of the multiset $\langle a_1^{m_1}, \ldots, a_n^{m_n} \rangle$ by the string $a_1^{m_1} \ldots a_n^{m_n}$ 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 [11] and [27].

Register machines

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

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \backslash \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Increases the value of register j by one, followed by a non-deterministic jump to instruction l_2 or l_3. This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$, with $l_1 \in B \backslash \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  If the value of register j is zero then jump to instruction l_3; otherwise, the value of register j is decreased by one, followed by a jump to instruction l_2. The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$. Stops 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 instruction l_0 of P, and terminate with reaching the HALT-instruction l_h.

For useful results on the computational power of register machines, we refer to [22].

In *partially blind* register machines, the SUB instruction has the form $p : (SUB(r), q)$: if the register r is not empty, it is decremented and the register machine moves to state q, otherwise the machine crashes – the computation stops in a non-halting configuration, yielding no result. Moreover, we require that valid computations of a partially blind register machine have 0 in all non-output registers in halting configurations.

3 P Systems with Rule and Anti-Rule Objects

Formally, a *P system with anti-rules* (a *PARS* for short) is a construct

$$\Pi = \left(V, T, H \cup H^-, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, g_H, f, \Longrightarrow_{\Pi,\delta}\right) \text{ where}$$

- V is the alphabet of *objects*;
- $T \subseteq V$ is the alphabet of *terminal objects*;
- $H \cup H^-$ is the alphabet of *rule objects* (H) and *anti-rule objects* (H^-), respectively; we also define $V_H := V \cup H \cup H^-$;
- μ is the hierarchical membrane structure (a rooted tree of membranes) with the membranes uniquely labeled by the numbers from 1 to m;
- $w_i \in V^*$, $1 \leq i \leq m$, is the *initial multiset* in membrane i;
- R_i, $1 \leq i \leq m$, is a finite set of *rules of type* X assigned to membrane i; we also define $R = \bigcup_{1 \leq i \leq m} \{(i, r) \mid r \in R_i\}$;
- g_H is a function assigning a rule from R to every rule object in H;
- f is the label of the membrane from which the result of a computation has to be taken from (in the generative case) or into which the initial multiset has to be given in addition to w_f (in the accepting case),
- $\Longrightarrow_{\Pi,\delta}$ is the derivation relation under the derivation mode δ.

The symbol X in "rules of type X" may stand for "cooperative", "non-cooperative","purely catalytic", "catalytic", etc., see Subsection 3.1.

A *configuration* is a list of the contents of each membrane region; a sequence of configurations $C_1, \ldots, C_k$ is called a *computation* in the derivation mode δ if $C_i \Longrightarrow_{\Pi,\delta} C_{i+1}$ for $1 \leq i < k$. The derivation relation $\Longrightarrow_{\Pi,\delta}$ is defined by the set of rules in Π and the given derivation mode which determines the multiset of rules to be applied to the multisets contained in each membrane also taking into account the available rule objects, as we will explain in more detail in Subsection 3.4.

3.1 Standard Rule Variants

Non-cooperative rules have the form $a \to w$, where a is a symbol and w is a multiset, catalytic rules have the form $ca \to cw$, where the symbol c is called the *catalyst*, and cooperative rules have no restrictions on the form of the left-hand side. These types of rules will be denoted by *ncoo* (*non-cooperative*), *pcat* (*purely catalytic*), and *coo* (*cooperative*); if both non-cooperative and catalytic rules are allowed, we write *cat* (*catalytic*).

If in general a P system has more than one membrane, each symbol on the right-hand side may have assigned a target where the symbol has to be sent after the application of the rule, where the targets take into account the tree structure of the membranes as follows:

here the symbol stays in the membrane where the rule is applied;
out the symbol is sent to the outer membrane, i.e., the membrane enclosing the membrane where the rule is applied;

in the symbol is sent to an inner membrane, i.e., a membrane enclosed by the membrane where the rule is applied;
in_j the symbol is sent to the inner membrane labeled by j.

3.2 Derivation Modes

In general, the set of all multisets of rules applicable in a P system to a given configuration C is denoted by $Appl(\Pi, C)$ and can be restricted by imposing specific conditions, thus yielding the following basic derivation modes (for example, see [21] for formal definitions):

- asynchronous mode (abbreviated $asyn$): at least one rule is applied;
- sequential mode ($sequ$): only one rule is applied;
- maximally parallel mode (max): a non-extendable multiset of rules is applied;
- maximally parallel mode with maximal number of rules (max_{rules}): a non-extendable multiset of rules of maximal possible cardinality is applied;
- maximally parallel mode with maximal number of objects ($max_{objects}$): a non-extendable multiset of rules affecting as many objects as possible is applied.

In [7], these derivation modes are restricted in such a way that each rule can be applied at most once, thus yielding the set modes $sasyn$, $smax$, $smax_{rules}$, and $smax_{objects}$ (the sequential mode is already a set mode by definition):

- asynchronous set mode (abbreviated $sasyn$): at least one rule is applied, but each rule at most once;
- maximally parallel set mode ($smax$): a non-extendable set of rules is applied;
- maximally parallel set mode with maximal number of rules ($smax_{rules}$): a non-extendable set of rules of maximal possible cardinality is applied;
- maximally parallel set mode with maximal number of objects ($smax_{objects}$): a non-extendable set of rules affecting as many objects as possible is applied.

Let us denote the set of all multisets (possibly only sets) of rules applicable in a (tissue) P system Π to a given configuration C in the derivation mode δ by $Appl(\Pi, C, \delta)$. We immediately observe that $Appl(\Pi, C, asyn) = Appl(\Pi, C)$.
To collect the set and multiset derivation modes, we use the following notations:
$D_S = \{sequ, sasyn, smax, smax_{rules}, smax_{objects}\}$ and
$D_M = \{asyn, max, max_{rules}, max_{objects}\}$.

3.3 Halting Conditions

Besides the standard total halting with no (multi)set of rules being applicable any more to the current configuration, some more variants of halting conditions have been considered in the literature:

total halting (H) the common halting strategy where the computation stops with no (multi)set of rules being applicable any more

unconditional halting (u) the result of a computation can be taken from every configuration derived from the initial one (possibly only taking terminal results)

partial halting (h) the set of rules R is partitioned into disjoint subsets R_1 to R_h, and a computation stops if there is no multiset of rules applicable to the current configuration which contains a rule from every set R_j, $1 \leq j \leq h$

halting with states (s) the configuration with which a derivation may stop must fulfill a recursive condition (which corresponds with a *final state*)

The variant of unconditional halting was introduced in [10]. Partial halting, for example, was investigated in [6, 8, 18], using the membranes for partitioning the rules. Formal definitions for the halting conditions H, h, s can be found in [21].

In the description for P systems, the derivation relation under the derivation mode δ, $\Longrightarrow_{\Pi,\delta}$, is extended by the halting condition, i.e., we then write $\Longrightarrow_{\Pi,\delta,\beta}$ for $\beta \in \{H, h, u, s\}$. By default, β is understood to be the total halting H and then usually omitted.

3.4 Rule and Anti-Rule Objects

In the set of rules R, the rules on the left-hand side may only contain symbols from V, whereas on the right-hand side of a rules any symbol from V_H may appear.

In any computation step of a PARS Π only a multiset R' of rules can be applied such that for each (copy of a) rule r a rule object h with $g(h) = r$ is present in the current configuration. With the application of a rule r, a copy of a rule object h with $g(h) = r$ is consumed, i.e., the number of rule objects in the sense of multisets is important for the applicability of a multiset of rules.

As we allow anti-rule objects to be generated, too, we implicitly assume the rule / anti-rule annihilation rule $hh^- \to \lambda$ to be present in every membrane, for every $h \in H$. Before the next derivation step as described above can be carried out, all such annihilation rules have to be executed, as we assume them to have (weak) priority over all other rules.

As already mentioned in Section 1, each copy of a rule object allows for exactly one application of the corresponding rule it represents, i.e., we deal with multisets of rules only applicable if the corresponding multiset of rule objects is present, which restricts the set of applicable sets of multisets of rules in a given derivation mode, especially in the maximally parallel derivation modes.

Hence, there are two possible ways how to define the applicable multisets of rules applicable to a given configuration; we observe that a configuration may contain elements from V_H, not only V, but as the rule / anti-rule annihilation rules $hh^- \to \lambda$ have weak priority, no pair hh^- can be present any more:

starting with applicable multisets (δ_a) We start with the set of all applicable sets of multisets of rules, no matter which and how many rule objects are present, then take out all the multisets of rules which conform with the given derivation

mode, and then only take those for which sufficient resources of rule objects are available.

starting with available rule objects (δ_r) We first take only those multisets of rules for which there are sufficient resources of rule objects available, take these as the set of applicable rule multisets and only afterwards apply the condition for the derivation mode.

Which variant we choose for a given derivation mode, will be indicated by a subscript a or r to δ, i.e., we write δ_a and δ_r, respectively.

The *language generated by* the PARS Π is the set of all terminal multisets which can be obtained in the output membrane f starting from the initial configuration $C_1 = (w_1, \ldots, w_m)$ using the derivation mode δ_α, $\alpha \in \{a, r\}$, in a halting computation using the halting condition β, i.e.,

$$L_{gen,\delta_\alpha,\beta}(\Pi) = \left\{ C(f) \in T^\circ \mid C_1 \stackrel{*}{\Longrightarrow}_{\Pi,\delta_\alpha,\beta} C \wedge \text{halting}_{\delta_\alpha,\beta}(C) \right\},$$

where $C(f)$ stands for the multiset contained in the output membrane f of the final configuration C and $\text{halting}_{\delta_\alpha,\beta}(C)$ indicates that C is a halting configuration with respect to the halting condition β when using δ_α.

The family of languages of multisets generated by PARSs of type X with at most n membranes in the derivation mode δ_α using the halting condition β is denoted by $Ps_{gen,\delta_\alpha,\beta}OP_n(X)$.

We may also consider PARSs as accepting mechanisms: in membrane f, we add the input multiset w_0 to w_f in the initial configuration $C_1 = (w_1, \ldots, w_m)$ thus obtaining $C_1[w_0] = (w_1, \ldots, w_f w_0, \ldots, w_m)$; the input multiset w_0 is accepted if there exists a halting computation in the derivation mode δ_α starting from $C_1[w_0]$, i.e.,

$$L_{acc,\delta_\alpha,\beta}(\Pi) = \left\{ w_0 \in T^\circ \mid \exists C : \ \left(C_1[w_0] \stackrel{*}{\Longrightarrow}_{\Pi,\delta_\alpha,\beta} C \wedge \text{halting}_{\delta_\alpha,\beta}(C) \right) \right\}.$$

Then the family of languages of multisets accepted by PARSs of type X with at most n membranes in the derivation mode δ_α using the halting condition β is denoted by $Ps_{acc,\delta_\alpha,\beta}OP_n(X)$.

We finally mention that PARSs can also be used to compute functions and relations, with using f both as input and output membrane or even using two different membranes for the input and the output. Yet in this paper we will mainly focus on the generating case.

3.5 Flattening

As many variants of P systems can be *flattened* to only one membrane, see [17], we often may assume the simplest membrane structure of only one membrane which in effect reduces the P system to a multiset processing mechanism, and, observing that $f = 1$, in what follows we then will use the reduced notation

$$\Pi = \left(V, T, H \cup H^-, w, R, g_H, \Longrightarrow_{\Pi,\delta_\alpha,\beta}\right)$$

for a PARS with only one membrane, for which the definitions for the *language generated by* Π and the *language accepted by* Π can be written in an easier way, i.e.,

$$L_{gen,\delta_\alpha,\beta}(\Pi) = \left\{v \in T^\circ \mid w \overset{*}{\Longrightarrow}_{\Pi,\delta_\alpha,\beta} v \wedge \text{halting}_{\delta_\alpha,\beta}(v)\right\} \text{ and}$$
$$L_{acc,\delta_\alpha,\beta}(\Pi) = \left\{w_0 \in T^\circ \mid \exists v : \left(ww_0 \overset{*}{\Longrightarrow}_{\Pi,\delta_\alpha,\beta} v \wedge \text{halting}_{\delta_\alpha,\beta}(v)\right)\right\}.$$

The family of languages of multisets generated by one-membrane PARSs of type X in the derivation mode δ_α using the halting condition β is denoted by $Ps_{gen,\delta_\alpha,\beta}OP(X)$.

The family of languages of multisets accepted by one-membrane PARSs of type X in the derivation mode δ_α using the halting condition β is denoted by $Ps_{acc,\delta_\alpha,\beta}OP(X)$.

The following example illustrates that the two variants δ_a and δ_a need not yield the same results:

Example 1. Take the PARS

$$\Pi = \left(V = \{a\}, T = \{a\}, H \cup H^-, w = ar, R = \{a \to aar\}, g_H, \Longrightarrow_{\Pi,\delta_\alpha,u}\right)$$

with $H = \{r\}$ and $g_H = \{(r, a \to aar)\}$. Then, with every application of the rule $r : a \to aar$ we get one more symbol a, i.e., for $\alpha \in \{a, r\}$, we have

$$\begin{aligned}\{a\}^+ &= L_{gen,sequ_\alpha,u}(\Pi) = L_{gen,smax_\alpha,u}(\Pi)\\ &= L_{gen,max_r,u}(\Pi).\end{aligned}$$

But on the other hand, $L_{gen,max_a,u}(\Pi) = \{a, aa\}$, because after the first application of rule r we obtain the configuration aar, which is a terminal one, as the only applicable multiset of rules which is not extendable would contain two copies of the rule $a \to aar$, yet only one corresponding rule object r is available.

The following example shows that different results are obtained with different derivation modes δ:

Example 2. Take the PARS

$$\Pi = \left(V = \{a\}, T = \{a\}, H \cup H^-, w = ar, R = \{a \to aarr\}, g_H, \Longrightarrow_{\Pi,\delta_\alpha,u}\right)$$

with $H = \{r\}$ and $g_H = \{(r, a \to aarr)\}$. Then, with every application of the rule $r : a \to aarr$ we double the number of symbols a when using a maximally parallel derivation mode, i.e., for $\alpha \in \{a, r\}$, we have

$$L_{gen,\delta_\alpha,u}(\Pi) = \left\{a^{2^n} \mid n \geq 0\right\}$$

for any $\delta \in \{max, max_{rules}, max_{objects}\}$, because now the number of rule objects r grows in the same way as the number of symbols a. But on the other hand, we still get $L_{gen,sequ_\alpha,u}(\Pi) = \{a\}^+$, because in every derivation step the rule $r : a \to aarr$ can only be applied once, although the number of rule objects grows in a similar way as the number of symbols a, i.e., we obtain a sequence of configurations $a^n r^n$, $n \geq 1$, from which we extract the terminal results a^n, $n \geq 1$.

4 Results

As our first result, we will show that with the sequential derivation mode PARSs at least are as powerful as partially blind register machines.

Afterwards, computational completeness will be established for PARSs working in the derivation modes max_r and $smax_r, smax_a$.

4.1 Sequential PARSs

In order to easily comply with the final condition that in halting computations of partially blind register machines all non-output registers should be zero, for the PARSs in the following theorem we use halting with states, where the final states are defined in such a way that the PARS can only halt with yielding a result if only terminal symbols are present any more. Now let $NPBRM$ and $PsPBRM$ denote the families of sets of multisets and vectors of multisets, respectively, generated by partially blind register machines.

Theorem 1. *For any $Y \in \{N, Ps\}$ and $\alpha \in \{a, r\}$,*

$$YPBRM \subseteq Y_{gen,sequ_\alpha,s}OP(ncoo).$$

Proof. Let $M = (m, B, l_0, l_h, P)$ be a partially blind register machine; we assume the output registers to be the first k ones. We now construct a PARS Π which simulates (the computations of) M; the contents of register r is represented by the number of copies of symbols a_r:

$$\Pi = \left(V, T, H \cup H^-, w, R, g_H, \Longrightarrow_{\Pi,sequ_\alpha,s}\right)$$

where

- $V = \{a_j \mid 1 \leq j \leq m\} \cup \{s, \#\}$;
- $T = \{a_j \mid 1 \leq j \leq k\}$;
- $H = B \cup B'_{SUB} \cup \{l_\#\}$, i.e., the *rule objects* are the instruction labels in B, their primed variants, but only for the SUB-instructions, i.e.,

  $$B'_{SUB} = \{l'_1 \in B \mid l_1 : (SUB(r), l_2) \in P\},$$

  and the label $l_\#$ for the trap rule $s \to \#$; we also define $V_H := V \cup H \cup H^-$;

- $w = l_0 s$, i.e., we start with the symbol s and the rule object l_0 representing the initial instruction.

The instructions of M are simulated by the following rules in R; we will write $l : r$ to both indicate the rule r and the label l, having assigned the rule r by the function g_H.

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Simulated by the rules $l_1 : s \to a_r l_2$ and $l_1 : s \to a_r l_3$.

- $l_1 : (SUB(r), l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $1 \leq r \leq m$.
  As labeled rule common for the simulations of all SUB-instructions, we have $l_\# : s \to \#$ and the rule / anti-rule annihilation rule $l_\# {l_\#}^- \to \lambda$.
  The application of the rule $l_\# : s \to \#$ traps the whole computation, as $\#$ is no terminal symbol and thus any configuration containing the trap symbol cannot fulfill the final state condition.
  The *decrement case* for instruction l_1 is simulated by the rules
  $l_1 : s \to s l_1' l_\#$ generating the rule objects l_1' and $l_\#$, and
  $l_1' : a_r \to l_2 {l_\#}^-$, which guarantees a correct simulation of a possible decrement instruction in case register r is not empty (and also eliminates the rule object $l_\#$ by generating its anti-rule object ${l_\#}^-$).
  If register r is empty, then the application of the rule labeled with $l_\#$ generating the trap symbol $\#$ is enforced, which can only be avoided if register r is not empty and the rule labeled with l_1' can be applied instead.

- $l_h : HALT$. Simulated by $l_h : s \to \lambda$.

When the computation in M halts, the state symbol s is removed, and no further rules can be applied provided the simulation of a valid computation in M has been carried out correctly, i.e., if no trap symbols $\#$ are present in this situation, which then guarantees that the halting condition is fulfilled. The terminal symbols in the skin membrane represent the result computed by M. □

In the preceding proof we have taken advantage of the halting condition with final states guaranteeing to take only terminal results. Yet we can also take the standard total halting, but paying the price with having rule objects remaining in the halting computations:

Theorem 2. *For any* $Y \in \{N, Ps\}$ *and* $\alpha \in \{a, r\}$,

$$YPBRM \subseteq Y_{gen,sequ_\alpha,H}OP(ncoo).$$

Proof. As in the preceding proof we simulate a partially blind register machine $M = (m, B, l_0, l_h, P)$, where we assume the output registers to be the first k ones.

We then construct a PARS Π' which simulates (the computations of) M, where Π' is obtained from Π as constructed in the proof of Theorem 1 by extending it in the following way:

$$\Pi' = \left(V, T, H' \cup H'^{-}, w, R', g_{H'}, \Longrightarrow_{\Pi, sequ_\alpha, H}\right)$$

where

- $V = \{a_j \mid 1 \leq j \leq m\} \cup \{s, \#\}$;
- $T = \{a_j \mid 1 \leq j \leq k\}$;
- $H' = B \cup B'_{SUB} \cup \{l_\#, l'_\#\} \cup B_r$, i.e., the *rule objects* are the instruction labels in B, their primed variants, but only for the SUB-instructions, i.e.,

  $$B'_{SUB} = \{l'_1 \in B \mid l_1 : (SUB\,(r)\,, l_2) \in P\},$$

  the label $l_\#$ for the trap rule $s \to \#$, as well as, in addition now, the labels in $B_r = \{l_{j,\#} \mid k+1 \leq j \leq m\}$ needed for the final zero check and the label $l'_\#$, which is needed for the additional trap rule $\# \to \#$; we also define $V_{H'} := V \cup H' \cup H'^{-}$;
- $w = l_0 s$, i.e., we start with the symbol s and the rule object l_0 representing the initial instruction.

For simulating the instructions of M we take all rules in R' constructed as follows:

- $l_1 : (ADD\,(j)\,, l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Simulated by the rules $l_1 : s \to a_r l_2$ and $l_1 : s \to a_r l_3$.

- $l_1 : (SUB\,(r)\,, l_2)$, with $l_1 \in B \setminus \{l_h\}$, $l_2 \in B$, $1 \leq r \leq m$.
  As labeled rule common for the simulations of all SUB-instructions, we have $l_\# : s \to \#$ and the rule / anti-rule annihilation rule $l_\# {l_\#}^{-} \to \lambda$.
  The application of the rule $l_\# : s \to \#$ traps the whole computation by introducing the trap symbol $\#$.
  The *decrement case* for instruction l_1 is simulated by the rules
  $l_1 : s \to s l'_1 l_\#$ generating the rule objects l'_1 and $l_\#$, and
  $l'_1 : a_r \to l_2 {l_\#}^{-}$, which guarantees a correct simulation of a possible decrement instruction in case register r is not empty (and also eliminates the rule object $l_\#$ by generating its anti-rule object ${l_\#}^{-}$).
  If register r is empty, then the application of the rule labeled with $l_\#$ generating the trap symbol $\#$ is enforced, which can only be avoided if register r is not empty and the rule labeled with l'_1 can be applied instead.

- $l_h : HALT$.
  Instead of $l_h : s \to \lambda$ we now use the final rule $l_h : s \to l'_\# l_{k+1,\#} \ldots l_{m,\#}$, where $l_{j,\#} : a_j \to \#$, $k+1 \leq j \leq m$, introduces the trap symbol $\#$, if any of the non-output registers is not empty when l_h is reached. If at some moment the trap symbol is introduced, an infinite (non-halting) derivation finally is guaranteed by the rule $l'_\# : \# \to \# l'_\#$.

When the computation in M halts, the state symbol s is removed, and no further rules can be applied provided the simulation of a valid computation in M has been carried out correctly, i.e., if no trap symbols # are present in this situation, *AND* the final zero test is successful, i.e., none of the rules $l_{j,\#} : a_j \to \#$, $k+1 \leq j \leq m$ is applicable in the step after the application of the final rule $l_h : s \to l'_{\#} l_{k+1,\#} \ldots l_{m,\#}$.

Only the terminal symbols in the skin membrane represent the result computed by M. □

In the proof of Theorem 1 we needed the anti-rule object ${l_{\#}}^-$ and the rule / anti-rule annihilation rule $l_{\#}{l_{\#}}^- \to \lambda$ to finally obtain a clean result without any rule object. Yet as in the construction of the PARS in Theorem 2 we anyway get rule objects remaining in the final configuration we need not use this anti-rule object ${l_{\#}}^-$ and the corresponding rule / anti-rule annihilation rule $l_{\#}{l_{\#}}^- \to \lambda$ for the PARS Π' constructed in the proof of Theorem 2 any more; the remaining technical details are left to the interested reader.

Denoting a PARS not needing anti-rule objects as well as the corresponding rule / anti-rule annihilation rules by PRS and indicating this by writing α^0 instead of α, we immediately may state the following result:

Corollary 1. *For any* $Y \in \{N, Ps\}$ *and* $\alpha \in \{a, r\}$,

$$YPBRM \subseteq Y_{gen,sequ_{\alpha^0},H}OP(ncoo).$$

We remark that the PARSs in Examples 1 and 2 in fact also are only using rule objects and thus are PRSs only.

It remains a challenging open question if we can go beyond $YPBRM$ when using the sequential derivation mode and non-cooperative rules.

4.2 Computational Completeness

We now show that PARSs characterize the families NRE and $PsRE$, respectively. The main proof idea – as used very often in the area of P systems – is to simulate (the computations of) register machines, as carried out in a similar way in [1] for P systems with anti-matter.

Theorem 3. *For any* $Y \in \{N, Ps\}$ *and* $\gamma \in \{gen, acc\}$,

$$Y_{\gamma,\delta_r,H}OP(ncoo) = YRE$$

for any $\delta \in \{max, max_{rules}, max_{objects}\}$.

Proof. Let $M = (m, B, l_0, l_h, P)$ be a register machine. We now construct a PARS Π which simulates (the computations of) M; the contents of register r is represented by the number of copies of symbols a_r:

$$\Pi' = \left(V, T, H \cup H^-, w, R, g_H, \Longrightarrow_{\Pi, sequ_\alpha, H}\right)$$

where

- $V = \{a_j \mid 1 \leq j \leq m\} \cup \{s\}$;
- $T = \{a_j \mid 1 \leq j \leq k\}$;
- $H = B \cup \tilde{B} \cup B_r$, where
  $\tilde{B} = \{l_1', l_1'' \mid l_1 \in B, l_1 : (SUB\,(r)\,, l_2, l_3) \in P\}$ and
  $B_r = \{l_{j,l_1} \mid l_1 \in B, l_1 : (SUB\,(r)\,, l_2, l_3) \in P, \text{ and } 1 \leq r \leq m\}$;
  we also define $V_H := V \cup H \cup H^-$;
- $w = l_0 s$, i.e., we start with the symbol s and the rule object l_0 representing the initial instruction.

The instructions of M are simulated by the following rules in R (we emphasize that by definition, for all rule objects $l \in H$ also the anti-rule objects l^- are part of the PARS Π as well as all the corresponding rule / anti-rule annihilation rules $l\, l^- \to \lambda$ are in R):

- $l_1 : (ADD\,(j)\,, l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Simulated by the rules $l_1 : s \to a_r l_2$ and $l_1 : s \to a_r l_3$.

- $l_1 : (SUB\,(r)\,, l_2, l_3)$, with $l_1 \in B \setminus \{l_h\}$, $l_2, l_3 \in B$, $1 \leq r \leq m$.

  We start the simulation by using the rule
  $l_1 : s \to s l_1' l_{r,l_1}$ and then possibly use the two rules
  $l_1' : s \to s l_1''$ and $l_{r,l_1} : a_r \to l_2 {l_1''}^-$.
  If register r is not empty, then $l_{r,l_1} : a_r \to l_2 {l_1''}^-$ can be applied, at the same time generating the label l_2 to continue the computation after the successful simulation of the *decrement case* and eliminating the rule object l_1'' by generating its anti-rule object ${l_1''}^-$.
  In case the register is empty, the rule object l_1'' is still present in the next derivation step and correctly ends the simulation of the *zero test case* with
  $l_1'' : s \to s {l_{r,l_1}}^- l_3$,
  at the same time eliminating the rule object l_{r,l_1} by generating the corresponding anti-rule object ${l_{r,l_1}}^-$.

- $l_h : HALT$. Simulated by $l_h : s \to \lambda$.

When the computation in M halts, the symbol s is removed, and no further rules can be applied; as the simulation has been carried out correctly, the terminal symbols in the skin membrane at the end of a halting computation represent the result computed by M.

We finally observe the important fact that Π simulates the SUB-instructions of M in a deterministic way, i.e., in the accepting case the simulation of a deterministic register machine is deterministic. □

In the maximally parallel derivation modes δ we can only use the variant δ_r, because in case that more than one object a_r is present, we still want only one a_r to be erased in the decrement case, which is guaranteed by having only one rule object l_{r,l_1}, whereas without the restriction for the presence of rule objects, this rule would be used for every copy of a_r. On the other hand, in any of the set maximally derivation modes δ already the definition of the mode guarantees that this rule can only be applied once; hence, using the same construction as in the proof of Theorem 3, we immediately get the corresponding result for all δ_r *and* δ_a in case of the maximally parallel set derivation modes:

Corollary 2. *For any* $Y \in \{N, Ps\}$, $\alpha \in \{a, r\}$, *and* $\gamma \in \{gen, acc\}$,

$$Y_{\gamma,\delta_\alpha,H}OP(ncoo) = YRE$$

for any $\delta \in \{smax, smax_{rules}, smax_{objects}\}$.

5 Conclusion

In this paper we have taken over the idea of matter and anti-matter objects in P systems to P systems with rule objects and anti-rule objects. For each rule to be applied, a rule object must be present, which is consumed by the application of the rule. Whenever a rule object h meets its anti-rule object h^- the rule / anti-rule annihilation rule $hh^- \to \lambda$, independent from the underlying derivation mode, has to be applied before the next derivation step is executed.

The use of anti-rule objects and the corresponding rule / anti-rule annihilation rules allows for the simulation of register machines with only non-cooperative rules and any of the maximally (set) derivation modes. In the sequential mode, non-cooperative rules together with rule objects, but even without anti-rule objects and the corresponding rule / anti-rule annihilation rules at least allow for the simulation of partially blind register machines. Whether we could obtain more, remains as a challenge for future research.

In that sense, anti-rule objects (and the corresponding rule / anti-rule annihilation rules) may also constitute a frontier of tractability as it was shown for anti-matter, for example, see [12]. Investigating these kinds of complexity issues is also a project for future research.

In this paper, we have only investigated some of the possible combinations of derivation modes and halting conditions. Considering other combinations of derivation modes and halting conditions as well as other kinds of rules, for example, insertion, deletion, and substitution, is a promising topic for the future, too.

Another concept that can be added to the model of P systems with anti-rule objects is to use decaying objects as in [15], but only for the rule objects and the

anti-rule objects. A rule object decaying in t time steps means that it can only activate the application of the rule it stands for during the next t derivation steps, whereafter the rule object vanishes even without the application of the rule / anti-rule annihilation rule.

Acknowledgements

The ideas for this paper came up in the inspiring atmosphere of the Brainstorming Week on Membrane Computing in Sevilla this year.

References

1. Alhazov, A., Aman, B., Freund, R.: P systems with anti-matter. In: Gheorghe, M., Rozenberg, G., Salomaa, A., Sosík, P., Zandron, C. (eds.) Membrane Computing – 15th International Conference, CMC 2014, Prague, Czech Republic, August 20–22, 2014, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8961, pp. 66–85. Springer (2014). https://doi.org/10.1007/978-3-319-14370-5_5
2. Alhazov, A., Aman, B., Freund, R., Păun, Gh.: Matter and anti-matter in membrane systems. In: Macías-Ramos, L.F., Martínez-del-Amor, M.A., Păun, Gh., Riscos-Núñez, A., Valencia-Cabrera, L. (eds.) Proceedings of the Twelfth Brainstorming Week on Membrane Computing. pp. 1–26 (2014), http://www.gcn.us.es/files/12bwmc/001_bwmc2014AntiMatter.pdf
3. Alhazov, A., Freund, R., Ivanov, S.: Introducing the concept of activation and blocking of rules in the general framework for regulated rewriting in sequential grammars. In: Proceedings of BWMC 2018 (2018)
4. Alhazov, A., Freund, R., Ivanov, S.: P systems with activation and blocking of rules. In: Stepney, S., Verlan, S. (eds.) Unconventional Computation and Natural Computation – 17th International Conference, UCNC 2018, Fontainebleau, France, June 25-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10867, pp. 1–15. Springer (2018). https://doi.org/10.1007/978-3-319-92435-9_1
5. Alhazov, A., Freund, R., Ivanov, S.: Sequential grammars with activation and blocking of rules. In: Machines, Computations, and Universality – 8th International Conference, MCU 2018, Fontainebleau, France, June 28-30, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10881, pp. 51–68 (2018). https://doi.org/10.1007/978-3-319-92402-1_3
6. Alhazov, A., Freund, R., Oswald, M., Verlan, S.: Partial halting in P systems using membrane rules with permitting contexts. In: Durand-Lose, J., Margenstern, M. (eds.) Machines, Computations, and Universality. pp. 110–121. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74593-8_10
7. Alhazov, A., Freund, R., Verlan, S.: P systems working in maximal variants of the set derivation mode. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10105, pp. 83–102. Springer (2017). https://doi.org/10.1007/978-3-319-54072-6_6
8. Alhazov, A., Oswald, M., Freund, R., Verlan, S.: Partial halting and minimal parallelism based on arbitrary rule partitions. Fundam. Inform. **91**(1), 17–34 (2009). https://doi.org/10.3233/FI-2009-0031

9. Alhazov, A., Sburlan, D.: Static sorting P systems. In: Ciobanu, G., Pérez-Jiménez, M.J., Păun, Gh. (eds.) Applications of Membrane Computing, pp. 215–252. Natural Computing Series, Springer (2006). https://doi.org/10.1007/3-540-29937-8_8
10. Beyreder, M., Freund, R.: Membrane systems using noncooperative rules with unconditional halting. In: Corne, D.W., Frisco, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. pp. 129–136. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95885-7_10
11. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer (1989), `https://www.springer.com/de/book/9783642749346`
12. Díaz-Pernil, D., Peña-Cantillana, F., Alhazov, A., Freund, R., Gutiérrez-Naranjo, M.A.: Antimatter as a frontier of tractability in membrane computing. Fundam. Inform. **134**(1–2), 83–96 (2014). https://doi.org/10.3233/FI-2014-1092
13. Freund, R.: Generalized P-Systems. In: Ciobanu, G., Păun, Gh. (eds.) Fundamentals of Computation Theory, 12th International Symposium, FCT '99, Iaşi, Romania, August 30 – September 3, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1684, pp. 281–292. Springer (1999)
14. Freund, R.: Purely catalytic P systems: Two catalysts can be sufficient for computational completeness. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu. (eds.) CMC14 Proceedings – The 14th International Conference on Membrane Computing, Chişinău, August 20–23, 2013. pp. 153–166. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova (2013), `http://www.math.md/cmc14/CMC14_Proceedings.pdf`
15. Freund, R.: (Tissue) P systems with decaying objects. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, G. (eds.) Membrane Computing – 13th International Conference, CMC 2012, Budapest, Hungary, August 28–31, 2012, Revised Selected Papers. Lecture Notes in Computer Science, vol. 7762, pp. 1–25. Springer (2013)
16. Freund, R., Kari, L., Oswald, M., Sosík, P.: Computationally universal P systems without priorities: two catalysts are sufficient. Theoretical Computer Science **330**(2), 251–266 (2005). https://doi.org/10.1016/j.tcs.2004.06.029
17. Freund, R., Leporati, A., Mauri, G., Porreca, A.E., Verlan, S., Zandron, C.: Flattening in (tissue) P systems. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 8340, pp. 173–188. Springer (2014). https://doi.org/10.1007/978-3-642-54239-8_13
18. Freund, R., Oswald, M.: Partial halting in P systems. Int. J. Found. Comput. Sci. **18**(6), 1215–1225 (2007). https://doi.org/10.1142/S0129054107005261
19. Freund, R., Oswald, M.: Catalytic and purely catalytic P automata: control mechanisms for obtaining computational completeness. In: Bensch, S., Drewes, F., Freund, R., Otto, F. (eds.) Fifth Workshop on Non-Classical Models for Automata and Applications – NCMA 2013, Umeå, Sweden, August 13 – August 14, 2013, Proceedings. books@ocg.at, vol. 294, pp. 133–150. Österreichische Computer Gesellschaft (2013)
20. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9–11, 2013. EPTCS, vol. 128, pp. 47–61 (2013). https://doi.org/10.4204/EPTCS.128.13
21. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane

Computing, Lecture Notes in Computer Science, vol. 4860, pp. 271–284. Springer (2007). https://doi.org/10.1007/978-3-540-77312-2_17
22. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
23. Pan, L., Păun, Gh.: Spiking neural P systems with anti-matter. International Journal of Computers, Communications & Control **4**(3), 273–282 (2009). https://doi.org/10.15837/ijccc.2009.3.2435, http://univagora.ro/jour/index.php/ijccc/article/download/2435/901
24. Păun, Gh.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
25. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002). https://doi.org/10.1007/978-3-642-56196-2
26. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
27. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer (1997). https://doi.org/10.1007/978-3-642-59136-5
28. The P Systems Website. http://ppage.psystems.eu/

Membrane Systems with Priority, Dissolution, Promoters and Inhibitors and Time Petri Nets

Péter Battyányi, György Vaszil

Department of Computer Science, Faculty of Informatics
University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
{battyanyi.peter,vaszil.gyorgy}@inf.unideb.hu

Summary. We continue the investigations on exploring the connection between membrane systems and time Petri nets already commenced in [4] by extending membrane systems with promoters/inhibitors, membrane dissolution and priority for rules compared to the simple symbol-object membrane system. By constructing the simulating Petri net, we retain one of the main characteristics of the Petri net model, namely, the firings of the transitions can take place in any order: we do not impose any additional stipulation on the transition sequences in order to obtain a Petri net model equivalent to the general Turing machine. Instead, we substantially exploit the gain in computational strength obtained by the introduction of the timing feature for Petri nets.

1 Introduction

Several models have emerged in the past decades to model distributed systems with interactive, parallel components. One of them was developed by C. A. Petri [20], and since then the Petri nets have become the underlying system of a vast field of research with a considerable practical interest on the other hand. The theory of membrane systems was invented by Gh. Păun [18], and it has proved to be a very convenient and many-sided model of distributed systems with concurrent processes.

Place/transition Petri nets are bipartite graphs, the conditions of the events of a distributed system are represented by places and directed arcs connect the places to the transitions, that model the events. The conditions for the events are expressed by tokens, an event can take place, i.e., a transition can fire, if there are enough number of tokens an the places at the ends of the incoming arcs of a transition. these places are called as preconditions. The outgoing edges of a transition represent the postcondition of the events. Firing of a transition means removing tokens from the preconditions and adding them to the postconditions. The number of tokens moved in this way are prescribed by the multiplicities of the incoming and outgoing arcs, respectively.

In some cases the original place/transition model has turned out not to be satisfactory, for example, we are not able to model systems where a certain order of events must be taken into account. Various extensions of the Petri net model have appeared, in this paper we deal with the time Petri net model developed by Merlin [17]. In this model, time intervals are associated to transitions. The local time observed from a transition can be modified by the Petri net state transition rules, and a transition can fire only if its observed time lies in the interval assigned to the transition by the construction of the model. In this way, the computational power of Petri nets is increased: the time Petri net model is Turing complete in contrast with the original state/transition Petri net.

Membrane systems are models of distributed, synchronized computational systems ordered in a tree-like structure. The building blocks are compartments, which contain multisets of objects. The multisets evolve in each compartment in a parallel manner, and the compartments, in each computational step, wait for the others to finish their computation, hence the system acts in a synchronized manner. In every computational step, the multisets in the compartments evolve in a maximal parallel manner, this means that, in each step, as many rules of the compartment are applied simultaneously as possible.

In this paper, we continue the research on the connection between time Petri nets and membrane systems initiated in [4]. We extend the basic construction of the time Petri net simulating a symbol object membrane system developed in [4] in order to represent some more membrane computational tools like promoters/inhibitors, membrane dissolution and priority of rules. One of the main features of our construction is that the unsynchronized characteristics of Petri nets is retained when a Petri net equivalent of a membrane system is presented. That is, unlike the construction in [13], we do not stipulate that the Petri nets should perform their computational steps in a maximal parallel manner, the attached time intervals provide the synchronization in the corresponding Petri nets.

2 Membrane Systems

Membrane systems are computational models operating on multisets. A finite multiset over an alphabet O is a mapping $M : O \to \mathbb{N}$, where $\mathbb{N}$ is the set of non-negative integers. The number $M(a)$ for $a \in O$ is called the multiplicity of a in M. We write that $M_1 \subseteq M_2$ if for all $a \in O$, $M_1(a) \leq M_2(a)$. The union or sum of two multisets over O is defined as $(M_1 + M_2)(a) = M_1(a) + M_2(a)$, while the difference is defined for $M_2 \subseteq M_1$ as $(M_1 - M_2)(a) = M_1(a) - M_2(a)$ for all $a \in O$. The set of all finite multisets over an alphabet O is denoted by $\mathcal{M}(O)$; the empty multiset is denoted by $\emptyset$.

The notation $\mathbb{N}_{>0}$ stands for the set of positive integers, while $\mathbb{Q}$ and $\mathbb{Q}_{\geq 0}$ denotes the set of rational numbers and non-negative rational numbers and $\mathbb{R}$ and $\mathbb{R}_{\geq 0}$ the set of real numbers and non-negative real numbers, respectively.

We define the notion of the basic symbol-object membrane system [19] together with the additional features discussed in Chapter 5. A membrane system (or P sys-

tem) is a tree-like structure of hierarchically arranged membranes. The outermost membrane is usually called the *skin* membrane. The membranes are labelled by natural numbers $\{1, \ldots, n\}$, and we use the notation m_i for a membrane labelled by i. Each membrane, except for the *skin* membrane, has its parent membrane. We use μ for representing the structure of the membrane system itself, in fact, the structure itself can be given as a balanced string of left and right brackets indexed by their labels. For example, $\mu = [_{skin}\, [_1\, [_2]_2\, [_3]_3\, [_4]_4\,]_{skin}$. Here, the skin membrane has two submembranes, while region 1 also contains two embedded regions. Abusing the notation, $\mu(i) = k$ can also mean that the parent of the i-th region is region k.

The regions contain multisets over a finite alphabet O. The contents of the regions of a P system evolve through rules associated with the regions. The rules constitute the micro steps of the computations. They are applied in a maximal parallel manner: the computation in a region proceeds until no more rule can further be applied. A computational step is the macro step of the process: it ends when each of the regions have finished their computations. A computational sequence is a sequence of computational steps.

Here we make the usual conditions on the presentation of a computational step: we assume that the computational steps in the regions consist of two phases- first the rule application part produces from the objects on the left-hand sides of the rules the labelled objects on the right-hand sides (the labels of the labelled objects describe the way they should be moved between the regions: stay where they are, move to the parent region, or move into one of the children regions); then we have the communication phase when the labels are removed and all the objects find their regions indicated by their labels.

The P system gives a result when it halts, i.e., when no more rules can be applied in any of the regions. The result is a number or a tuple of natural numbers counting certain objects in the membrane designated as as the output membrane. More formally,

Definition 1. *A P system of degree $n \geq 1$ is $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n)$ where O is an alphabet of objects, μ is a membrane structure of n membranes, $w_i \in \mathcal{M}(O)$ with $1 \leq i \leq n$ are the initial contents of the n regions, R_i with $(1 \leq i \leq n)$ are the sets of evolution rules associated with the regions; they are of the form $u \to v$, where $u \in \mathcal{M}(O)$ and $v \in \mathcal{M}(O \times tar)$, and $tar = \{here, out\} \cup \{in_j \mid 1 \leq j \leq n\}$.*

Unless stated otherwise, we consider the n-th membrane as the output membrane. A configuration is the sequence $W = (w_1, \ldots, w_n)$, where w_k is the multiset contained by membrane m_k $(1 \leq k \leq n)$. By the application of a rule $u \to v \in R_i$ we mean the process of removing the elements of u from the multiset w_i and extending w_i with the labelled elements, which are called messages. As a result, during a computational step, a region can contain both elements of O and messages. An intermediate configuration is an n-tuple of multisets over $O \cup (O \times tar)$. We say that W is a proper configuration, if $w_i \in \mathcal{M}(O)$ for each of its regions w_i.

The communication phase means that the elements coming from the right-hand sides of the rules of region i should be added to the regions as specified by the target indicators associated with them. If $rhs(r)$ contains a pair $(a, here) \in O \times tar$, then $a \in O$ is added to region i, the region where the rule is applied. If $rhs(r)$ contains $(a, out) \in O \times tar$, then a is added to the parent region of region i. If $rhs(r)$ contains $(a, in_j) \in O \times tar$, then a is added to the contents of region j. In the latter case, $\mu(m_j) = m_i$ holds.

Given a (proper) configuration W, we obtain a new (proper) configuration W' by executing the two phases of the transformations determined by the maximal parallel sets of rules chosen for each compartment of the membrane system. We call this a computational step, and denote it by $W \Rightarrow W'$.

In the general case, the symbol-object membrane systems can already generate the recursively enumerable sets of natural numbers. We might consider additional features being present in the membrane system, though, by the previous remark they do not enhance the computational power of the P system. First of all, we can add promoters and inhibitors to the rules that regulate the rule applications in a way that the promoter assigned to the rule $r \in R_i$, when $1 \leq i \leq n$ is fixed, prescribes how many copies of object a ($a \in O$) should be present in m_i for the rule to be applied, while the inhibitor prevents the rule from being applied if the number of a certain object a exceeds the number determined by the inhibitor. Second, we deal with the so-called membrane dissolution. The set of objects is extended with an additional element δ that can appear on the right hand sides of the rules. If $\delta \in rhs(r)$, where r is a rule of the i-th region for some i, and rule r is chosen to be in the maximal multiset of rules in R_i applied in that computational step, then the communication phase is executed as before and, as the result of the presence of δ in m_i, the region m_i together with its set of rules R_i disappears from the P system. This means, it is invisible to the subsequent computational steps. The elements of m_i, except for δ, which disappears, are passed over to the region containing m_i and the rules in R_i can never be applied anymore. The region *skin* cannot dissolve. Finally, we consider a priority relation among rules. That is, we consider a two-place relation ρ_i on the set R_i for each $1 \leq i \leq n$. Let $r', r \in R_i$ for some fixed $i \in \{1, \ldots, n\}$. We say that r' has priority over r, or r' has higher priority than r, if $(r', r) \in \rho_i$. In this case, if both r' and r were applicable in a maximal parallel step, then r is suppressed, that is, not allowed to be applied. The exact definitions can be found in the relevant parts of Section 5. In Section 5 we show that all these features can be smoothly modelled by time Petri nets. The advantage of using time Petri nets instead of the Petri net models mostly applied in the literature (see e.g. [13]) is the fact that the usual order of the firings of the transitions is preserved- we do not inflict any additional firing condition on the transitions of the Petri nets, like the requirement that the transitions fired in a computational step should constitute a maximal multiset of fireable transitions.

3 Time Petri Nets

In this section, following the definitions in [21] we define time Petri nets– a model rendering time intervals to transitions along the concept of Merlin [17]. First of all we define the underlying place/transition Petri nets, and then extend this model to the timed version.

Definition 2. *A Petri net is a tuple* $U = (P, T, F, V, m_0)$ *such that*

1. P, T, F *are finite, where* $P \cap T = \emptyset$, $P \cup T \neq \emptyset$ *and* $F \subseteq (P \times T) \cup (T \times P)$,
2. $V : F \to \mathbb{N}_{>0}$,
3. $m_0 : P \to \mathbb{N}$.

The elements of P *and* T *are called places and transitions, respectively. The elements of* T *are the arcs, and* F *is the flow relation of* U. *The function* V *is the multiplicity (weight) of the arcs, and* m_0 *is the initial marking. In general, a marking is a function* $m : P \to \mathbb{N}$. *We may occasionally omit the initial marking and simply refer to a Petri net as the tuple* $U = (P, T, F, V)$. *We stipulate that for every transition* $t \in T$, *there is a place* $p \in P$ *such that* $f = (p, t) \in F$ *and* $V(f) \neq 0$.

Let $x \in P \cup T$. The pre- and post-sets of x, denoted by ${}^\bullet x$ and $x^\bullet$ respectively, are defined as ${}^\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$.

Definition 3. *For each transition* $t \in T$, *we define two markings,* $t^-, t^+ : P \to \mathbb{N}$ *as follows:*

$$t^-(p) = \begin{cases} V(p,t), & \text{if } (p,t) \in F, \\ 0 & \text{otherwise}, \end{cases} \quad t^+(p) = \begin{cases} V(t,p), & \text{if } (t,p) \in F, \\ 0 & \text{otherwise}. \end{cases}$$

A transition $t \in T$ *is said to be enabled if* $t^-(p) \leq m(p)$ *for all* $p \in {}^\bullet t$.

Applying the notation $\triangle\, t(p) = t^+(p) - t^-(p)$ for $p \in P$, we are able to define the firing of a Petri net.

Definition 4. *Let* $U = (P, T, F, V, m_0)$ *be a Petri net, and* m *be a marking in* U. *A transition* $t \in T$ *can fire in* m *(notation:* $m \longrightarrow^t$ *) if* t *is enabled in* m. *After the firing of* t, *the Petri net obtains the new marking* $m' : P \to \mathbb{N}$ *with* $m'(p) = m(p) + \triangle\, t(p)$ *for all* $p \in P$. *Notation:* $m \longrightarrow^t m'$.

We arrive at time Petri nets if we add time assigned to transitions of the Petri net. Intuitively, the time associated with a transition denote the last time when the transition was fired. We are considering only bounded time intervals.

Definition 5. *A time Petri net is a 6-tuple* $N = (P, T, F, V, m_0, I)$ *such that*

1. *the skeleton of* N *given by* $S(N) = (P, T, F, V, m_0)$ *is a Petri net, and*
2. $I : T \to \mathbb{Q} \times \mathbb{Q}$ *is a function assigning a rational interval to each transition, that is, for each* $t \in T$ *and* $I(t) = (I(t)_1, I(t)_2)$ *we have that* $0 \leq I(t)_1 \leq I(t)_2$.

We call $I(t)_1$ and $I(t)_2$ the earliest and the latest firing times belonging to t, and denote them by $eft(t)$ and $lft(t)$, respectively.

Given a time Petri nets $N = (P, T, F, V, m_0, I)$, a function $m : P \to \mathbb{N}$ is called a p-marking of N. Note that talking about a p-marking of N is the same as talking about a marking of $S(N)$.

Definition 6. *Let $N = (P, T, F, V, m_o, I)$ be a time Petri net, $m : P \to \mathbb{N}$ a p-marking in N, and h be a function called a transition marking (or t-marking) in N, $h : T \to \mathbb{R}_{\geq 0} \cup \{\#\}$. A state in N is a pair $u = (m, h)$ such that the two markings m and h satisfy the following properties: for all $t \in T$,*

1. *if t is not enabled in m (that is, if $t^-(p) > m(p)$ for some $p \in {}^\bullet t$), then $h(t) = \#$,*
2. *if t is enabled in m (that is, if $t^-(p) \leq m(p)$ for all $p \in {}^\bullet t$), then $h(t) \in \mathbb{R}$ with $h(t) \leq lft(t)$).*

The initial state is the pair $u_0 = (m_0, h_0)$, where m_0 is the initial marking and for all $t \in T$,

$$h_0(t) = \begin{cases} 0, & \text{if } t^-(p) \leq m_0(p) \text{ for all } p \in {}^\bullet t, \\ \#, & \text{otherwise .} \end{cases}$$

Definition 7. *A transition $t \in T$ is ready to fire in state $u = (m, h)$ (denoted by $u \longrightarrow^t$) if t is enabled and $eft(t) \leq h(t)$.*

We define the result of the firing for a transition that is ready to fire.

Definition 8. *Let $t \in T$ be a transition and $u = (m, h)$ be a state such that $u \longrightarrow^t$. Then the state u' resulting after the firing of t denoted by $u \longrightarrow^t u'$ is a new state $u' = (m', h')$, such that $m'(p) = m(p) + \triangle t(p)$ for all $p \in P$. Now, for all transitions $s \in T$, we have*

$$h'(s) = \begin{cases} h(s), \text{ if } s^-(p) \leq m(p),\ s^-(p) \leq m'(p) \text{ for all } p \in {}^\bullet s, \\ 0, \quad \text{if } s^-(p) > m(p) \text{ for some } p \in {}^\bullet s, \text{ but} \\ \qquad s^-(p) \leq m'(p) \text{ for all } p \in {}^\bullet s, \\ \#, \quad \text{if } s^-(p) > m'(p) \text{ for some } p \in {}^\bullet s. \end{cases}$$

Hence, the firing of a transition changes not only the p-marking of the Petri net, but also the time values corresponding to the transitions. If a transition $s \in T$ which was enabled before the firing of t remains enabled after the firing, then the value $h(s)$ remains the same, even if s is t itself. If an $s \in T$ is newly enabled with the firing of transition t, then we set $h(s) = 0$. Finally, if s is not enabled after firing of transition t, then $h(s) = \#$.

Observe that we ensure that a rule can be chosen more than once in a maximal parallel step that we allow transitions to be fired several times in a row: if t is fired resulting in the new p-marking m' and $t^-(p) \leq m'(p)$ holds for all $p \in {}^\bullet t$, then $h(t)$ remains the same.

Besides the firing of a transition there is another possibility for a state to alter, and this is the time delay step.

Definition 9. *Let $u = (m, h)$ be a state of a time Petri net, and $\tau \in \mathbb{R}_{\geq 0}$. Then, the elapsing of time with τ is possible for the state u (denoted $u \longrightarrow^{\tau}$) if for all $t \in T$, $h(t) \neq \#$ we have $h(t) + \tau \leq lft(t)$. Then the state u', namely the result of the elapsing of time by τ denoted by $u \longrightarrow^{\tau} u'$ is defined as $u' = (m', h')$, where $m = m'$ and*

$$h'(t) = \begin{cases} h(t) + \tau, & \text{if } h(t) \neq \#, \\ \# & \text{otherwise.} \end{cases}$$

Note that Definition 9 ensures that we are not able to skip a transition when it is enabled: a transition cannot be disabled by a time jump. This kind of semantics is called the strong semantics in the literature [22].

We remark that classic Petri nets can be obviously obtained by having $h(t) = [0, 0]$ for every transition, and no time delay step is ever made.

4 Connecting Petri nets and membrane systems

First of all, we introduce the time Petri net model constructed in [4], which serves as our starting point in the constructions below. Our model relies on the correspondence between Petri nets and membrane systems described in [13], with the additional property that we do not require that our Petri net model should operate in a maximal parallel manner. In general, both by membrane systems and by Petri nets, a computational step can be considered as a multiset of rules or as a multiset of transitions, respectively. In the case of Petri nets, an application of a multiset of transitions is maximal parallel, if augmenting the multiset by any other transition results in a multiset of transitions that cannot be fired simultaneously in that configuration. In the case of membrane systems, maximal parallel execution means that, if we consider any membrane m_k, no rule of m_k can be added to our multiset of rules such that the remaining multiset still forms a multiset of executable rules in m_k. In our construction, the fireable transitions of the simulating Petri nets can be executed in any order, we do not impose a restriction on the computational sequence of the Petri nets. This involves that we have made an essential use of the time feature, since the original place/transition Petri net model is not Turing complete, unlike the majority of the symbol object membrane systems.

We remark that, similarly to membrane systems, Petri nets can also be considered as computational devices, which means that, if we start from an initial configuration such that an input is represented by the tokens contained by some designated places, then, when the computation halts, the content of the output places provide the result of the computation. Depending on the construction, the result can either be a number, or a tuple. The following statement is a reformulation of Theorem 4.2 in [4]. We present the proof with details here, since the subsequent Petri nets in the next section build upon this construction.

Theorem 1. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n)$ be a membrane system. Then there is a time Petri net $N = (P, T, F, V, m_0, I)$ such that N halts if and only if Π halts and, if either of them halts, then both systems provide the same result.*

Proof. The proof is a reinterpretation of that of Theorem 1 in [4]. We elaborate the construction again in order to keep our presentation self-contained. Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n)$ be a membrane system and let $N = (P, T, F, V, m_0, I)$ be the corresponding Petri net. We define N so that a computational step of Π is simulated by two subnets of N. The two subnets correspond to the two computational phases of a computational step of a membrane system, namely, the rule application and the communication phases. In our Petri net model, the tokens in the places $P_0 = O \times \{1, \ldots, n\}$ stand for the objects in the various compartments, while the tokens of the places $\bar{P}_0 = \bar{O} \times \{1, \ldots, n\}$, where $\bar{O} = \{\bar{a} \mid a \in O\}$, represent the messages obtained in the course of the rule applications. Let us see the construction in detail.

- $P = P_0 \cup \bar{P}_0 \cup \{init_{app}, init_{com}, sem, enabld\}$, where $P_0 = O \times \{1, \ldots, n\}$ and $\bar{P}_0 = \bar{O} \times \{1, \ldots, n\}$. Let $m_0(p) = w_j(a)$ for every place $p = (a, j) \in P_0$.

Intuitively, the relation $|p| = t$, where $p = (a, i) \in O \times \{1, \ldots, n\}$, means that there are as many as t objects $a \in O$ in compartment m_i. In other words, $w_i(a) = t$. On the other hand, the equality $|\bar{p}| = s$, where $\bar{p} = (\bar{b}, j) \in \bar{O} \times \{1, \ldots, n\}$, expresses the fact that there are s copies of object b that will enter into membrane m_j at the end of the computational step. The places $init_{com}, init_{app}, sem, enabld$ are places enabling the synchronization of the Petri net model.

- $T = T_0 \cup T_0^* \cup T^{\#} \cup \{t_{app}, t_{com}, t^1_{sem}, t^2_{sem}\}$,

where the transitions are defined as follows.
Let $r_{i,j} \in R_i$, where $1 \leq j \leq |R_i|$ and $1 \leq i \leq n$, be a rule in m_i. Then the transition $t_{i,j} \in T_0$ corresponds to $r_{i,j} \in R_i$. Let us define the arcs associated with $t_{i,j}$ for a fixed i and j together with their multiplicities.

- Assume $t_{i,j} \in T_0$, where $1 \leq i \leq n$ and $1 \leq j \leq |R_i|$. Then $p = (a, i) \in {}^\bullet t_{i,j}$ if and only if $a \in lhs(r_{i,j})$, and $\bar{p} = (\bar{b}, k) \in t^\bullet_{i,j}$ if and only if either $(b, in_k) \in rhs(r_{i,j})$, that is, m_i is the parent region of m_k, or $(b, out) \in rhs(r_{i,j})$, where region k is the parent region of i, or $k = j$ and $(b, here) \in rhs(r_{i,j})$.
  In addition, $enabld \in {}^\bullet t_{i,j} \cap t^\bullet_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq |R_i|)$.
  Regarding the weights of the arcs, let $p = (a, i)$ and $f = (p, t_{i,j}) \in F$. Then the weight of f is the multiplicity of $a \in O$ on the left-hand side of $r_{i,j}$, namely, $V(f) = lhs(r_{i,j})(a)$; furthermore, for $\bar{p} = (\bar{b}, k)$ and $f = (t_{i,j}, \bar{p}) \in F$, the weight of f is $V(f) = rhs(r_{i,j})(b, in_k)$ if region k is a child region of i, $V(f) = rhs(r_{i,j})(b, out)$ if region k is the parent region of i, or $V(f) = rhs(r_{i,j})(b, here)$ for $k = j$. Additionally, if $f = (t_{i,j}, enabld)$, then $V(f) = 1$ $(1 \leq i \leq n, 1 \leq j \leq |R_i|)$.

The transitions in $T^{\#}$ are in charge with the correct simulation of a maximal parallel step: they fire only if there are any enabled rules in any of the regions. Their inputs are the same as those of the elements of T_0, only their outputs differ, since they should not give rise to a change in the original distribution of the tokens before the computational step takes place.

- Let $r_{i,j} \in R_i$, where $1 \leq j \leq |R_i|,\ 1 \leq i \leq n$; then $t^{\#}_{i,j} \in T^{\#}$ is the transition checking the applicability of $r_{i,j}$. Let $p = (a,i) \in P_0$ and let $t^{\#}_{i,j} \in T^{\#}$; then $p \in {}^{\bullet}(t^{\#}_{i,j})$ and $init_{app} \in {}^{\bullet}t^{\#}_{i,j}$ and $enabld \in (t^{\#}_{i,j})^{\bullet}$ and $p \in (t^{\#}_{i,j})^{\bullet}$. In words, for a fixed i and j, $t^{\#}_{i,j}$ expects as many tokens from its outgoing places as the number of distinct objects that is necessitated by an execution of the rule $r_{i,j}$. In the meantime, a token arrives in *enabld*, which ensures the continuation of the simulation of the rule application phase. Then $t^{\#}_{i,j}$ gives back the tokens to the places in P_0.
  As regards the multiplicities, if $f = (init_{app}, t^{\#}_{i,j})$ then $V(f) = 1$, and if $f = (t^{\#}_{i,j}, enabld)$ then $V(f) = 1$; furthermore, if $f = (p, t^{\#}_{i,j})$ or $f = (t^{\#}_{i,j}, p)$ where $p = (a,i)$, then $V(f) = lhs(r_{i,j})(a)$.

The transitions in T^*_0 ensure that tokens can flow back from $\bar{P}_0$ to P_0, thus representing the communication phase of the membrane computation.

- $T^*_0 = \{s_{a,i} \mid a \in O,\ 1 \leq i \leq n\}$. Let $\bar{p} = (\bar{a}, i) \in \bar{P}_0$, then $\bar{p} \in {}^{\bullet}s_{a,i}$ and $init_{com} \in {}^{\bullet}s_{a,i}$. Moreover, if $p = (a,i)$, then $p \in s^{\bullet}_{a,i}$ and $init_{com} \in s^{\bullet}_{a,i}$. Regarding the multiplicities, each arc has multiplicity 1.

The intervals belonging to the elements of $T = T_0 \cup T^*_0 \cup T^{\#}$ are $[0,0]$. The rest of the transitions are defined as follows.

- t_{app} connects *enabld*, and hence the rule application part of the Petri net with the semaphore: $enabld \in {}^{\bullet}t_{app}$ and $sem \in t^{\bullet}_{app}$. Moreover, $V(enabld, t_{app}) = 1$ and $V(t_{app}, sem) = 2$ and $I(t_{app}) = [1,1]$.

The role of t_{app} is to guarantee that a sequence of firings of transitions correctly simulates a maximal parallel application of membrane rules: every transition, $t_{i,j}$ $(1 \leq j \leq |R_i|,\ 1 \leq i \leq n)$, corresponding to a rule execution can fire only if a token is found in *enabld*. On the other hand, if no transition $t_{i,j}$ can fire, then the transition t_{app} connected only to *enabld* will be activated after a time unit's delay.

- t_{com} connects $init_{com}$, and hence the communication part of the Petri net with the semaphore: $init_{com} \in {}^{\bullet}t_{com}$ and $sem \in t^{\bullet}_{com}$. Moreover, $V(init_{com}, t_{com}) = V(t_{com}, sem) = 1$ and $I(t_{com}) = [1,1]$.
  The role of the semaphore is to make sure that the simulation of the rule application and the communication phases takes place in an alternating order. This is achieved by the following machinery.
- Let t^1_{sem} and t^2_{sem} be transitions of the semaphore. Then $sem \in {}^{\bullet}(t^1_{sem})$ and $sem \in {}^{\bullet}(t^2_{sem})$; furthermore, $init_{app} \in (t^1_{sem})^{\bullet}$ and $init_{com} \in (t^1_{sem})^{\bullet}$. If $f = (sem, t^2_{sem})$, then $V(f) = 2$. The weights of the other arcs are 1. In addition, $I(t^1_{sem}) = [1,1]$ and $I(t^2_{sem}) = [0,0]$.

To sum up the above construction: a computational step of a membrane system is split into a rule application and a communication phase, and those two phases are simulated separately and in an alternating order. The simulation of a phase finishes when no more rule applications are possible, hence we ensure that a maximal

parallel step is correctly simulated. When the rule application phase finishes its operation, 2 tokens are sent to the semaphore via t_{app}, and the simulation of the communication phase can immediately begin by forwarding the 2 tokens to $init_{com}$. Otherwise, when the communication phase finishes its operation, only 1 token is sent to the semaphore, so the rule application phase is initiated after a time unit's wait. The structure of the various subnets are described in Figures 1, 2 and 3, respectively.

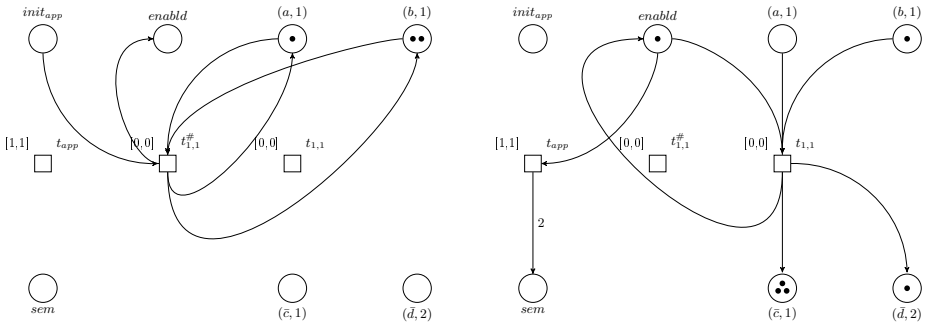


Fig. 1. Assume $a, b^2 \in w(1)$ and $r_{1,1} = ab \to c^3(d, in_2)$, where m_2 is child of m_1. The figure shows the result of a single application of the rule in a split table: to the left is the subnet testing the applicability of $r_{1,1}$ and to the right is the application of the rule itself. The rule consumes an a and a b in region 1 and three tokens are sent to the place $(\bar{c}, 1)$, and one token to $(\bar{d}, 2)$, in accordance with the fact that three objects of c should be added to region 1, and one copy of d should be added to region 2 in the communication phase.

By this, we have simulated a membrane system with a time Petri net such that in the Petri net model no restriction on the transitions is made: the transitions that are ready to fire can be fired in any order. □

5 Extending the correspondence to membrane systems with more features

In this section we examine the possibility of extending our core model to Petri nets that are able to represent various properties of membrane systems, such as the presence of promoters/inhibitors, membrane dissolution and priority among rules. The obtained Petri nets each build upon the basic model defined in the previous section, so, in most of the cases, we restrict ourselves to emphasize only the new elements of the constructions by which the basic Petri net model is extended. First of all, we begin with discussing the case of promoters and inhibitors in the membrane system. Below we present the necessary definitions.

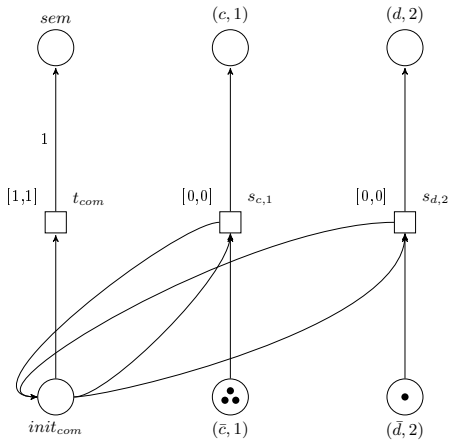


Fig. 2. The Petri net simulating the communication phase of a membrane computational step. When the simulation of a maximal parallel rule application step is finished, a token is given to the semaphore *sem*. The transitions $s_{c,1}, s_{d,2} \in T_0^*$ ensure the correct placement of the tokens corresponding to the messages.

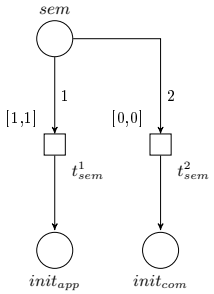


Fig. 3. The semaphore for the Petri net. When the simulation of the rule application phase of a computational step of the membrane system is complete, two tokens appear at *sem*, and then sent to $init_{com}$, activating the simulation of the communication phase of the computational step. When the simulation of the communication phase is completed, one token appears at *sem*, which is then sent to $init_0$, activating the simulation of the rule application phase of a subsequent computational step.

Definition 10. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \mathcal{P})$ be a membrane system with promoters/inhibitors, where $\mathcal{P}(r_{j,i}) \subseteq O^* \times O^*$, for every $r_{j,i} \in R_i$. Then $\mathcal{P}(r_{j,i}) = (\rho_{j,i}, \tau_{j,i}) \subseteq O^* \times O^*$ is a promoter/inhibitor pair for r_j. We denote the pair $(\rho_{j,i}, \tau_{j,i})$ by $(prom^r, inhib^r)$. Let $\mathcal{R}$ be a multiset of rules. A multiset $\mathcal{R}$ is applicable, if each of the following conditions fulfill.*

1. $lhs(r_{j,i})(a) \cdot \mathcal{R}(j,i) \leq w_i(a)$ $(a \in O)$,

2. $prom^r(a) \leq w_i(a)$ $(r \in \mathcal{R}, a \in O)$,
3. $w_i(a) < inhib^r(a)$ $(r \in \mathcal{R}, a \in O)$.

In what follows, we give the structure of the Petri net simulating a general example of a membrane system with promoters/inhibitors.

Theorem 2. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \mathcal{P})$ be a membrane system with promoters/inhibitors. Then there is a time Petri net $N = (P, T, F, V, m_0, I)$ such that N halts if and only if Π halts and, if any of them halts, then both systems provide the same result.*

Proof. Let Π be as in the statement of the theorem. We construct N in a way analogous to the construction of the Petri net of Theorem 1. The Petri net simulates the rule application and the communication phase separately, we only concentrate on the rule application part, since the other parts of the construction are identical to that of the proof of Theorem 1. Let us detail the proof a bit more.

- $P = P_0 \cup \bar{P}_0 \cup \{init_{app}, init_{com}, sem, enabld, contd\}$, where $P_0 = O \times \{1, \ldots, n\}$ and $\bar{P}_0 = \bar{O} \times \{1, \ldots, n\}$. Let $m_0(p) = w_j(a)$ for every place $p = (a, j) \in P_0$.

As before, if $|p| = t$, where $p = (a, i) \in O \times \{1, \ldots, n\}$, then there are as many as t objects $a \in O$ in compartment m_i. Likewise, we retain the meaning of $\bar{p} = (\bar{b}, j) \in \bar{O} \times \{1, \ldots, n\}$, where $|\bar{p}| = s$ expresses the fact that there are s copies of object b that are going to appear in membrane m_j at the end of the computational step. The places $init_{com}, init_{app}, sem, enabld, contd$ are places enabling the synchronization of the Petri net model. The new element here is the place *contd*, which is introduced in order to handle conditions 2 and 3 for rule applicability in Definition 10.

- $T = T_0 \cup T_0^* \cup T^{\#} \cup T^{\#\#} \cup \{t_{app}, t_{com}, t^1_{sem}, t^2_{sem}\}$,

where the transitions are defined as follows.

- The definitions of the transitions T_0 and T_0^* are unchanged. The construction of the arcs and their weights, with respect to T_0 and T_0^*, is exactly the same as above.

The difference lies in the definitions of $T^{\#}$ and $T^{\#\#}$. They ensure that the conditions of rule applications presented in Definition 10 are simulated correctly.

- Let $r_{i,j} \in R_i$, where $1 \leq j \leq |R_i|$, $1 \leq i \leq n$; then $t^{\#}_{i,j} \in T^{\#}$ is the transition checking conditions 1 and 2 in Definition 10. Let $p = (a, i) \in P_0$ and let $t^{\#}_{i,j} \in T^{\#}$; then $p \in {}^\bullet(t^{\#}_{i,j})$ and $init_{app} \in {}^\bullet t^{\#}_{i,j}$ and $contd \in (t^{\#}_{i,j})^\bullet$ and $p \in (t^{\#}_{i,j})^\bullet$ and $enabld \in (t^{\#}_{i,j})^\bullet$.

  As regards the multiplicities, if $f = (init_{app}, t^{\#}_{i,j})$ then $V(f) = 1$, and if $f = (contd, t^{\#}_{i,j})$ or $f = (t^{\#}_{i,j}, enabld)$ then $V(f) = 1$; furthermore, if $f = (p, t^{\#}_{i,j})$ or $f = (t^{\#}_{i,j}, p)$, where $p = (a, i)$, then $V(f) = max\{lhs(r_{i,j})(a), prom^{r_{i,j}}(a)\}$. The time interval assigned to $t^{\#}_{i,j}$ is $[1, 1]$.

The novelty in this Petri net is the appearance of the transitions $T^{\#\#} = \{t^{\#\#}_{i,j} \mid r_{i,j} \in R_i\}$, that are responsible for the correct simulation of the inhibitors.

- Let $r_{i,j} \in R_i$, where $1 \leq j \leq |R_i|$, $1 \leq i \leq n$; then $t^{\#\#}_{i,j} \in T^{\#\#}$ is the transition checking condition 3 in Definiton 10. Let $p = (a,i) \in P_0$ and let $t^{\#\#}_{i,j} \in T^{\#}$; then $p \in {}^\bullet(t^{\#\#}_{i,j}) \cap (t^{\#\#}_{i,j})^\bullet$.
  If $f = (p, t^{\#\#}_{i,j})$ or $f = (t^{\#\#}_{i,j}, p)$, where $p = (a,i)$, then $V(f) = inhib^{r_{i,j}}(a)$.
  The time interval assigned to $t^{\#\#}_{i,j}$ is $[0,0]$.

In words, the transitions $t^{\#\#}_{i,j}$ capture the tokens of $p = (a,i)$ in the case when $|p| \geq inhib^{r_{i,j}}(a)$. The firing sequence can continue with the simulation of the application of rule $r_{i,j}$ only if $|p| < inhib^{r_{i,j}}(a)$ holds for every $a \in O$ for which $lhs(r_{i,j})(a) > 0$.

The rest of the construction is the same as that of Theorem 1, hence we omit the details. The changes in the Petri net compared to the core model are illustrated in Figures 4. □

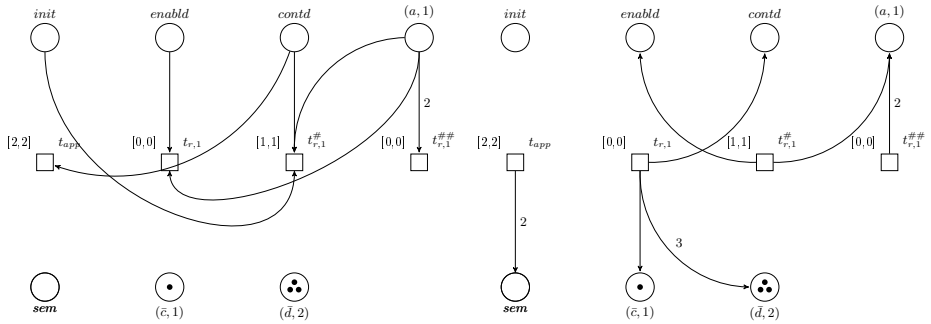


Fig. 4. The rule application phase for the Petri net, where $a \in w_1$ and $r = a \to c(d, in_2)^3$ and $prom^r(a) = inhib^r(a) = 1$.

Next, we turn our attention to membrane systems with dissolution. Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \delta)$ be a membrane system with dissolution. We recall form our previous definitions that this means that there exists a special element $\delta \in O$, which can appear on the right side of a rule only. Assume $r \in R_i$ and $\delta ß rhs(r)$. Suppose r is chosen in the actual maximal parallel rule application of m_i. Then all the rules of R_i appearing in that computational step are executed as usual, and, after the maximal parallel step is over, the region m_i disappears, its objects wander into the parent region and the rules R_i cease to operate. With this in mind, we construct a time Petri net simulating the operation of the membrane system in the sense below.

Theorem 3. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \delta)$ be a membrane system with dissolution. Then there is a time Petri net $N = (P, T, F, V, m_0, I)$ such that N halts if and only if Π halts and, if either of them halts, then both systems provide the same result.*

Proof. Let Π be as in the theorem. We construct the Petri net N with the required properties. The construction again leans on the proof of Theorem 1. The rule application phase is exactly the same with one exception: places δ_i symbolizing the dissolution of membrane m_i appear. The difference manifests itself in the definition of the communication phase. Moreover, we introduce one more phase, a δ-phase, that serves for moving the elements of a previously dissolved membrane to the parent region. First of all, we define the set of places as before.

- $P = P_0 \cup \bar{P}_0 \cup \{init_{app}, init_{com}, sem, enabld, \delta_i\}$, where $P_0 = O \times \{1, \ldots, n\}$ and $\bar{P}_0 = \bar{O} \times \{1, \ldots, n\}$ and $1 \leq i \leq n$. Let $m_0(p) = w_j(a)$ for every place $p = (a, j) \in P_0$.

The only change is the presence of the places δ_i for every region m_i. Intuitively, they are indicators whether a membrane is going to disappear in the next step or has been dissolved already. This is reflected in the design of the arcs for the rule application phase. We require an extended set of transitions, since we have a third phase also that transfers the objects of the dissolved membranes to their parent membranes.

- $T = T_0 \cup T_0^* \cup T_0^\delta \cup T^\# \cup T^\flat \cup \{t_{app}, t_{com}, t_{clean}, t^1_{sem}, t^2_{sem}, t^3_{sem}\}$,

and the arcs for the rule application phase are identical to those of Theorem 1 with the only exception of the arcs pointing from $t_{i,j}$, where $r_{i,j} \in R_i$, to the place δ_i with multiplicity 1 provided $\delta \in rhs(r_{i,j})$. Thus we are only interested in the transitions T_0^*, $T^\flat$ and their corresponding arcs.

- Let $T_0^* = \{s_{a,i} \mid a \in O,\ 1 \leq i \leq n\}$ and $T_0^\delta = \{s^\delta_{a,i} \mid a \in O,\ 1 \leq i \leq n\}$. Let m_i be a region other than the skin membrane, assume m_k is its parent region. (If m_i is the skin membrane, it cannot disappear.) Let $\bar{p} = (\bar{a}, i) \in \bar{P}_0$, then $\bar{p} \in {}^\bullet s_{a,i}$ and, if $p = (a, i)$, then $p \in s^\bullet_{a,i}$. Moreover, $init_{com} \in {}^\bullet s_{a,i} \cap s^\bullet_{a,i}$. In addition, δ_i is connected with $s^\delta_{a,i}$ for every object $a \in O$, that is: $\delta_i \in {}^\bullet s^\delta_{a,k} \cap s^\delta_{a,i}{}^\bullet$ and we have $\bar{q} = (\bar{a}, k) \in s^\delta_{a,i}{}^\bullet$. Regarding the multiplicities, each arc has multiplicity 1.
  Furthermore, $I(s_{a,i}) = [1, 1]$, $I(s^\delta_{a,i}) = [0, 0]$ and $I(t_{com}) = [2, 2]$, where t_{com} is the transition connecting $init_{com}$ with the place sem.

Intuitively, if m_i is a region with parent region m_k, the communication phase transfers the tokens of $\bar{p} = (\bar{a}, i)$ to the place $p = (a, i)$, as long as the membrane m_i exists. When m_i is marked for dissolution or has been already dissolved, that is, $|\delta_i| = 1$, then the tokens of $\bar{p} = (\bar{a}, i)$ are redirected to $\bar{q} = (\bar{a}, k)$. This is achieved by a time gap between the possible firings of the transitions $s_{a,i}$ and $s^\delta_{a,i}$. This implies that the elements appearing on the right hand side of the rules of a dissolved membrane find their correct place: they wander to the upper levels of the

tree until they find the first ancestor region not dissolved. The main ingredients of the construction are illustrated in Figure 5.

The only missing part is the subnet directing the remaining object of a dissolved membrane into an existing container membrane. We term this phase the cleaning phase. The construction is quite simple: let m_k be a region and assume that m_i is its parent region. Then, for every place $p = (a, k)$, there corresponds a transition $t_p^f lat$ which transfers the objects of p to $q = (a, i)$ when δ_k contains a token. The place $init_{clean}$ is connected to all the transitions $t_p^\flat$ in order to perceive when the tidying phase is ready. After this, 3 tokens are sent to the semaphore and a new application phase activates. More formally,

- let $T^\flat = \{t_{a,k}^\flat \mid a \in O,\ 1 \leq k \leq n\}$. Assume m_i is the parent region of region m_k. Then $p = (a, k) \in {}^\bullet t_p^\flat$ and $q = (a, i) \in t_p^{\flat\bullet}$ and $\delta_k \in {}^\bullet t_p^\flat \cap t_p^{\flat\bullet}$. Moreover, $init_{clean} \in {}^\bullet t_p^\flat \cap t_p^{\flat\bullet}$ and $init_{clean} \in {}^\bullet t_{clean}$ and $sem \in t_{clean}^\bullet$. Regarding the multiplicities, each arc has multiplicity 1, except for (t_{clean}, sem), which has multiplicity 3.
  Furthermore, $I(t_{a,i}^\flat) = [0, 0]$, $I(t_{clean}) = [1, 1]$.

This is described in Figure 6. The semaphore is extended with a new transition, t_{sem}^3 which leads to the initialization of the third phase in the simulation of a maximal parallel step. The new semaphore is depicted in Figure 7.

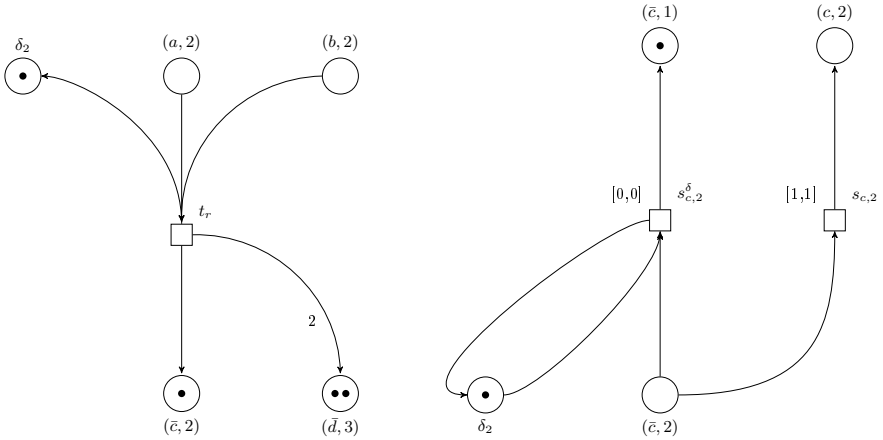


Fig. 5. The Petri net simulating a membrane system with dissolution. In this case, m_2 is dissolved, hence $s_{c,2}^\delta$ can be activated moving the elements of $\bar{p}(c, 2)$ to $\bar{p}(c, 1)$.

□

Finally, we tackle the problem of to the representation membrane systems with priorities in terms of Petri nets. Again, our construction is a slight modification of

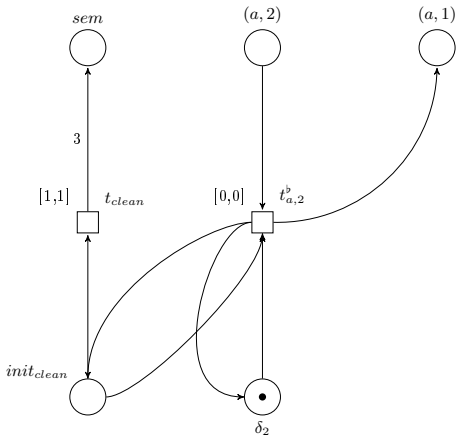


Fig. 6. The Petri net simulating the phase when the objects of a dissolved membrane are directed towards the parent membrane. Here we assume that region 1 is the parent of region 2, and the place δ_2 already has a token.

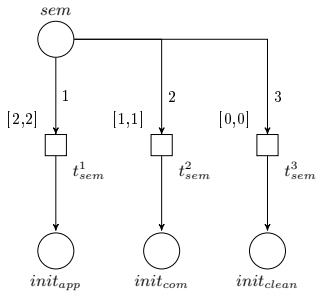


Fig. 7. The semaphore for the Petri net with dissolution. The choice of the next phase is uniquely determined by the number of tokens arriving in the place *sem*.

the core model. What we have to do is to introduce some pieces of information in the simulation of the rule application phase that accounts for the treatment of the priorities. Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho)$ be a membrane system with priorities. This means that $\rho \subseteq R \times R$, and the rule application is modified in the following way.

Definition 11. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho)$ be a membrane system with priorities. Let $r \in R_i$, if $1 \leq i \leq n$. Then r is applicable, if*

1. *$lhs(r_i) \leq w_i$,*
2. *for every $r' \in R_i$ such that $r' > r$, r' is not applicable.*

Let r_1, $r_2 \in R_k$ be two rules of region m_k, assume that $(r_1, r_2) \in \rho$, that is, $r_1 > r_2$. Then, considering a computational step, r_2 can be applied if r_2 is

applicable in the usual sense and, in addition, r_1 fails to be applicable in the maximal parallel step belonging to region m_k. We remark that we use priority in the strong sense: assume $w_k = a^2b$, $r_1 = a \to c$ and $r_2 = ab \to d$. Then the result of the maximal parallel step will be ad, instead of cd, since $r_1 > r_2$ and r_1 is applicable, which implies that r_2 cannot be applied in that maximal parallel step at all, even if r_1 is not applicable any more. We understand applicability in the sense of Definition 11, that is, a rule is not only required to have enough resources for being a candidate for that computational step, but also it is demanded that no other rule with higher priority should be applicable in that stronger sense. Moreover, conforming to the applications of priority suggested by the literature on P systems, we stipulate that priorities do not interfere with each other, i.e., no rule appears on the left hand side of a priority relation and on the right hand side of a (possibly different) priority relation. We can handle the membrane systems even if we omit this stipulation: to obtain an idea how to treat the other case, the reader should refer to page 526 in [3].

Now we are in a position to state the theorem on the simulation.

Theorem 4. *Let $\Pi = (O, \mu, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho)$ be a membrane system with priorities. Then there is a time Petri net $N = (P, T, F, V, m_0, I)$ such that N halts if and only if Π halts and, if either of them halts, then they provide the same result.*

Proof. Let Π as above. We describe the Petri net N simulating Π. The only differences in comparison with the model in Theorem 1 occur by the rule application phase when we select the transitions that stand for the applicable rules. We adopt the stipulation that no rule can occur both on the left hand side of a priority relation and on the right hand side of a (possibly different) priority relation. We omit repeating the construction of the Petri net in detail and we confine ourselves to the rule application phase that represents the difference. The places are

- $P = P_0 \cup \bar{P}_0 \cup PT^A \cup PT^{NA} \cup \{init_{app}, init_{com}, sem, enabld\}$, where $P_0 = O \times \{1, \ldots, n\}$ and $\bar{P}_0 = \bar{O} \times \{1, \ldots, n\}$ and the auxiliary places are defined as in Theorem 1. Regarding the new places, $PT^A = \{pt^A_{i,j} \mid r_{i,j} \in R_i, 1 \leq j \leq |R_i|\}$ and $PT^{NA} = \{pt^{NA}_{i,j} \mid r_{i,j} \in R_i, 1 \leq j \leq |R_i|\}$.

The new places accomplish some bookkeeping in order to keep track of which rules are applicable and which ones are not. A token in $pt^A_{i,j}$ should symbolize the applicability of an arbitrary $r_{i,j} \in R_i$, while a token in $pt^{NA}_{i,j}$ should mean that $r_{i,j}$ is not applicable in that maximal parallel step. We define now the new transitions together with the arcs induced by these transitions.

- $T = T_0 \cup T_0^* \cup T^\# \cup T_\rho \cup T^\flat \cup \{t_{app}, t_{com}, t^1_{sem}, t^2_{sem}\}$.

The transitions are defined as before, except for the elements of T_ρ and $T^\flat$, where $T_\rho = \{t_{r_i > r_j} \mid r_i, r_j \in m_k, \text{ and } (r_1, r_2) \in \rho\}$ and $T^\flat = \{t^\flat_{i,j} \mid t_{i,j} \in T_0\}$. We retain the arcs defined in the construction of the core model between the places and transitions. Modifications take place only in connection with the new states and transitions. We detail the rule application phase only.

- Let $r_{i,j} \in R_i$, where $1 \leq j \leq |R_i|$, $1 \leq i \leq n$; then, as in the previous constructions, $t^{\#}_{i,j} \in T^{\#}$ is checking the applicability of $r_{i,j}$. Let $p = (a, i) \in P_0$ and let $t^{\#}_{i,j} \in T^{\#}$; then $p \in {}^{\bullet}(t^{\#}_{i,j}) \cap (t^{\#}_{i,j})^{\bullet}$ and $init_{app} \in {}^{\bullet}t^{\#}_{i,j}$ and $enabld \in {}^{\bullet}t^{\#}_{i,j} \cap (t^{\#}_{i,j})^{\bullet}$, which is a slight modification compared to Theorem 1. Furthermore, $pt^{A}_{i,j} \in t^{\#}_{i,j}{}^{\bullet}$.
  Regarding the multiplicities, $V((init_{app}, t^{\#}_{i,j})) = 1$, and if $f = (t^{\#}_{i,j}, enabld)$ then $V(f) = 1$; furthermore, if $f = (p, t^{\#}_{i,j})$ or $f = (t^{\#}_{i,j}, p)$ where $p = (a, i)$, then $V(f) = lhs(r_{i,j})(a)$. in addition, the multiplicity of $(t^{\#}_{i,j}, pt^{A}_{i,j})$ is 1.
- Now, we turn to determining the operations of the transitions T_ρ. Let $r_{i,j}$, $r_{i,k} \in R_i$ with $r_{i,k} > r_{i,j}$. Then $t_{r_{i,k} > r_{i,j}} \in T_\rho$, and $pt^{A}_{i,k} \in {}^{\bullet}t_{r_{i,k} > r_{i,j}} \cap t^{\bullet}_{r_{i,k} > r_{i,j}}$, and $pt^{A}_{i,j} \in {}^{\bullet}t_{r_{i,k} > r_{i,j}}$ and $pt^{NA}_{i,j} \in t^{\bullet}_{r_{i,k} > r_{i,j}}$.
  The multiplicities of all the new arcs is 1.
  The elements of $T^{\flat}$ collect the tokens that might remain in the places PT^{A} and PT^{NA} after a finished maximal parallel step. We have $pt^{A}_{i,j}$, $pt^{NA}_{i,j} \in {}^{\bullet}t^{\flat}_{i,j}$ for every index pair i, j such that $t^{\flat}_{i,j} \in T^{\flat}$. The multiplicities of the arc is 1.
  For each $t_{r_{i,k} > r_{i,j}} \in T_\rho$, we have $I(t_{r_{i,k} > r_{i,j}}) = [0, 0]$, moreover, $I(t^{\flat}_{i,j}) = [2, 2]$.
- Finally, let $t_{i,j} \in T_0$, where $1 \leq i \leq n$ and $1 \leq j \leq |R_i|$. Then, as before, $p = (a, i) \in {}^{\bullet}t_{i,j}$ if and only if $a \in lhs(r_{i,j})$. In addition, $enabld \in {}^{\bullet}t_{i,j} \cap t^{\bullet}_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq |R_i|)$.
  Regarding the weights of the arcs, let $p = (a, i)$ and $f = (p, t_{i,j}) \in F$. Then the weight of f is the multiplicity of $a \in O$ on the left-hand side of $r_{i,j}$, namely, $V(f) = lhs(r_{i,j})(a)$. If $f = (t_{i,j}, enabld)$ or $f = (enabld, t_{i,j})$, then $V(f) = 1$ $(1 \leq i \leq n, 1 \leq j \leq |R_i|)$. Moreover, if t_{app} is the transition connecting $enabld$ to sem, then $I(t_{app}) = [2, 2]$.

The definition of the communication part is the same as that of the proof of Theorem 1, we ignore repeating the rest of the construction.

Intuitively, the tokens in the places $pt^{A}_{i,j}$, where $r_{i,j} \in R_i$, stand for the applicability of rule $r_{i,j}$ in region m_i. If $r_{i,k} > r_{i,j}$, then the token in $pt^{A}_{i,j}$ is directed into $pt^{NA}_{i,j}$ and remains there until that maximal parallel step is finished. The time intervals assigned to the transitions $t_{r_{i,k} > r_{i,j}}$ ensure that every such pair of the priority relation is discovered before we begin the actual simulation of the rule applications. Hence, only transitions corresponding to rules applicable in the weak sense of of priority are able to fire. If no more rule can be applied, the token in $enabld$ is passed over to sem at time instance 2, and the simulation of the rule application phase terminates. □

6 Conclusions

In this paper, we have made a step forward in relating the membrane systems and time Petri nets. We connected membrane systems with promoters/inhibitors,

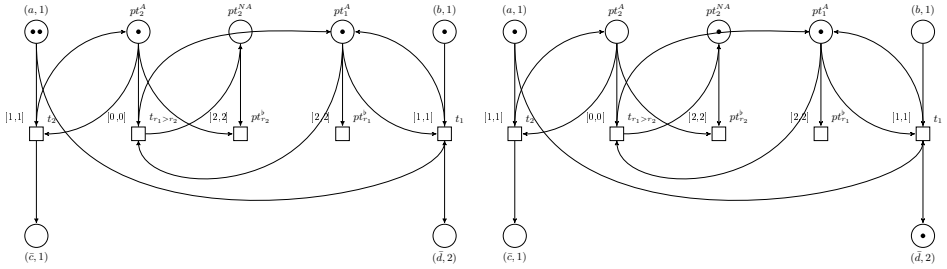


Fig. 8. Assume $w_1 = a^2b$ and $r_1,\ r_2 \in R_1$, $r_1 = ab \to d$, $r_2 = a \to c$ such that $r_1 > r_2$. Then t_1 can fire only, the token from pt_2^A eventually moves to pt_2^{NA}.

membrane dissolution and priority for rules with time Petri nets by extending the Petri net model presented in [4]. We preserved the main characteristic of Petri nets, namely, the firings of the transitions can take place in any order: we do not impose any additional condition on the transition sequences in order to obtain a Petri net model equivalent to the general Turing machine. We can ignore the requirement of computing with maximal parallel transition sequences in the case of the Petri nets. Instead, our simulating Petri net model adopts the usual semantics: the fireable transitions can fire in any possible order.

References

1. B. Aman, G. Ciobanu. Adding Lifetime to Objects and Membranes in P Systems. *International Journal of Computers, Communications and Control*, 5(3) (2010) 268–279.
2. B. Aman, G. Ciobanu. Verification of Membrane Systems with Delays via Petri Nets with Delays. *Theoretical Computer Science*, 598 (2015) 87–101.
3. B. Aman, P. Battyányi, G. Ciobanu, Gy. Vaszil, Simulating P systems with membrane dissolution in a chemical calculus. *Natural Computing* 15 (4) (2016), 521-532.
4. B. Aman, P. Battyányi, G. Ciobanu, Gy. Vaszil. Local time membrane systems and time Petri nets. *Theoretical Computer Science*, (2018), https://doi.org/10.1016/j.tcs.2018.06.013
5. B. Aman, G. Ciobanu, G.M. Pinna. Timed Catalytic Petri Nets. In Proceedings SYNASC, IEEE Computer Society, 319–326, 2012.
6. M. Cavaliere, D. Sburlan. Time and Synchronization in Membrane Systems. *Fundamenta Informaticae*, 64(1) (2005) 65 – 77.
7. M. Cavaliere, D. Sburlan. Time Independent P Systems Towards a Petri Net Semantics for Membrane Systems. *Lecture Notes in Computer Science*, vol.3365, 239–258, 2005.
8. G. Ciobanu, G.M. Pinna. Catalytic and Communicating Petri Nets are Turing Complete. *Information and Computation*, 239 (2014) 55–70.
9. R. Freund, O. Ibarra, A. Păun, P. Sosík, H.-C. Yen. *Catalytic P Systems.* In [19], 83–117, 2010.
10. M.H.T. Hack. Decidability Questions for Petri Nets, *PhD Thesis*, M.I.T., 1976.

11. M. Ionescu, Gh. Păun, T. Yokomori. Spiking Neural P Systems. *Fundamenta Informaticae*, 71 (2006) 279–308.
12. R.M. Karp, R.E. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3 (1969) 147–195.
13. J.H.C.M. Kleijn, M. Koutny, G. Rozenberg. Towards a Petri Net Semantics for Membrane Systems. *Lecture Notes in Computer Science*, vol.3850, 292–309, 2005.
14. S.R. Kosaraju. Decidability of Reachability in Vector Addition Systems. *14th ACM Symposium on Theory of Computing*, 267–281, 1982.
15. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríguez-Patón. Tissue P Systems. *Theoretical Computer Science*, 296 (2003) 295–326.
16. E.W. Mayr. Persistence of Vector Replacement Systems is Decidable. *Acta Informatica*, 15 (1981) 309–318.
17. P.M. Merlin. A Study of the Recoverability of Computing Systems. *PhD Thesis*, University of California, Irvine, 1974.
18. Gh. Păun. *Membrane Computing - An Introduction*, Springer, 2002.
19. Gh. Păun, G. Rozenberg, A. Salomaa. *The Oxford Handbook of Membrane Computing*. Oxford University Press, 2010.
20. C.A. Petri. Kommunikation mit Automaten. *Dissertation*, Universität Hamburg, 1962.
21. L. Popova. On Time Petri Nets. *Journal of Information Processing and Cybernetics*, 27(4) (1991) 227–244.
22. L. Popova-Zeugmann. *Time and Petri Nets*, Springer, 2013.

Further Results on the Power of Generating APCol Systems

Lucie Ciencialová[1], Luděk Cienciala[1], and Erzsébet Csuhaj-Varjú[2]

[1] Institute of Computer Science, Silesian University in Opava, Czech Republic
lucie.ciencialova@fpf.slu.cz
ludek.cienciala@fpf.slu.cz
[2] Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary
csuhaj@inf.elte.hu

Summary. In this paper we continue our investigations in APCol systems (Automaton-like P colonies), variants of P colonies where the environment of the agents is given by a string and the functioning of the system resembles to the functioning of standard finite automaton. We first deal with the concept of determinism in these systems and compare deterministic APCol systems with deterministic register machines. Then we focus on generating non-deterministic APCol systems with only one agent. We show that these systems are as powerful as 0-type grammars, i.e., generate any recursively enumerable language. If the APCol system is non-erasing, then any context-sensitive language can be generated by a non-deterministic APCol systems with only one agent.

1 Introduction

Automaton-like P colonies (APCol systems, for short), introduced in [1], are variants of of P colonies (introduced in [10]) - very simple membrane systems inspired by colonies of formal grammars. The interested reader is referred to [14] for detailed information on membrane systems (P systems) and to [11] and [5] for more information to grammar systems theory. For more details on P colonies consult the surveys [9] and [4].

An APCol system consists of a finite number of agents - finite collections of objects in a cell - and their joint shared environment. The agents have programs consisting of rules. These rules are of two types: they may change the objects of the agents and they can be used for interacting with the joint shared environment of the agents. While in the case of standard P colonies the environment is a multiset of objects, in case of APCol systems it is represented by a string. The number of objects inside each agent is set by definition and it is usually a very small number: 1, 2 or 3. The string representing the environment is processed by the agents and it is used as an indirect communication channel for the agents as well, since through the string, the agents are able to affect the behaviour of another agent. It can easily

be observed that APCol systems resemble automata. The current configuration of the system (the objects inside the agents) and the current string representing the environment correspond to the current state of the automaton and the currently processed input string.

The agents may perform rewriting, communication or checking rules [10]. A rewriting rule $a \to b$ allows the agent to rewrite one object a to object b. Rewriting rules are also called evolution rules. Both objects are placed inside the agent. Communication rule $c \leftrightarrow d$ makes possible to exchange object c placed inside the agent with object d in the string. A checking rule is formed from two rules r_1, r_2 of type rewriting or communication. It sets a kind of priority between the two rules r_1 and r_2. The agent tries to apply the first rule and if it cannot be performed, then the agent performs the second rule. The rules are combined into programs in such a way that all objects inside the agent are affected by execution of the rules. Thus, the number of rules in the program is the same as the number of objects inside the agent.

The computation in APCol systems starts with the an input string, representing the initial state of the environment, and with each agents having only symbols e inside.

A computational step means a maximally parallel action of the active agents, i.e., agents that can apply their rules. Every symbol can be object of the action of only one agent. The computation ends if the input string is reduced to the empty word, there are no more applicable programs in the system, and meantime at least one of the agents is in so-called final state. This mode of computation is called accepting. APCol systems can also be used not only for accepting but generating strings. For more detailed information on APCol systems we refer to [2, 3].

In the first part of this paper, we deal with both variants of modes of computation. In general, a computation of APCol system is non-deterministic. It means that in every configuration one set of maximal sets of applicable programs is non-deterministically chosen to be executed. We focus on such APCol systems that there exists only one maximal set of applicable programs in each configuration - deterministic APCol systems. The second part of this paper is devoted to non-deterministic generating APCol systems with one agent only.

2 Preliminaries and Basic Notions

Throughout the paper we assume the reader to be familiar with the basics of the formal language theory and membrane computing [15, 14].

For an alphabet Σ, the set of all words over Σ (including the empty word, ε), is denoted by Σ^*. We denote the length of a 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 $M = (O, f)$, where O is an arbitrary (not necessarily finite) set of objects and f is a mapping $f : O \to N$; f assigns to each object in O its multiplicity in M. Any multiset of objects M with the set of

objects $O = \{x_1, \dots x_n\}$ can be represented as a string w over alphabet O with $|w|_{x_i} = f(x_i);\ 1 \leq i \leq n$. Obviously, all words obtained from w by permuting the letters can also represent the same multiset M, and ε represents the empty multiset.

2.1 Register machine

Definition 1. [13] *A register machine is a construct* $M = (m, H, l_0, l_h, P)$ *where*

- m *is the number of registers,*
- H *is the set of instruction labels,*
- l_0 *is the start label,*
- l_h *is the final label,*
- P *is a finite set of instructions injectively labelled with the elements from the set* H*.*

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

$l_1 : (ADD(r), l_2, l_3)$ Add 1 to the content of the register r and proceed to the instruction (labelled with) l_2 or l_3.

$l_1 : (SUB(r), l_2, l_3)$ If the register r stores the value different from zero, then subtract 1 from its content and go to instruction l_2, otherwise proceed to instruction l_3.

$l_h : HALT$ Halt the machine. The final label l_h is only assigned to this instruction.

If every ADD-instructions of M is of the form ADD-instruction $l_1 : (ADD(r), l_2)$, then M is called a deterministic register machine.

Register machine M accepts a set $N(M)$ of numbers in the following way: it starts with number $x \in N$ in the first register and all the other registers are empty (hence storing the number zero) and with the instruction labelled l_0. Then it proceeds to apply the instructions as indicated by the labels (and made possible by the contents of registers). If it reaches the halt instruction and all registers are empty, the input is said to be accepted by M and hence it is introduced in $N(M)$.

It is known that any recursively enumerable set of natural numbers can be accepted by a deterministic register machine with at most three registers.

Register machines can also generate sets of natural numbers. In this case the first register is dedicated as the output register. The register machine starts with empty registers (registers storing zero) and with instruction l_0. Then it proceeds with executing instructions, according to their labels. After halting, the generated number is the value of stored in the first register.

2.2 APCol systems

In the following we recall the notion of an APCol system (an automaton-like P colony) [1].

As standard P colonies, agents of the APCol systems contain objects, each of them is an element of a finite alphabet. Every agent is associated with a set of programs, every program consists of two rules that can be one of the following two types. The first one, called an evolution rule or a rewriting rule, is of the form $a \rightarrow b$. This means that object a inside of the agent is rewritten to object b. The second type of rules, called a communication rule, is of the form $c \leftrightarrow d$. When this rule is applied, object c inside the agent and a symbol d in the string representing the environment (the input string) are exchanged. If $c = e$, then the agent erases d from the input string and if $d = e$, symbol c is inserted into the string.

The computation in APCol systems starts with an input string, representing the environment, and with each agent having only symbols e inside.

A computation step means a maximally parallel action of the active agents, i.e., a maximal number of agents that can perform at least one of their programs, has to execute such an action parallel. Every symbol can be object of the action of only one agent. The computation ends if the input string is reduced to the empty word and there are no more applicable programs in the whole system, and meantime at least one of the agents is in so-called final state.

An APCol system is a construct

$$\Pi = (O, e, A_1, \ldots, A_n), \text{ where}$$

- O is an alphabet; its elements are called the objects,
- $e \in O$, called the basic object,
- $A_i,\ 1 \leq i \leq n$, are agents. Each agent is a triplet $A_i = (\omega_i, P_i, F_i)$, where
  - ω_i is a multiset over O, describing the initial state (content) of the agent, $|\omega_i| = 2$,
  - $P_i = \{p_{i,1}, \ldots, p_{i,k_i}\}$ is a finite set of programs associated with the agent, where each program is a pair of rules. Each rule is in one of the following forms:
    - $a \rightarrow b$, where $a, b \in O$, called an evolution rule,
    - $c \leftrightarrow d$, where $c, d \in O$, called a communication rule,
  - $F_i \subseteq O^*$ is a finite set of final states (contents) of agent A_i.

In the following we explain the work of an APCol system.

During the work of the APCol system, the agents perform programs. Since both rules in a program can be communication rules, an agent can work with two objects in the string in the same step of the computation. In the case of program $\langle a \leftrightarrow b; c \leftrightarrow d \rangle$, a substring bd of the input string is replaced by string ac. Notice that although the order of rules in the programs is usually irrelevant, here it is significant, since it expresses context-dependence. If the program is of the form $\langle c \leftrightarrow d; a \leftrightarrow b \rangle$, then a substring db of the input string is replaced by string ca. Thus, the agent is allowed to act only at one position of the string in the one step of the computation and the result of its action to the string depends both on the order of the rules in the program and on the interacting objects. In particular, we have the following types of programs with two communication rules:

- $\langle a \leftrightarrow b; c \leftrightarrow e\rangle$ - b in the string is replaced by ac,
- $\langle c \leftrightarrow e; a \leftrightarrow b\rangle$ - b in the string is replaced by ca,
- $\langle a \leftrightarrow e; c \leftrightarrow e\rangle$ - ac is inserted in a non-deterministically chosen place in the string,
- $\langle e \leftrightarrow b; e \leftrightarrow d\rangle$ - bd is erased from the string,
- $\langle e \leftrightarrow d; e \leftrightarrow b\rangle$ - db is erased from the string,
- $\langle e \leftrightarrow e; e \leftrightarrow d\rangle$; $\langle e \leftrightarrow e; c \leftrightarrow d\rangle$, ...- these programs can be replaced by programs of type $\langle e \rightarrow e; c \leftrightarrow d\rangle$.

At the beginning of the work of the APCol system (at the beginning of the computation), the environment is given by a string ω of objects which are different from e. This string represents the initial state of the environment. Consequently, an initial configuration of the APCol system is an $(n+1)$-tuple $c = (\omega; \omega_1, \ldots, \omega_n)$ where w is the initial state of the environment and the other n components are multisets of strings of objects, given in the form of strings, the initial states the of agents.

A configuration of an APCol system Π is given by $(w; w_1, \ldots, w_n)$, where $|w_i| = 2$, $1 \leq i \leq n$, w_i represents all the objects inside the ith agent and $w \in (O - \{e\})^*$ is the string to be processed.

At each step of the computation every agent attempts to find one of its programs to use. If the number of applicable programs is higher than one, then the agent non-deterministically chooses one of them. At one step of computation, the maximal possible number of agents have to be active, i.e., have to perform a program.

By applying programs, the APCol system passes from one configuration to another configuration. A sequence of configurations started from the initial configuration is called a computation. A configuration is halting if the APCol system has no applicable program.

The result of computation depends on the mode in which the APCol system works. In the case of accepting mode a computation is called accepting if and only if at least one agent is in final state and the string to be processed is ε. Hence, the string ω is accepted by the APCol system Π if there exists a computation by Π such that it starts in the initial configuration $(\omega; \omega_1, \ldots, \omega_n)$ and the computation ends by halting in the configuration $(\varepsilon; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

In [1] it was shown that the family of languages accepted by jumping finite automata (introduced in [12]) is properly included in the family of languages accepted by APCol systems with one agent. It was also proved that any recursively enumerable language can be obtained as a projection of a language accepted by an APCol system with two agents.

In the case of generating mode, the string w_F is generated by Π iff there exists computation starting in an initial configuration $(\varepsilon; \omega_1, \ldots, \omega_n)$ and the computation ends by halting in the configuration $(w_F; w_1, \ldots, w_n)$, where at least one of $w_i \in F_i$ for $1 \leq i \leq n$.

In both cases, instead of a string, we can work with the number of symbols in the string as the result of the computation. The set of natural numbers accepted or generated with an APCol system Π is denoted $N_{acc}(\Pi)$ or $N_{gen}(\Pi)$, respectively.

The family of sets of numbers generated by APCol systems with n agents is denoted by $NAPCol_{gen}(n)$, if we consider only restricted APCol systems, then we use notation $NAPCol_{gen}R(n)$. The family of recursively enumerable sets of natural numbers is denoted by NRE, and the family of sets of natural numbers acceptable by partially blind register machines is denoted by NRM_{pb}.

Results have been obtained about the generative power of APCol systems[3]:

- Restricted APCol systems with two agents working in generating mode generate any recursively set of natural numbers and conversely. Thus,

$$NAPCol_{gen}R(2) = NRE.$$

- The family of sets of natural numbers acceptable by partially blind register machines can be generated by restricted APCol systems with one agent and conversely. Thus,

$$NRM_{pb} \subseteq NAPCol_{gen}R(1).$$

3 Generative Power of APCol Systems

3.1 Deterministic APCol systems

The concept of determinism can be interpreted for APCol systems in several ways. Here we consider the following concept:

Let $c = (w_1, \ldots, w_n; w_E)$ be an arbitrary configuration of an APCol system $\Pi = (O, e, A_1, \ldots, A_n)$, $n \geq 1$. We say that Π is deterministic if there is only one maximal applicable multiset of programs M_P in Π that can be applied to c.

We can construct an n-tuple x_c of strings of length two, $x_i y_i$ corresponding to the string that agent i consumes from the environmental string by applying a program from M_P. If there is rewriting rule in the program, then e appears in the string $x_i y_i$. If some agent has no applicable program then it is represented by ee in the x_c. Let O be a set of objects and Σ is input alphabet, then let $f : O \to \Sigma^*$ be a function defined as follows: $\forall x \in O - \{e\} f(x) = a; f(e) = \varepsilon$ and

$$u_0\ a_{i_1} b_{i_1}\ u_1\ a_{i_2} b_{i_2}\ u_2 \ldots u_{n-1}\ a_{i_n} b_{i_n}\ u_n = w_E \tag{1}$$

We focus on deterministic APCol system working in generating mode.

The deterministic APCol system working in generating mode starts its computation in initial configuration given by initial contents of agents and with empty string as environmental string. By execution of programs it passes from one configuration to another one. Notice that to every configuration there is only one maximal set of programs such that environmental string is formed as (1). The result of the computation - environmental string - is obtained only if APCol system halts and at least one agent is in final state.

Theorem 1. *Let M be a deterministic register machine. Then there exists a deterministic APCol system Π with two agents such that M and Π generate the same set of natural numbers.*

Idea of the proof:

The environmental string stores information about the contents of the registers. It is in a form $\#111\ldots1222\ldots2333\ldots n\#'$. When ADD-instruction is performed on register r, then an agent puts $\downarrow$ just after $\#$ and moves $\downarrow$ through the string from the left to the right until the agent consumes the number $s \geq r$. Then the agent insert new symbol r just before s, deletes $\downarrow$ and generates the label of the next instruction performed by register machine.

The idea how to do zero-check, and thus subtraction $(l_1 : (SUB(r), l_2, l_3))$ is the following: Content of register r is represented by the number of objects r in the environmental string. If the agent needs to erase some r, then it places mark $\uparrow$ just after $\#$ and moves it through the string. If there is any object r, then the agent erases it and generates label l_2. If there is no r, then the agent consumes $\uparrow$ together with s $(s > r)$ or $\#'$ it generates label l_3.

If the next instruction is the halt instruction, then the agent exchanges $\#$ with $\updownarrow$ and pushes it through the string. it leaves symbol 1 unchanged, the symbols representing the contents of other registers are deleted. Finally, if agent consume $\#'$, then it erases $\updownarrow$ and stops working.

It can be seen that the instruction of the register machine can be simulated by the deterministic APCol system.

3.2 APCol systems with one agent

In this part we deal with APCol systems with only one agent and working in non-deterministic manner.

By the analogy of a non-decreasing Chomsky grammar, we introduce the notion of a non-decreasing APCol systems working in the generating mode. We say that an APCol system Π is non-decreasing, if no agent of Π has a rule of the form $e \leftrightarrow y$, i.e., there is no rule for erasing a symbol from the string representing the environment.

We first show that any ε-free context-sensitive language can be generated by an APCol system with only one agent. Furthermore, the APCol system is non-decreasing.

Theorem 2. *Any ε-free context-sensitive language can be generated by a non-decreasing APCol system with only one agent.*

Sketch of the proof:

We show that to every context-sensitive grammar $G = (N, T, P, S)$ in Kuroda normal form there exists an APCol system Π with one agent working in generating

mode such that $L(G) = L(\Pi)$. To do this, we construct an APCol system Π such that every generation in G can be simulated by a computation in Π and any successful computation in Π corresponds to a terminating derivation in G. To help the easier reading, we provide only the description of the simulation of the rules of G, the other components of Π can be extracted from the descriptions below.

At the beginning of the simulating computation in Π we need to initialize environmental string - there is only starting non-terminal at the beginning of derivation in G. Let symbols $X, X' \notin N \cup T$.

Initialization

Agent	Program	String
(ee)	$\langle e \to S; e \to X' \rangle$	ε
(SX')	$\langle S \leftrightarrow e; X' \to X \rangle$	ε
(eX)		S

There are four types of rules in grammar in Kuroda normal form: $AB \to CD$; $A \to BC; A \to B$ and $A \to a$, $a \in T$, $A, B, C, D \in N$.

To every rule in form $AB \to CD$, to the set of programs of the agent A, we add the following set of programs:

$p_i : AB \to CD$

Agent	Program	String
(eX)	$\langle e \to p_i; X \to X' \rangle$	$u\ AB\ v$
(p_iX')	$\langle p_i \leftrightarrow A; X' \leftrightarrow B \rangle$	$u\ p_i'X'\ v$
(AB)	$\langle A \to p_i''; B \to C \rangle$	$u\ p_i'X'\ v$
$(p_i''C)$	$\langle p_i'' \to p_i'''; C \leftrightarrow p_i' \rangle$	$u\ p_i'X'\ v$
$(p_i'''p_i')$	$\langle p_i''' \to q_i; p_i' \to D \rangle$	$u\ CX'\ v$
(q_iD)	$\langle q_i \to q_i'; D \leftrightarrow X' \rangle$	$u\ CX'\ v$
$(q_i'X')$	$\langle q_i' \to X; X \to e \rangle$	$u\ CD\ v$
(Xe)		$u\ CD\ v$
$(p_i''C)$	$\langle p_i'' \to A; C \to B \rangle$	$u\ p_i'X'\ v$

The program $\langle p_i'' \to A; C \to B \rangle$ can be used just after program $\langle p_i'' \to p_i'''; C \leftrightarrow p_i' \rangle$ and these two programs can cause loop in computation. This is because there can exist more than one rule with AB on the left side and the agent can generate label of a rule different from p_i.

For every rule in a form $A \to BC$ we add the following programs to the set of programs of A:

$p_i : A \to BC$

Agent	Program	String
(eX)	$\langle e \to p_i; X \to X' \rangle$	$u\ A\ v$
(p_iX')	$\langle p_i \to p_i'; X' \leftrightarrow A \rangle$	$u\ A\ v$
$(p_i'A)$	$\langle p_i' \to p_i''; A \to B \rangle$	$u\ X'\ v$
$(p_i''B)$	$\langle p_i'' \leftrightarrow X'; B \leftrightarrow e \rangle$	$u\ X'\ v$
$(X'e)$	$\langle X' \leftrightarrow p_i''; e \to e \rangle$	$u\ Bp_i''\ v$
$(p_i''e)$	$\langle p_i'' \to p_i'''; e \to C \rangle$	$u\ BX'\ v$
$(p_i'''C)$	$\langle p_i''' \to p_i'''; C \leftrightarrow X' \rangle$	$u\ BX'\ v$
$(p_i'''X')$	$\langle p_i''' \to e; X' \to X \rangle$	$u\ BC\ v$

For every rule in a form $A \to B$ or $A \to a$ we add the following programs to the set of programs of A (α is non-terminal or terminal symbol):

$p_i : A \to \alpha$	Agent Program	String
(eX)	$\langle e \to p_i; X \to \alpha \rangle$	$u\ A\ v$
$(p_i\alpha)$	$\langle p_i \to p_i'; \alpha \leftrightarrow A \rangle$	$u\ A\ v$
$(p_i'A)$	$\langle p_i' \to e; A \to X \rangle$	$u\ \alpha\ v$

At the end we add one more set of programs to the set of programs of A that is of the form

$$\langle e \to Y; X \leftrightarrow A \rangle ; \langle Y \to Y; A \to A \rangle$$

where $Y \notin N \cup T$ and $A \in N$. By execution of the first program, the computation enters a loop and the computation never halts. We have to add these programs to ensure that the computation will not halt if the derivation of a string in the grammar stops with a non-terminal in the string.

The agent simulates execution of rules and computation ends only when there is no non-terminal symbol in environmental string. By the definition of the rules and the programs above, it can easily be seen that the program set of agent A can only simulate the rules of G, furthermore any computation in Π successfully halts only if the corresponding derivation successfully terminates in G. We also observe that Π is non-decreasing.

It is known that any recursively enumerable language can be generated by a Kuroda-like normal form grammar $G = (N, T, P, S)$, where the rules are one of the forms $AB \to CD$, $A \to BC, A \to B$, $A \to a$, and $A \to \varepsilon$, where $a \in T$, $A, B, C, D \in N$. Modifying the proof of the above theorem (simulation of rule $A \to \alpha$), we can extend the proof to obtain any recursively enumerable language. The modification is the following: in the case of $A \to \varepsilon$ we use rule $e \leftrightarrow A$ in the corresponding rule set.

We also note that to generate a context-sensitive language which contains ε, we have an extension of the Kuroda normal form grammar where the rules are one of the forms $AB \to CD$, $A \to BC, A \to B$, $A \to a$, and $S \to \varepsilon$, where $a \in T$, $A, B, C, D \in N$, and S does not appear at the right-hand side of any rule. It is easy to see that the proof of the above theorem can be modified to be a proof of this statement as well.

Let CS, CS^{ε}, RE denote the family of ε-free context-sensitive, context-sensitive, and recursively enumerable languages, respectively. Since APCol systems (both in the generating and in the accepting mode) can be simulated by Turing machines, we obtain the following statement

Theorem 3.

$$CS \subset CS^{\varepsilon} \subset RE = APCol_{gen}(1).$$

4 Conclusion

In this paper we examined APCol systems working in generating mode. We defined a deterministic version of APCol systems and showed that they are able to simulate

functioning of deterministic register machines. In second part of paper we focused on generating APCol systems with only one agent. We have showed that these systems generate the family of recursively enumerable languages, and if they are non-decreasing (have no rule for decreasing the length of the environment).

Acknowledgments.

The work of L. Ciencialová and L. Cienciala was supported by The Ministry of Education, Youth and Sports from the National Programme of Sustainability (NPU II) project IT4Innovations excellence in science - LQ1602, by SGS/11/2019. The work of E. Csuhaj-Varjú was supported by Grant No. K 120558 of the National Research, Development, and Innovation Office, Hungary.

References

1. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: Towards on P Colonies Processing Strings. In: Proc. BWMC 2014, Sevilla, 2014. pp. 102–118. Fénix Editora, Sevilla, Spain (2014)
2. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: P colonies processing strings. Fundamenta Informaticae 134(1-2), 51–65 (2014)
3. Cienciala, L., Ciencialová, L., Csuhaj-Varjú, E.: A Class of Restricted P Colonies with String Environment. Natural Computing 15(4), 541–549 (2016)
4. L. Ciencialová, E. Csuhaj-Varjú, L. Cienciala, and P. Sosík. P colonies. Bulletin of the International Membrane Computing Society 1(2):119–156 (2016).
5. Csuhaj-Varjú, E., Kelemen, J., Păun, Gh., Dassow, J.(eds.): Grammar Systems: A Grammatical Approach to Distribution and Cooperation. Gordon and Breach Science Publishers, Inc., Newark, NJ, USA (1994)
6. Csuhaj-Varjú, E., Vaszil, G.: Finite dP Automata versus Multi-head Finite Automata In: Gheorghe, M. et. al. (eds.) CMC 2011, LNCS, vol. 7184, pp. 120-138. Springer-Verlag, Berlin Heidelberg (2012)
7. Holzer, M., Kutrib, M., Malcher, A.: Complexity of multi-head finite automata: Origins and directions, Theoretical Computer Science 412, 83–96 (2011)
8. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, Mass. (1979)
9. Kelemenová, A.: P Colonies. Chapter 23.1, In: Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) The Oxford Handbook of Membrane Computing, pp. 584–593. Oxford University Press (2010)
10. Kelemen, J., Kelemenová, A., Păun, G.: Preview of P Colonies: A Biochemically Inspired Computing Model. In: Workshop and Tutorial Proceedings. Ninth International Conference on the Simulation and Synthesis of Living Systems (Alife IX). pp. 82–86. Boston, Mass (2004)
11. Kelemen, J., Kelemenová, A.: A Grammar-Theoretic Treatment of Multiagent Systems. Cybern. Syst. 23(6), 621–633 (1992),
12. Meduna, A., Zemek, P.: Jumping Finite Automata. Int. J. Found. Comput. Sci. 23(7), 1555–1578 (2012)

13. Minsky, Marvin L.: Computation: Finite and Infinite Machines. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1967)
14. Păun, Gh., Rozenberg, G., Salomaa, A.(eds.): The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
15. Rozenberg, G., Salomaa, A.(eds.): Handbook of Formal Languages I-III. Springer Verlag., Berin-Heidelberg-New York (1997)

Playing with Derivation Modes and Halting Conditions

Rudolf Freund

TU Wien, Institut für Logic and Computation
Favoritenstraße 9–11, 1040 Wien, Austria
E-mail: rudi@emcc.at

Summary. In the area of P systems, besides the standard maximally parallel derivation mode, many other derivation modes have been investigated, too. In this paper, many variants of hierarchical P systems and tissue P systems using different derivation modes are considered and the effects of using different derivation modes, especially the maximally parallel derivation modes and the maximally parallel set derivation modes, on the generative and accepting power are illustrated. Moreover, an overview on some control mechanisms used for (tissue) P systems is given.

Furthermore, besides the standard total halting mode, we also consider different halting conditions such as unconditional halting and partial halting and explain how the use of different halting modes may considerably change the computing power of P systems and tissue P systems.

1 Introduction

The basic model of *P systems* as introduced in [19] can be considered as a distributed multiset rewriting system, where all objects – if possible – evolve in parallel in the membrane regions and may be communicated through the membranes. But also P systems operating on more complex objects (e.g., strings, arrays) are often considered, too, for instance, see [8].

Besides the maximally parallel derivation mode, many other derivation modes have been investigated during the last two decades. Thus in this paper the definitions of the standard derivation modes used for P systems and tissue P systems are recalled. Various interpretations of derivation modes known from the P systems area are illustrated and well-known results are presented in a different manner.

Moreover, we not only consider the standard total halting, but also other halting conditions such as unconditional halting, see [5], and partial halting, see [12]. We explain and give some examples how the use of different halting modes may considerably change the computing power of P systems and tissue P systems.

Overviews on the field of P systems can be found in the monograph [20] and the Handbook of Membrane Computing [21]; for actual news and results we refer

to the P systems webpage [23] as well as to the Bulletin of the International Membrane Computing Society. The reader is assumed to be very familiar with the basic definitions and notations of P systems and tissue P systems as well as of the commonly used derivation modes and halting conditions.

2 Prerequisites

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

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

The family of regular, context-free, and recursively enumerable string languages is denoted by REG, CF, and RE, respectively. For example, $PsREG = PsCF$, which is the reason why in the area of multiset rewriting CF plays no role at all, and in the area of membrane computing we usually get characterizations of $PsREG$ and $PsRE$.

An *extended Lindenmayer system* (an *E0L system* for short) is a construct $G = (V, T, P, w)$, where V is an alphabet, $T \subseteq V$ is the terminal alphabet, $w \in V^*$ is the axiom, and P is a finite set of non-cooperative rules over V of the form $a \rightarrow u$. In a derivation step, each symbol present in the current sentential form is rewritten using one rule arbitrarily chosen from P. The language generated by G, denoted by $L(G)$, consists of all the strings over T which can be generated in this way by starting from the initial string w. An E0L system with $T = V$ is called a *0L system*.

For more details of formal language theory the reader is referred to the monographs and handbooks in this area as [7] and [22].

Register machines

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

- $l_1 : (ADD(j), l_2, l_3)$, with $l_1 \in B \backslash \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  Increases the value of register j by one, followed by a non-deterministic jump to instruction l_2 or l_3. This instruction is usually called *increment*.
- $l_1 : (SUB(j), l_2, l_3)$, with $l_1 \in B \backslash \{l_h\}$, $l_2, l_3 \in B$, $1 \leq j \leq m$.
  If the value of register j is zero then jump to instruction l_3; otherwise, the value of register j is decreased by one, followed by a jump to instruction l_2. The two cases of this instruction are usually called *zero-test* and *decrement*, respectively.
- $l_h : HALT$. Stops 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 instruction l_0 of P, and terminate with reaching the HALT-instruction l_h.

M is called deterministic if in all ADD-instructions $p : (\texttt{ADD}\,(r)\,, q, s)$, it holds that $q = s$; in this case we write $p : (\texttt{ADD}\,(r)\,, q)$.

For useful results on the computational power of register machines, we refer to [18]; for example, deterministic register machines can accept all recursively enumerable sets of vectors of natural numbers with k components using precisely $k + 2$ registers.

3 A General Model for Tissue and Hierarchical P Systems

We now recall the main definitions of the general model for tissue P systems and hierarchical P systems and the basic derivation modes as defined, for example, in [16]. Moreover, we define the halting conditions discussed in this paper.

A *(hierarchical) P system (with rules of type X)* working in the derivation mode δ is a construct

$$\Pi = (V, T, \mu, w_1, \ldots, w_m, R_1, \ldots, R_m, f, \Longrightarrow_{\Pi,\delta}) \text{ where}$$

- V is the alphabet of *objects*;
- $T \subseteq V$ is the alphabet of *terminal objects*;
- μ is the hierarchical membrane structure (a rooted tree of membranes) with the membranes uniquely labeled by the numbers from 1 to m;
- $w_i \in V^*$, $1 \leq i \leq m$, is the *initial multiset* in membrane i;
- R_i, $1 \leq i \leq m$, is a finite set of *rules of type X* assigned to membrane i;

- f is the label of the membrane from which the result of a computation has to be taken from (in the generative case) or into which the initial multiset has to be given in addition to w_f (in the accepting case),
- $\Longrightarrow_{\Pi,\delta}$ is the derivation relation under the derivation mode δ.

The symbol X in "rules of type X" may stand for "evolution", "communication", "membrane evolution", etc. In this paper, we will mainly consider non-cooperative as well as catalytic and purely catalytic rules, see Subsection 3.2.

In hierarchical P systems, the membranes are arranged in a tree structure. If we allow arbitrary graphs as communication structure, with the membranes now also called *cells*, floating in the environment instead of being enclosed in the skin membrane, we come to the model of tissue P systems, where in the static case we simply number the cells from 1 to m:

A *(static) tissue P system (with rules of type X)* working in the derivation mode δ is a construct

$$\Pi = (V, T, m, w_1, \ldots, w_m, R_1, \ldots, R_m, f, \Longrightarrow_{\Pi,\delta}) \text{ where}$$

- V is the alphabet of *objects*;
- $T \subseteq V$ is the alphabet of *terminal objects*;
- m is the number of cells uniquely labeled by the numbers from 1 to m;
- $w_i \in V^*$, $1 \leq i \leq m$, is the *initial multiset* in cell i;
- R_i, $1 \leq i \leq m$, is a finite set of *rules of type X* assigned to cell i;
- f is the label of the cell from which the result of a computation has to be taken from (in the generative case) or into which the initial multiset has to be given in addition to w_f (in the accepting case),
- $\Longrightarrow_{\Pi,\delta}$ is the derivation relation under the derivation mode δ.

Each of the cells may have assigned its own set of rules R_i, but in the most general case the rules (for multisets) are of the form

$$(1, u_1) \ldots (m, u_m) \rightarrow (1, v_1) \ldots (m, v_m)$$

where $u_1, \ldots, u_m$ and $v_1, \ldots, v_m$ are multisets over V, and then instead of $R_1, \ldots, R_m$ we specify only one set of rules R for the whole tissue P system Π.

A configuration is a list of the contents of each cell or membrane region, respectively; a sequence of configurations $C_1, \ldots, C_k$ is called a *computation* in the derivation mode δ if $C_i \Longrightarrow_{\Pi,\delta} C_{i+1}$ for $1 \leq i < k$. The derivation relation $\Longrightarrow_{\Pi,\delta}$ is defined by the set of rules in Π and the given derivation mode which determines the multiset of rules to be applied to the multisets contained in each membrane or cell or even in the overall tissue P system.

The *language generated by* Π is the set of all terminal multisets which can be obtained in the output membrane / cell f starting from the initial configuration $C_1 = (w_1, \ldots, w_m)$ using the derivation mode δ in a halting computation, i.e.,

$$L_{gen,\delta}(\Pi) = \left\{ C(f) \in T^{\circ} \mid C_1 \stackrel{*}{\Longrightarrow}_{\Pi,\delta} C \wedge \neg\exists C' : \; C \Longrightarrow_{\Pi,\delta} C' \right\},$$

where $C(f)$ stands for the multiset contained in the output membrane or cell f of the configuration C. The configuration C is halting, i.e., no further configuration C' can be derived from it.

The family of languages of multisets generated by P systems and tissue P systems of type X with at most n membranes / cells in the derivation mode δ is denoted by $Ps_{gen,\delta}OP_n(X)$ and $Ps_{gen,\delta}OtP_n(X)$, respectively.

We may also consider (tissue) P systems as accepting mechanisms: in membrane / cell f, we add the input multiset w_0 to w_f in the initial configuration $C_1 = (w_1, \ldots, w_m)$ thus obtaining $C_1[w_0] = (w_1, \ldots, w_f w_0, \ldots, w_m)$; the input multiset w_0 is accepted if there exists a halting computation in the derivation mode δ starting from $C_1[w_0]$, i.e.,

$$L_{acc,\delta}(\Pi) = \left\{ w_0 \in T^{\circ} \mid \exists C : \; \left(C_1[w_0] \stackrel{*}{\Longrightarrow}_{\Pi,\delta} C \wedge \neg\exists C' : \; C \Longrightarrow_{\Pi,\delta} C' \right) \right\}.$$

Then the family of languages of multisets accepted by P systems and tissue P systems of type X with at most n membranes / cells in the derivation mode δ is denoted by $Ps_{acc,\delta}OP_n(X)$ and $Ps_{acc,\delta}OtP_n(X)$, respectively.

We finally mention that (tissue) P systems can also be used to compute functions and relations, with using f both as input and output membrane / cell or even using two different membranes / cells for the input and the output. Yet in this paper we will mainly focus on the generating case.

3.1 Derivation Modes

The set of all multisets of rules applicable in a (tissue) P system to a given configuration C is denoted by $Appl(\Pi, C)$ and can be restricted by imposing specific conditions, thus yielding the following basic derivation modes (for example, see [16] for formal definitions):

- asynchronous mode (abbreviated *asyn*): at least one rule is applied;
- sequential mode (*sequ*): only one rule is applied;
- maximally parallel mode (*max*): a non-extendable multiset of rules is applied;
- maximally parallel mode with maximal number of rules (max_{rules}): a non-extendable multiset of rules of maximal possible cardinality is applied;
- maximally parallel mode with maximal number of objects ($max_{objects}$): a non-extendable multiset of rules affecting as many objects as possible is applied.

In [3], these derivation modes are restricted in such a way that each rule can be applied at most once, thus yielding the set modes *sasyn*, *smax*, $smax_{rules}$, and $smax_{objects}$ (the sequential mode is already a set mode by definition):

- asynchronous set mode (abbreviated *sasyn*): at least one rule is applied, but each rule at most once;

- maximally parallel set mode ($smax$): a non-extendable set of rules is applied;
- maximally parallel set mode with maximal number of rules ($smax_{rules}$): a non-extendable set of rules of maximal possible cardinality is applied;
- maximally parallel set mode with maximal number of objects ($smax_{objects}$): a non-extendable set of rules affecting as many objects as possible is applied.

Let us denote the set of all multisets (possibly only sets) of rules applicable in a (tissue) P system Π to a given configuration C in the derivation mode δ by $Appl(\Pi, C, \delta)$. We immediately observe that $Appl(\Pi, C, asyn) = Appl(\Pi, C)$.
To collect the set and multiset derivation modes, we use the following notations:

$D_S = \{sequ, sasyn, smax, smax_{rules}, smax_{objects}\}$ and
$D_M = \{asyn, max, max_{rules}, max_{objects}\}$.

3.2 Standard Rule Variants

Non-cooperative rules have the form $a \to w$, where a is a symbol and w is a multiset, catalytic rules have the form $ca \to cw$, where the symbol c is called the *catalyst*, and cooperative rules have no restrictions on the form of the left-hand side. These types of rules will be denoted by *ncoo* (*non-cooperative*), *pcat* (*purely catalytic*), and *coo* (*cooperative*); if both non-cooperative and catalytic rules are allowed, we write *cat* (*catalytic*).

If the P system has more than one membrane, each symbol on the right-hand side may have assigned a target where the symbol has to be sent after the application of the rule. In tissue P systems this target is simply the number of the cell, whereas in hierarchical P systems the targets take into account the tree structure of the membranes:

here the symbol stays in the membrane where the rule is applied;
out the symbol is sent to the outer membrane, i.e., the membrane enclosing the membrane where the rule is applied;
in the symbol is sent to an inner membrane, i.e., a membrane enclosed by the membrane where the rule is applied;
in_j the symbol is sent to the inner membrane labeled by j.

3.3 Flattening

As many variants of P systems can be *flattened* to only one membrane, see [11], we often may assume the simplest membrane structure of only one membrane which in effect reduces the P system to a multiset processing mechanism, and, observing that $f = 1$, in what follows we then will use the reduced notation

$$\Pi = (V, T, w, R, \Longrightarrow_{\Pi,\delta}) .$$

For a one-membrane system, the definitions for the *language generated by* Π and the *language accepted by* Π can be written in an easier way, i.e.,

$$L_{gen,\delta}(\Pi) = \left\{ v \in T^\circ \mid w \overset{*}{\Longrightarrow}_{\Pi,\delta} v \wedge \neg\exists z : \ v \Longrightarrow_{\Pi,\delta} z \right\} \text{ and}$$
$$L_{acc,\delta}(\Pi) = \left\{ w_0 \in T^\circ \mid \exists v : \ \left(w w_0 \overset{*}{\Longrightarrow}_{\Pi,\delta} v \wedge \neg\exists z : \ v \Longrightarrow_{\Pi,\delta} z \right) \right\}.$$

The family of languages of multisets generated by one-membrane P systems of type X in the derivation mode δ is denoted by $Ps_{gen,\delta}OP(X)$.

The family of languages of multisets accepted by one-membrane P systems of type X in the derivation mode δ is denoted by $Ps_{acc,\delta}OP(X)$.

In the following, we will mainly focus on the generative case, and when writing $Ps_{\delta}OP(X)$ we by default will mean $Ps_{gen,\delta}OP(X)$.

3.4 Halting Conditions

Besides the standard total halting with no (multi)set of rules being applicable any more to the current configuration, some more variants of halting conditions have been considered in the literature:

total halting (H) the common halting strategy where the computation stops with no (multi)set of rules being applicable any more

unconditional halting (u) the result of a computation can be taken from every configuration derived from the initial one (possibly only taking terminal results)

partial halting (h) the set of rules R is partitioned into disjoint subsets R_1 to R_h, and a computation stops if there is no multiset of rules applicable to the current configuration which contains a rule from every set R_j, $1 \leq j \leq h$

halting with states (s) the configuration with which a derivation may stop must fulfill a recursive condition (which corresponds with a *final state*)

The variant of unconditional halting was introduced in [5]. Partial halting, for example, was investigated in [2, 4, 12], using the membranes for partitioning the rules. Formal definitions for the halting conditions H, h, s can be found in [16].

In the description for (tissue) P systems, the derivation relation under the derivation mode δ, $\Longrightarrow_{\Pi,\delta}$, is extended by the halting condition, i.e., we then write $\Longrightarrow_{\Pi,\delta,\beta}$ for $\beta \in \{H, h, u, s\}$. Moreover, we add the halting condition in the description of the generated or accepted language, i.e., we then write $L_{\gamma,\delta,\beta}(\Pi)$, $\gamma \in \{gen, acc\}$. The same extension is made for the corresponding families of languages of multisets, i.e., for $n \geq 1$, we write $Y_{\gamma,\delta,\beta}OP_n(X)$ and $Y_{\gamma,\delta,\beta}OtP_n(X)$, respectively. By default, β is understood to be the total halting H and then usually omitted in all these notations.

4 Some Well-Known Results

In this section we recall some well-known results, which usually are not stated in the compact form given here.

4.1 Non-Cooperative Rules

Using only non-cooperative rules leaves us on the level of semi-linear sets, as for the derivation with context-free rules (and non-cooperative rules correspond to those), the resulting derivation tree does not depend on an interpretation of a sequential or a parallel derivation of any kind. Moreover, context-free (string or multiset) languages are closed under projections, hence, taking (even only terminal) results out from a specific output membrane / cell does not make any difference. Therefore, we may state the following result:

Theorem 1. *For any $Y \in \{N, Ps\}$ and any $n \geq 1$ as well as any derivation mode $\delta \in D_S \cup D_M$,*

$$Y_{gen,\delta}OP_n\,(ncoo) = Y_{gen,\delta}OtP_n\,(ncoo) = YREG.$$

Although P systems working in the maximally parallel derivation mode are a parallel mechanism, we cannot go beyond $PsREG$, see Theorem 1.

For example, the rule $a \to aa$ used in parallel very much reminds us of a $0L$ system, i.e., a Lindenmayer system of the simplest form, which, when starting from the axiom aa, yields the language $L_1 = \{a^{2^n} \mid n \geq 1\}$. In order to also get this language with P systems working in one of the maximally parallel derivation modes, we either need some control mechanism (see Section 5) or some other special halting condition (see Section 7).

4.2 The Importance of Using Catalysts

If in a one-membrane system we only have one catalyst c and only catalytic rules assigned to c, then this corresponds to a sequential use of non-cooperative rules, which together with Theorem 1 yields the following result:

Theorem 2. *For any $Y \in \{N, Ps\}$ and any derivation mode $\delta \in D_S \cup D_M$,*

$$Y_{gen,\delta}OP\,(pcat_1) = Y_{gen,sequ}OP\,(pcat_1) = Y_{gen,sequ}OP\,(ncoo) = YREG.$$

Without additional control mechanisms, at least three catalysts are needed to obtain computational completeness for purely catalytic P systems using the derivation mode max, see [10]. In a more general way, the following results were already proved there:

Theorem 3. *For any $d \geq 1$ and any $k \geq d + 2$,*

$$Ps_{acc,max}OP\,(pcat_{k+1}) = Ps_{acc,max}OP\,(cat_k) = N^dRE.$$

Although not yet stated in [10], we mention that these results are also valid when replacing the derivation mode max by any other maximally parallel (set) derivation mode, i.e., for any δ in

$$\{max, max_{rules}, max_{objects}, smax, smax_{rules}, smax_{objects}\}.$$

The complexity of the construction, for all these derivation modes, has been considerably reduced since the original paper from 2005, for example, see [3].

These results are obtained by simulating register machines, which in fact means that a sequential machine has to be simulated by a parallel mechanism. Exactly this feature of breaking down the parallelism to sequentiality is the main importance of using catalysts: when using a maximally parallel derivation mode $\delta \in \{max, max_{rules}, max_{objects}\}$, for decrementing the number of a symbol a_r to carry out the decrement case of a SUB-instruction of a register machine, we cannot do this by a non-cooperative rule $a_r \to \lambda$, instead we have to use a catalytic rule $ca_r \to c$.

What happens in the case of two catalysts in purely catalytic P systems (and one catalyst in the case of catalytic P systems), is one of the most intriguing open problems in the area of P systems since long time, e.g., see [15], where it is shown that catalytic P systems with one catalyst can simulate partially blind register machines and partially blind counter automata.

With respect to the importance of using catalytic rules, the set derivation modes offer new opportunities, i.e., using specific control mechanisms they are not needed any more, as eliminating only one symbol a_r to carry out the decrement case of a SUB-instruction of a register machine now *can* be done by a non-cooperative rule $a_r \to \lambda$, because due to the set restriction this rule is not applied more than once.

5 Control Mechanisms

To reduce the number of catalysts needed for obtaining computational completeness, specific control mechanisms can be used. Some of these control mechanisms are considered in this section. For example, label selection or control languages allow for using only one catalyst (two catalysts) in (purely) catalytic P systems for getting computational completeness, for instance, see [9, 13, 14, 3]. With target agreement and maximally parallel set derivation modes, catalysts can even be avoided completely, only non-cooperative rules are needed.

For all the control mechanisms described in this section, as a special example we will show how the $0L$ language $L_1 = \{a^{2^n} \mid n \geq 1\}$ can be generated using the maximally parallel derivation mode.

5.1 P Systems with Label Selection

For all the variants of (tissue) P systems of type X, we may consider labeling all the rules in the sets $R_1, \ldots, R_m$ in a one-to-one manner by labels from a set H

and taking a set W containing subsets of H. In any derivation step of a *(tissue) P system with label selection* Π we first select a set of labels $U \in W$ and then, in the given derivation mode, we apply a non-empty multiset R of rules such that all the labels of these rules from R are in U.

Example 1. Consider the one-membrane P system

$$\Pi = (V = \{A, a\}, T = \{a\}, w = AA, R = \{r_1 : A \to AA, r_2 : A \to a\},$$
$$W = \{\{r_1\}, \{r_2\}\}, \Longrightarrow_{\Pi,\max}).$$

with the labeled rules $r_1 : A \to AA$ and $r_2 : A \to a$; only one of these can be used according to the sets of labels in W. Using r_1 in $n-1$ derivation steps and finally using r_2 yields a^{2^n}, for any $n \geq 1$, i.e., we get $N_{gen,max}(\Pi) = L_1$, where $L_1 = \{a^{2^n} \mid n \geq 1\}$.

The families of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, $\gamma \in \{gen, acc\}$, and $\delta \in D_M \cup D_S$ computed by (tissue) P systems with label selection with at most m membranes and rules of type X are denoted by $Y_{\gamma,\delta}OP_m(X, ls)$ ($Y_{\gamma,\delta}OtP_m(X, ls)$).

Theorem 4. *$Y_{\gamma,\delta}OP(cat_1, ls) = Y_{\gamma,\delta}OP(pcat_2, ls) = YRE$ for any $Y \in \{N, Ps\}$, $\gamma \in \{gen, acc\}$, and any maximally parallel (set) derivation mode δ,*

$$\delta \in \{max, max_{rules}, max_{objects}, smax, smax_{rules}, smax_{objects}\}.$$

The proof given in [14] for the maximally parallel mode max can be taken over for the other maximally parallel (set) derivation modes word by word; the only difference again is that in set derivation modes, in non-successful computations where more than one trap symbol # has been generated, the trap rule $\# \to \#$ is only applied once.

5.2 Controlled (Tissue) P Systems and Time-Varying (Tissue) P Systems

Another method to control the application of the labeled rules is to use control languages (see [17] and [1]).

In a *controlled (tissue) P system* Π, in addition we use a set H of labels for the rules in Π, and a string language L over 2^H (each subset of H represents an element of the alphabet for L) from a family FL. Every successful computation in Π has to follow a control word $U_1 \dots U_n \in L$: in derivation step i, only rules with labels in U_i are allowed to be applied (in the underlying derivation mode, for example, max or $smax$), and after the n-th derivation, the computation halts; we may relax this end condition, i.e., we may stop after the i-th derivation for any $i \leq n$, and then we speak of *weakly controlled P systems*. If $L = (U_1 \dots U_p)^*$, Π is called a *(weakly) time-varying (tissue) 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*.

Example 2. Consider the one-membrane P system

$$\Pi = (V = \{A, a\}, T = \{a\}, w = AA, R = \{r_1 : A \to AA, r_2 : A \to a\},$$
$$L = \{r_1\}^*\{r_2\}, \Longrightarrow_{\Pi,\max})$$

with the labeled rules $r_1 : A \to AA$ and $r_2 : A \to a$. Using the control word ${r_1}^{n-1}r_2$ means using r_1 in $n-1$ derivation steps and finally using r_2, thus yielding a^{2^n}, for any $n \geq 1$, i.e., as in Example 1, we get $N_{gen,max}(\Pi) = L_1$.

As now we do not have to distinguish between non-terminal and terminal symbols due to the use of control words, the same result can be obtained by the much simpler system

$$\Pi' = (V = \{a\}, T = \{a\}, w = aa, R = \{r_1 : a \to aa\},$$
$$L = \{r_1\}^*, \Longrightarrow_{\Pi',\max})$$

again yielding $N_{gen,max}(\Pi') = L_1$.

The family of sets $Y_{\gamma,\delta}(\Pi)$, $Y \in \{N, Ps\}$, computed by (weakly) controlled P systems and (weakly) time-varying P systems with period p, with at most m membranes and rules of type X as well as control languages in FL is denoted by $Y_{\gamma,\delta}OP_m(X, C(FL))$ $(Y_{\gamma,\delta}OP_m(X, wC(FL)))$ and $Y_{\gamma,\delta}OP_m(X, TV_p)$ $(Y_{\gamma,\delta}OP_m(X, wTV_p))$, respectively, for $\gamma \in \{gen, acc\}$ and $\delta \in D_M \cup D_S$. Similar notations hold for tissue P systems.

Theorem 5. *$Y_{\gamma,\delta}OP(cat_1, \alpha TV_6) = Y_{\gamma,\delta}OP(pcat_2, \alpha TV_6) = YRE$, for any $\alpha \in \{\lambda, w\}$, $Y \in \{N, Ps\}$, $\gamma \in \{gen, acc\}$, and*

$$\delta \in \{max, max_{rules}, max_{objects}, smax, smax_{rules}, smax_{objects}\}.$$

The proof given in [14] for the maximally parallel mode *max* again can be taken over for the other maximally parallel (set) derivation modes word by word, e.g., see [3].

5.3 Target Selection

In P systems with target selection, all objects on the right-hand side of a rule must have the same target, and in each derivation step, for each region a (multi)set of rules – non-empty if possible – having the same target is chosen. In [3] it was shown that for P systems with target selection in the derivation mode *smax* ***no*** catalyst is needed any more, and with $smax_{rules}$, we even obtain a deterministic simulation of deterministic register machines:

Theorem 6. *For any $Y \in \{N, Ps\}$,*

$$Y_{gen,smax}OP(ncoo, target\ selection) = YRE.$$

Theorem 7. *For any* $Y \in \{N, Ps\}$,

$$Y_{detacc,smax_{rules}}OP\,(ncoo, target\ selection) = YRE.$$

In contrast to all the other variants of P systems, P systems with target selection really take advantage of the membrane structure, no flattening is used or even reasonable. In that sense, this variant of P systems reflects the spirit of membrane systems with a non-trivial membrane structure in the best way.

Example 3. Consider the two-membrane P system

$$\Pi = (V = \{a\}, T = \{a\}, \mu = [\,[\,]_2\,]_1, w_1 = aa, w_2 = \lambda,$$
$$R_1 = \{a \to aa, a \to (a, in)\}, R_2 = \emptyset, \Longrightarrow_{\Pi,\max})$$

with the rule $a \to aa$ having target *here* and the rule $a \to (a, in)$ having target *in*; only one of these two rules can be used in one derivation step according to the condition of target agreement. Using $a \to aa$ in $n-1$ derivation steps in the skin membrane and finally using $a \to (a, in)$ yields a^{2^n} in the elementary membrane $[\,]_2$, for any $n \geq 1$, i.e., we again get $N_{gen,max}(\Pi) = L_1$.

6 The Strangeness of Minimal Parallelism

There is another derivation mode known from literature, which has two possible basic definitions, but these two variants unfortunately do not yield the same results.

Following the definition given in [16], for the minimally parallel derivation mode (min), we need an additional feature for the set of rules R used in the overall (tissue) P system, i.e., we consider a partitioning θ of R into disjoint subsets R_1 to R_h. Usually, this partitioning of R may coincide with a specific assignment of the rules to the membranes or cells. We observe that this partitioning θ may, but need not be the same as the partitioning η used for partial halting.

There are now several possible interpretations of this minimally parallel derivation mode which in an informal way can be described as applying multisets such that from every set R_j, $1 \leq j \leq h$, at least one rule – if possible – has to be used (e.g., see [6]). Yet this *if possible* allows for two possible interpretations:

Minimal parallelism as a restriction of asyn

As defined in [16], we start with a multiset R' of rules from $Appl(\Pi, C, asyn)$ and only take it if it cannot be extended to a multiset R' of rules from $Appl(\Pi, C, asyn)$ by some rule from a set R_j from which so far no rule is in R'.

Minimal parallelism as an extension of smax

We start with a set R' of rules from $Appl(\Pi, C, smax_\theta)$, where the notion $smax_\theta$ indicates that we are using $smax$ with respect to the partitioning of R into the subsets R_1 to R_h, and then possibly extend it to a multiset R'' of rules from $Appl(\Pi, C, asyn)$ which contains R'. This definition finally was used in [21] without using the notion $smax$, because at the moment when this handbook was written the notion of maximally parallel set derivation modes had not been invented yet. Moreover, the use of the notion $smax$ so far was restricted to the discrete topology, where every rule formed its own set R_j, whereas for $smax_\theta$ the condition is fulfilled if *one* of the rules in the R_j is used if possible.

Example 4. Consider the one-membrane P system working in the min-mode

$$\Pi = (V = \{a, b\}, T = \{b\}, w = aa, R = R_1 \cup R_2, \Longrightarrow_{\Pi,min})$$

with $R_1 = \{a \to bb\}$ and $R_2 = \{a \to bbb\}$ being the partitions of $R = R_1 \cup R_2$.

Starting from $smax$, we get only one set of rules, i.e., $R' = \{a \to bb, a \to bbb\}$, whose application yields the result b^5.

In the case of starting with $asyn$, we may use one of the two rules twice, thus also getting the results b^4 and b^6.

Hence, when two rules are competing for the same objects, the results obtained with the two different definitions may be different, where the set of results obtained when using the first definition will always include the results obtained by the second definition.

The condition that the sets R_j, $1 \leq j \leq h$, have to be disjoint may be alleviated, for example, see [4].

A special variant of the minimally parallel derivation mode, with the sets R_j, $1 \leq j \leq h$, not being required to be disjoint, is the mode min_1, which in fact means that we stay with $smax_\theta$. Now let $smax_{\theta,k}$ denote a partioning θ with k sets of rules. As an interesting result we then get the interpretation of a purely catalytic P system using max as a P system using min_1 with the partitioning R_j, $1 \leq j \leq k$, where R_j is the set of non-cooperative rules $a \to u$ representing the corresponding catalytic rules $c_j a \to c_j u$. Denoting a partitioning in k sets of rules by θ_k, we obtain the following result:

Theorem 8. *For any $d \geq 1$ and any $k \geq d + 3$,*

$$Ps_{acc,min_1}OP\,(ncoo, \theta_k) = Ps_{gen,min_1}OP\,(ncoo, \theta_3) = N^dRE.$$

Minimal parallelism with all applicable sets

There is an even stranger variant for minimal parallelism already defined in [16]:

To a configuration C we can only apply a multiset of rules which contains at least one rule from each R_j, $1 \leq j \leq h$, that contains a rule applicable to C, i.e., we take all possible multisets R' from $Appl(\Pi, C, asyn)$ which also fulfill the condition that $R' \cap R_j \neq \emptyset$ provided $Appl(\Pi, C, asyn) \cap R_j \neq \emptyset$, for all $1 \leq j \leq h$.

This derivation mode is abbreviated $all_{aset}min$ in [16] and used under the notion *amin* in [4].

Example 5. Consider the one-membrane P system from Example 4, now working in the *amin*-mode,

$$\Pi = (V = \{a, b\}, T = \{b\}, w = aa, R = R_1 \cup R_2, \Longrightarrow_{\Pi, amin})$$

with $R_1 = \{a \to bb\}$ and $R_2 = \{a \to bbb\}$.

As both the rule from R_1 and the rule from R_2 are applicable, the only (multi)set of rules applicable to the configuration aa is the same as that one when starting from *smax*, i.e., $R' = \{a \to bb, a \to bbb\}$, whose application yields the result b^5.

Yet if we take $w = a$ instead, then still both the rule from R_1 and the rule from R_2 are applicable, but there are not enough resources of symbols a for applying both rules, hence, no derivation step is possible in this case with the derivation mode *amin*. On the other hand, with the first two variants of the minimally parallel derivation mode, in both cases we may either apply $a \to bb$ or $a \to bbb$, thus getting bb and bbb, respectively.

Again we observe that the results with different definitions of the minimally parallel derivation mode may be different when two rules are competing for the same object(s).

7 Halting Conditions

As already mentioned, P systems working in the maximally parallel derivation mode at first sight look like (E)0L systems. Only the total halting condition completely destroys this similarity which looks so obvious at first sight. Yet this connection between P systems working in the maximally parallel derivation mode and (E)0L systems can be shown when using unconditional halting, see [5].

Besides unconditional halting, in this section we will also discuss some results for partial halting and halting with states. In each case, as in Section 5, we will show how to obtain the special multiset language $L_1 = \{a^{2^n} \mid n \geq 1\}$.

7.1 Unconditional Halting

Example 6. Consider the one-membrane (or one-cell tissue) P system

$$\Pi = (V = \{a\}, T = \{a\}, w = aa, R = \{a \to aa\}, \Longrightarrow_{\Pi,\max,u})$$

with the single rule $a \to aa$; with every application of this rule the number of symbols a is doubled, i.e., after $n-1$ derivation steps, $n \geq 1$, we get a^{2^n}, i.e., we obtain $N_{gen,max,u}(\Pi) = L_1$.

According to the results shown in [5], the following results holds true, if we use extended systems (indicated by the additional symbol E) and only take results from the output membrane / cell which are terminal:

Theorem 9. *For any $Y \in \{N, Ps\}$ and any $m \geq 1$,*

$$Y_{gen,\delta,u}EOP_m\,(ncoo) = Y_{gen,\delta,u}EOtP_m\,(ncoo) = YE0L,$$

for any maximally parallel derivation mode δ,

$$\delta \in \{max, max_{rules}, max_{objects}\}.$$

If we do not use extended systems, i.e., $V = T$, we immediately obtain the following:

Corollary 1. *For any $Y \in \{N, Ps\}$,*

$$Y_{gen,\delta,u}OP_1\,(ncoo) = Y_{gen,\delta,u}OtP_1\,(ncoo) = Y0L,$$

for any maximally parallel derivation mode δ,

$$\delta \in \{max, max_{rules}, max_{objects}\}.$$

These results now show the – somehow expected – correspondence between the two parallel mechanisms (tissue) P systems and Lindenmayer systems.

We finally mention that with unconditional halting, considering acceptance would not make any sense, because according to the standard definition of accepting (tissue) P systems, in any case they would accept every input.

7.2 Partial Halting

Partial halting allows us to stop a derivation as soon as some specific symbols are not present any more:

Example 7. Consider the one-membrane P system

$$\Pi = (V = \{a, s\}, T = \{a\}, w = as, R_1 \cup R_2, \Longrightarrow_{\Pi,\max,h})$$

where $R_1 = \{a \to aa\}$ and $R_2 = \{s \to s, s \to \lambda\}$ are the two partitions of the rule set $R = \{a \to aa, s \to s, s \to \lambda\}$.

As long as one of the rules from R_2 can be applied to the symbol s, the symbols a are doubled as usual by the rule $a \to aa$ from R_1. Using $s \to s$ in $n-1$ derivation steps, $n \geq 1$, and finally applying $s \to \lambda$, we get a^{2^n}; hence, $N_{gen,max,h}(\Pi) = L_1$.

Some interesting results for the partial halting may be looked up in [2, 4, 12].

7.3 Halting with States

In general, speaking of states reminds us of mechanisms like register machines; there a computation halts when the halt instruction $l_h : HALT$ is applied. In simulations of register machines by (tissue) P systems the computation often is made halting by applying the final rule $l_h \to \lambda$, provided no trap rules are still applicable. When l_h disappears this means that no instruction label appears any more in the configuration of the simulating (tissue) P system; such a condition checking for the absence (or presence) of specific symbols in a given configuration is computable and therefore a condition we can use for halting with states (which ironically in this case means the absence of state symbols).

Example 8. Consider the one-membrane P system

$$\Pi = (V = \{a, s\}, T = \{a\}, w = as, R = \{a \to aa, s \to s, s \to \lambda\}, \Longrightarrow_{\Pi,\max,s}),$$

which uses the same ingredients as the one considered in Example 7, but instead of partial halting now uses the condition that a computation halts if no symbol s is present any more, which gives the same computations as for the P system in Example 7, with the only difference that the computations halt because of s having been deleted. Thus, we obtain $N_{gen,max,s}(\Pi) = L_1$.

8 Conclusion

In this paper the effects of using different derivation modes on the generative and accepting power of many variants of hierarchical P systems and tissue P systems have been illustrated. Especially some differences between the maximally parallel derivation modes and the maximally parallel set derivation modes have

been exhibited. We have also given an overview on some control mechanisms used for (tissue) P systems. Moreover, we have discussed the effect of using different halting conditions such as unconditional and partial halting.

Many more relations between derivation modes and halting conditions as well could have been discussed, but this would have gone much beyond such a normal article.

Acknowledgements

Many of the ideas for this paper came up in the inspiring atmosphere of the Brainstorming Week on Membrane Computing in Sevilla this year and even in some previous years, and they are based on many discussions with Artiom Alhazov, Sergiu Ivanov, and Sergey Verlan, but also other colleagues from the P community, especially with Gheorghe Păun.

References

1. Alhazov, A., Freund, R., Heikenwälder, H., Oswald, M., Rogozhin, Yu., Verlan, S.: Sequential P systems with regular control. In: Csuhaj-Varjú, E., Gheorghe, M., Rozenberg, G., Salomaa, A., Vaszil, Gy. (eds.) Membrane Computing – 13th International Conference, CMC 2012, Budapest, Hungary, August 28–31, 2012, Revised Selected Papers, Lecture Notes in Computer Science, vol. 7762, pp. 112–127. Springer (2013). https://doi.org/10.1007/978-3-642-36751-9_9
2. Alhazov, A., Freund, R., Oswald, M., Verlan, S.: Partial halting in P systems using membrane rules with permitting contexts. In: Durand-Lose, J., Margenstern, M. (eds.) Machines, Computations, and Universality. pp. 110–121. Springer, Berlin, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74593-8_10
3. Alhazov, A., Freund, R., Verlan, S.: P systems working in maximal variants of the set derivation mode. In: Leporati, A., Rozenberg, G., Salomaa, A., Zandron, C. (eds.) Membrane Computing – 17th International Conference, CMC 2016, Milan, Italy, July 25-29, 2016, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10105, pp. 83–102. Springer (2017). https://doi.org/10.1007/978-3-319-54072-6_6
4. Alhazov, A., Oswald, M., Freund, R., Verlan, S.: Partial halting and minimal parallelism based on arbitrary rule partitions. Fundam. Inform. **91**(1), 17–34 (2009). https://doi.org/10.3233/FI-2009-0031
5. Beyreder, M., Freund, R.: Membrane systems using noncooperative rules with unconditional halting. In: Corne, D.W., Frisco, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. pp. 129–136. Springer, Berlin, Heidelberg (2009). https://doi.org/10.1007/978-3-540-95885-7_10
6. Ciobanu, G., Pan, L., Păun, Gh., Pérez-Jiménez, M.: P systems with minimal parallelism. Theoretical Computer Science **378**(1), 117–130 (2007). https://doi.org/10.1016/j.tcs.2007.03.044
7. Dassow, J., Păun, Gh.: Regulated Rewriting in Formal Language Theory. Springer (1989), `https://www.springer.com/de/book/9783642749346`
8. Freund, R.: P systems working in the sequential mode on arrays and strings. Int. J. Found. Comput. Sci. **16**(4), 663–682 (2005). https://doi.org/10.1142/S0129054105003224

9. Freund, R.: Purely catalytic P systems: Two catalysts can be sufficient for computational completeness. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu. (eds.) CMC14 Proceedings – The 14th International Conference on Membrane Computing, Chişinău, August 20–23, 2013. pp. 153–166. Institute of Mathematics and Computer Science, Academy of Sciences of Moldova (2013), http://www.math.md/cmc14/CMC14_Proceedings.pdf
10. Freund, R., Kari, L., Oswald, M., Sosík, P.: Computationally universal P systems without priorities: two catalysts are sufficient. Theoretical Computer Science **330**(2), 251–266 (2005). https://doi.org/10.1016/j.tcs.2004.06.029
11. Freund, R., Leporati, A., Mauri, G., Porreca, A.E., Verlan, S., Zandron, C.: Flattening in (tissue) P systems. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Yu., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 8340, pp. 173–188. Springer (2014). https://doi.org/10.1007/978-3-642-54239-8_13
12. Freund, R., Oswald, M.: Partial halting in P systems. Int. J. Found. Comput. Sci. **18**(6), 1215–1225 (2007). https://doi.org/10.1142/S0129054107005261
13. Freund, R., Oswald, M.: Catalytic and purely catalytic P automata: control mechanisms for obtaining computational completeness. In: Bensch, S., Drewes, F., Freund, R., Otto, F. (eds.) Fifth Workshop on Non-Classical Models for Automata and Applications – NCMA 2013, Umeå, Sweden, August 13 – August 14, 2013, Proceedings. books@ocg.at, vol. 294, pp. 133–150. Österreichische Computer Gesellschaft (2013)
14. Freund, R., Păun, Gh.: How to obtain computational completeness in P systems with one catalyst. In: Neary, T., Cook, M. (eds.) Proceedings Machines, Computations and Universality 2013, MCU 2013, Zürich, Switzerland, September 9–11, 2013. EPTCS, vol. 128, pp. 47–61 (2013). https://doi.org/10.4204/EPTCS.128.13
15. Freund, R., Sosík, P.: On the power of catalytic P systems with one catalyst. In: Rozenberg, G., Salomaa, A., Sempere, J.M., Zandron, C. (eds.) Membrane Computing – 16th International Conference, CMC 2015, Valencia, Spain, August 17–21, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9504, pp. 137–152. Springer (2015). https://doi.org/10.1007/978-3-319-28475-0_10
16. Freund, R., Verlan, S.: A formal framework for static (tissue) P systems. In: Eleftherakis, G., Kefalas, P., Păun, Gh., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing, Lecture Notes in Computer Science, vol. 4860, pp. 271–284. Springer (2007). https://doi.org/10.1007/978-3-540-77312-2_17
17. Krithivasan, K., Păun, Gh., Ramanujan, A.: On controlled P systems. Fundam. Inform. **131**(3–4), 451–464 (2014). https://doi.org/10.3233/FI-2014-1025
18. Minsky, M.L.: Computation. Finite and Infinite Machines. Prentice Hall, Englewood Cliffs, NJ (1967)
19. Păun, Gh.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693
20. Păun, Gh.: Membrane Computing: An Introduction. Springer (2002). https://doi.org/10.1007/978-3-642-56196-2
21. Păun, Gh., Rozenberg, G., Salomaa, A. (eds.): The Oxford Handbook of Membrane Computing. Oxford University Press (2010)
22. Rozenberg, G., Salomaa, A. (eds.): Handbook of Formal Languages. Springer (1997). https://doi.org/10.1007/978-3-642-59136-5
23. The P Systems Website. http://ppage.psystems.eu/

Simulating counting oracles with cooperation

Alberto Leporati[1], Luca Manzoni[1], Giancarlo Mauri[1],
Antonio E. Porreca[2], and Claudio Zandron[1]

[1] Dipartimento di informatica, Sistemistica e Comunicazione
Università degli Studi di Milano-Bicocca,
Viale Sarca 336, 20126, Milan, Italy
alberto.leporati@unimib.it luca.manzoni@unimib.it
giancarlo.mauri@unimib.it claudio.zandron@unimib.it
[2] Aix Marseille Université, Université de Toulon, CNRS, LIS, Marseille, France
antonio.porreca@lis-lab.fr

Summary. We prove that monodirectional shallow chargeless P systems with active membranes and minimal cooperation working in polynomial time precisely characterise $\mathbf{P}_{\parallel}^{\#\mathbf{P}}$, the complexity class of problems solved in polynomial time by deterministic Turing machines with a polynomial number of parallel queries to an oracle for a counting problem.

1 Introduction

Many variants of P systems with active membranes [8] are able to solve traditionally intractable problems: with charges and bidirectional communication, uniform families of them are able to solve $\mathbf{P}^{\#\mathbf{P}}$-complete problems when only one level of membrane nesting (i.e., shallow systems) is allowed [2, 3], and **PSPACE**-complete problems when this restriction is removed [9]. The presence of simple cooperation rules, like the ones provided by antimatter, where two opposite objects can annihilate each other, allows the systems to reach $\mathbf{P}^{\#\mathbf{P}}$ with a shallow membrane structure, also when the systems have no charges [5]. Even when the communication is severely restricted, as in monodirectional systems, where send-in is forbidden, uniform families of P systems with active membranes with charges characterize $\mathbf{P}^{\mathbf{NP}}$ or, if shallow, the class $\mathbf{P}_{\parallel}^{\mathbf{NP}}$, as shown in [4]. It is interesting to see that this is not the case for monodirectional systems with antimatter: the additional cooperation provided by object annihilation makes possible to "count" once, thus allowing families of this kind of systems the ability to reach $\mathbf{P}^{\#\mathbf{P}[1]} = \mathbf{P}_{\parallel}^{\#\mathbf{P}}$, even with only one level of nesting [5].

In this paper we continue the investigation of the importance of cooperation to increase the computational power of P systems. In particular, we show that monodirectional systems with minimal cooperation [10] working in polynomial

time also characterize the class of all decision problems solvable in polynomial time by a deterministic Turing machine with access to a *single* query of a $\#\mathbf{P}$ oracle, i.e., $\mathbf{P}^{\#\mathbf{P}[1]} = \mathbf{P}_{\parallel}^{\#\mathbf{P}}$.

The paper is organized as follows: Section 2 introduces the basic notions necessary for the rest of the paper. Section 3 shows how a single $\#\mathbf{P}$ query can be simulated, and in Section 4 the main result is presented. Section 5 contains the conclusions, and shows some directions for future research.

2 Basic Notions

In this paper we consider (semi)uniform families of P systems with active membranes without charges and using minimal cooperative evolution rules $[ab \to w]_h$, send-out rules $[a]_h \to [\,]_h\, b$ and elementary division rules $[a]_h \to [b]_h\, [c]_h$, where a, b, and c are single objects and w is a multiset of objects. For the technical details we refer the reader to Valencia-Cabrera et al. [10].

We also consider polynomial-time Turing machines with oracles for counting problems in the complexity class $\#\mathbf{P}$ [7] and, in particular, the complexity classes $\mathbf{P}^{\#\mathbf{P}[1]}$, where only one query is allowed, and $\mathbf{P}_{\parallel}^{\#\mathbf{P}}$, when any polynomial number of queries is allowed, but they must all be carried out *in parallel*, that is, all query strings are prepared in advance before actually interrogating the oracle (in other words, later queries are not adaptive with respect to the answers to previous ones). The two classes $\mathbf{P}^{\#\mathbf{P}[1]}$ and $\mathbf{P}_{\parallel}^{\#\mathbf{P}}$ actually turn out to be equivalent:

Proposition 1 (Leporati et al. [5]). *A polynomial number of parallel $\#\mathbf{P}$ queries can be simulated by a single $\#\mathbf{P}$ query in polynomial time (in symbols $\mathbf{P}_{\parallel}^{\#\mathbf{P}} = \mathbf{P}^{\#\mathbf{P}[1]}$).*

Proof. A single query does never depend on the results of previous queries, thus $\mathbf{P}^{\#\mathbf{P}[1]} \subseteq \mathbf{P}_{\parallel}^{\#\mathbf{P}}$ by definition.

Conversely, let M be a deterministic Turing machine running in polynomial time $p(n)$ with parallel oracle queries for a function $f \in \#\mathbf{P}$, and let N be a nondeterministic Turing machine having $f(x)$ accepting computations for each input string x of length n and running in polynomial time $q(n)$.

Then $f(x) \leq 2^{q(|x|)}$ for each input string x, since $2^{q(n)}$ is the maximum number of computations of N on an input of length n (assuming binary nondeterministic choices). Clearly, due to its running time, the machine M can only ask queries with query strings of length bounded by $p(n)$, which means that each query answer is an integer bounded by $2^{q(p(n))}$, and M can ask up to $p(n)$ queries.

Let $x_1, x_2, \ldots, x_{p(n)}$ be the query strings of M on a run on a given input, letting $x_i = \epsilon$ if M asks less than i queries, and let $g\colon \Sigma^\star \to \mathbb{N}$, with Σ the union of the query alphabet and the separator symbol \$, be defined as

$$g(x_1\$x_2\$\ldots\$x_{p(n)}) = \sum_{i=1}^{p(n)} B^i \times f(x_i)$$

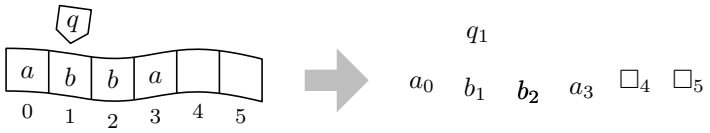
where $B = 2^{q(p(n))} + 1$; this corresponds to encoding all the query answers as a base-B integer. Then, a single query to g contains all the information that can be obtained by asking up to $p(n)$ parallel queries to f, since each value $f(x_i)$ can be recovered in polynomial time by computing

$$f(x_i) = \left\lfloor \frac{g(x_1\$x_2\$\ldots\$x_{p(n)})}{B^{i-1}} \right\rfloor \bmod B.$$

The function g is also in #**P**, since this class is closed under summations and products [1], and this proves $\mathbf{P}_{\parallel}^{\#\mathbf{P}} \subseteq \mathbf{P}^{\#\mathbf{P}[1]}$. □

3 Simulating a single #P query monodirectionally

It is quite easy to simulate efficiently (actually, in linear time) a deterministic Turing machine working in polynomial time, and thus using only a tape length, by means of a uniform family of P systems [6]. A configuration of the Turing machine can be encoded as a multiset of objects as follows:



that is, each symbol (including blanks) is subscripted by an index corresponding to the number of the tape cell, and the state of the machine is also represented as an object, subscripted by the index of the cell currently under the tape head. Blank tape cells are represented by the $\Box_i$ objects. Then, each transition of the machine, say $\delta(q, b) = (r, a, d)$ with $d = \pm 1$, is simulated by a set of cooperative evolution rules replicated for each legitimate tape position:

$$[q_i\ b_i \to r_{i+d}\ a_i]_h \qquad \text{for } 0 \le i < s(n) \tag{1}$$

where $s(n)$ is the polynomial space bound of the machine tape. These rules replace the symbol b_i under the tape head by a_i, and update the state symbol q_i to r_{i+d}, which also updates the position of the tape head.

A *nondeterministic* Turing machine can be simulated by dividing elementary membranes, replacing the rules (1) by

$$[q_i\ b_i \to \langle q_i, b_i\rangle]_h \qquad \text{for } 0 \le i < s(n) \tag{2}$$
$$[\langle q_i, b_i\rangle]_h \to [\langle r_{i+d}, a_i\rangle']_h\ [\langle s_{i+e}, c_i\rangle']_h \qquad \text{for } 0 \le i < s(n) \tag{3}$$
$$[\langle r_{i+d}, a_i\rangle' \to r_{i+d}\ c_i]_h \qquad \text{for } 0 \le i < s(n) \tag{4}$$
$$[\langle s_{i+e}, b_i\rangle' \to s_{i+e}\ c_i]_h \qquad \text{for } 0 \le i < s(n) \tag{5}$$

in the case of a nondeterministic transition such as $\delta(q, b) = \{(r, a, d), (s, c, e)\}$. The rules of type (2) "pack" the head-state object and the object representing the

symbol under the tape head into a single object (which is necessary in order to respect the membrane division rule format). The rules of type (3) then perform the transition rule by dividing the membrane and rewriting the packed object into the two objects representing the two possible evolutions of the configuration, one in each of the resulting membranes; these objects are primed, in order to signal that they are the *result* of the transition and not the left-hand side. Finally, the two resulting objects are "unpacked" by the rules of type (4) and (5), thus obtaining the two possible Turing machine configurations inside the divided membranes. In order to avoid synchronisation issues due to the three-step simulation of a nondeterministic transition vs the one-step simulation of deterministic ones, we also slow down the latter accordingly, using the rules

$$\begin{aligned} &[q_i\ b_i \to \langle q_i, b_i\rangle]_h && \text{for } 0 \le i < s(n) \\ &[\langle q_i, b_i\rangle \to \langle r_{i+d}, a_i\rangle']_h && \text{for } 0 \le i < s(n) \\ &[\langle r_{i+d}, a_i\rangle' \to r_{i+d}\ a_i]_h && \text{for } 0 \le i < s(n) \end{aligned}$$

in the case of a deterministic transition such as $\delta(q, b) = (r, a, d)$.

Thus, a single membrane (or a number of membranes obtained by division of a single initial one, in the case of nondeterminism) can efficiently simulate a polynomial-size tape Turing machine and, in particular, a Turing machine working in polynomial time. On the other hand, by using several nested membranes it is possible to efficiently simulate oracle queries [6]. With bidirectional P systems (i.e., standard P systems using both send-in and send-out rules) the simulation of the Turing machine is paused, then one usually sends the query string into a child membrane, where another Turing machine for the oracle language is simulated, possibly using membrane division; the answer is sent out and the simulation of the original Turing machine is resumed.

With monodirectional P systems we proceed in the opposite direction: we first duplicate and send out the multiset encoding the configuration of the Turing machine being simulated, then the oracle machine is simulated in the innermost membrane, and the result is sent out, where the simulation of the original Turing machine can then resume [6]. This process is depicted in Figure 1.

When the simulated Turing machine answering the query is nondeterministic, several result-objects yes are sent out from the divided membranes; they can be counted and operated upon by the Turing machine simulated in the external membrane by converting them in the binary representation of their multiplicity. This can be accomplished by using cooperative evolution rules as follows:

$$\begin{aligned} &[\mathsf{yes} \to 1_0]_k && \\ &[1_i\ 1_i \to 1_{i+1}]_k && \text{for } 0 \le i < m \end{aligned}$$

where m is the maximum number of bits for the answer, which can be computed as in the proof of Proposition 1. These rules produce the multiset of 1_i for all positions i where the number of yes objects produced as the answer to the query

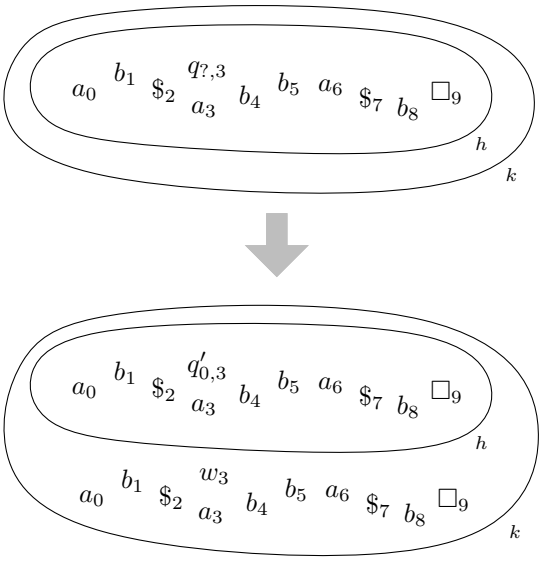


Fig. 1. Simulating an oracle query by means of monodirectional P systems. The portion of configuration delimited by \$ corresponds to the oracle query string, and the tape head is located on its first symbol. When the Turing machine being simulated inside membrane h enters the query state $q_?$, its configuration is duplicated and sent out. The head-state object inside membrane h now represents the state q'_0, the initial state of the Turing machine to be simulated in order to answer the oracle query, while the head-state object outside (in membrane k) represent a "dummy" symbol waiting for state w.

expressed in binary notation, contains a 1. By combining this with the multiset $0_m\ 0_{m-1}\ \cdots\ 0_1\ 0_0$ using the rules

$$[0_i\ 1_i \to 1_i]_k \qquad\qquad \text{for } 0 \leq i \leq m$$

i.e., by deleting the objects 0_i corresponding to the existing objects 1_i, we obtain the binary notation for the answer to the query, which can then be processed by the original Turing machine (now being simulated inside membrane k) as part of its tape.

The existence of the simulation described here, together with Proposition 1, prove the following result:

Theorem 1. *A deterministic polynomial time Turing machine with a polynomial number of queries to a* #**P** *problem can be simulated by shallow chargeless monodirectional P systems with active membranes and minimal cooperation rules in polynomial time.* □

4 Simulating P systems with a single #P query

The query used by the Turing machine that simulates a shallow chargeless P system with active membranes and minimal cooperation, working in polynomial time, is the following:

Query 1. *Given the description of a P system Π, an elementary membrane label h, an object type a and a time t in unary notation, how many objects of type a are collectively sent out by membranes with label h at time t?*

It is now necessary to prove that the answer to the query can be actually computed by a function in #**P**, i.e., it is a "valid" oracle query for the Turing machines that we are considering. Here we only give a sketch of the proof; all the details can be found in [5].

Lemma 1. *Query 1 is in #**P**.*

Proof. Query 1 can be answered by essentially simulating a single-membrane P system; indeed, if Π is monodirectional, no object can enter membrane h from the parent membrane, and if h is elementary, neither can objects from children membranes. By the Milano Theorem [11], a P system without division (thus, in particular, a single membrane without division), even with cooperative evolution rules, can be simulated in deterministic polynomial time. By allowing nondeterminism, the divisions $[a]_h \to [b]_h\ [c]_h$ of membrane h can be simulated in polynomial time by nondeterministically choosing whether to simulate the "left" (where a is rewritten as b) or the "right" membrane (where a is rewritten as c) resulting from the division. After simulating t steps, this results in a nondeterministic computation tree with a leaf for each instance of membrane h. Each computation must accept if and only if an object of type a is sent out at time t, which gives us a number of accepting computations identical to the number of objects a that are collectively sent out at time t by membranes labelled by h, proving that the query is in #**P**. □

Since we have shown that the query is actually computable by a function in #**P**, we are now ready to prove the main theorem of this section:

Theorem 2. *A family of (semi)uniform shallow chargeless monodirectional P systems with active membranes running in polynomial time can be efficiently simulated with a single #**P** query.*

Proof. Given an input string x, the corresponding P system Π_x can be constructed in polynomial time by a deterministic Turing machine M that simulates the (two) Turing machine(s) establishing the (semi)uniformity condition.

Before beginning the actual simulation of Π_x, we can ask a number of queries to an oracle for Query 1; in particular, we ask Query 1 for each possible value of h (labels of elementary membranes), of a (symbols of the alphabet of Π_x), and of t (time steps between 0 and $p(|x|)$, where p is the polynomial running time of the

family). Clearly, this is a polynomial number of parallel queries. These queries can then be combined into a single #**P** query by means of Proposition 1.

Then, the external membrane of Π_x can be simulated by Turing machine M. Since this membrane does not divide, by the Milano Theorem [11] it can be simulated deterministically in polynomial time, except for the objects coming from the children membranes, which are allowed to divide. But these have already been precomputed by asking the oracle queries, and can simply be added with the corresponding multiplicity in the correct time step to the configuration of the outermost membrane. The simulation can then be carried out correctly until the result object is sent out to the environment, and the simulation algorithm accepts or rejects correspondingly. □

5 Conclusions

We have shown that minimal cooperation for monodirectional, shallow P systems with active membranes without charges is sufficient to reach and characterize $\mathbf{P}_{\parallel}^{\#\mathbf{P}}$. This minimal amount of cooperation seems actually necessary and it might be expressed either explicitly (as done here), or implicitly (as with antimatter [5]). However, the minimal amount of cooperation actually required to "count", thus allowing the construction of a #**P** oracle, is still an open research avenue. In fact, while minimal cooperation can simulate annihilation rules in P systems with antimatter, it is unclear if there exist ways of performing cooperative actions that are even weaker, while still attaining the ability to "count" and perform #**P** queries.

References

1. Fortnow, L.: Counting complexity. In: Hemaspaandra, L.A., Selman, A.L. (eds.) Complexity Theory Retrospective II, pp. 81–107. Springer (1997)
2. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Simulating elementary active membranes, with an application to the P conjecture. In: Gheorghe, M., Rozenberg, G., Sosík, P., Zandron, C. (eds.) Membrane Computing, 15th International Conference, CMC 2014. Lecture Notes in Computer Science, vol. 8961, pp. 284–299. Springer (2014), https://doi.org/10.1007/978-3-319-14370-5_18
3. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Membrane division, oracles, and the counting hierarchy. Fundamenta Informaticae **138**(1–2), 97–111 (2015), https://doi.org/10.3233/FI-2015-1201
4. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Monodirectional P systems. Natural Computing **15**(4), 551–564 (2016), https://doi.org/10.1007/s11047-016-9565-2
5. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: The counting power of P systems with antimatter. Theoretical Computer Science **701**, 161–173 (2017), https://doi.org/10.1016/j.tcs.2017.03.045
6. Leporati, A., Manzoni, L., Mauri, G., Porreca, A.E., Zandron, C.: Subroutines in P systems and closure properties of their complexity classes. Theoretical Computer Science (2018), https://doi.org/10.1016/j.tcs.2018.06.012, in press

7. Papadimitriou, C.H.: Computational Complexity. Addison-Wesley (1993)
8. Păun, G.: P systems with active membranes: Attacking NP-complete problems. Journal of Automata, Languages and Combinatorics **6**(1), 75–90 (2001)
9. Sosík, P., Rodríguez-Patón, A.: Membrane computing and complexity theory: A characterization of PSPACE. Journal of Computer and System Sciences **73**(1), 137–152 (2007), https://doi.org/10.1016/j.jcss.2006.10.001
10. Valencia-Cabrera, L., Orellana-Martín, D., Martínez-del-Amor, M.A., Riscos-Núñez, A., Pérez-Jiménez, M.J.: Polarizationless P systems with active membranes: Computational complexity aspects. Journal of Automata, Languages and Combinatorics **21**(1–2), 107–123 (2016), https://doi.org/10.25596/jalc-2016-107
11. Zandron, C., Ferretti, C., Mauri, G.: Solving NP-complete problems using P systems with active membranes. In: Antoniou, I., Calude, C.S., Dinneen, M.J. (eds.) Unconventional Models of Computation, UMC'2K, Proceedings of the Second International Conference, pp. 289–301. Springer (2001), https://doi.org/10.1007/978-1-4471-0313-4_21

A new perspective on computational complexity theory in Membrane Computing

David Orellana-Martín, Luis Valencia-Cabrera,
Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: {dorellana, lvalencia, ariscosn, marper}@us.es

Summary. A single Turing machine can solve decision problems with an infinite number of instances. On the other hand, in the framework of membrane computing, a "solution" to an abstract decision problem consists of a *family* of membrane systems (where each system of the family is associated with a finite set of instances of the problem to be solved). An interesting question is to analyze the possibility to find a single membrane system able to deal with the infinitely many instances of a decision problem.

In this context, it is fundamental to define precisely how the instances of the problem are introduced into the system. In this paper, two different methods are considered: pre-computed (in polynomial time) resources and non-treated resources.

An extended version of this work will be presented in the 20th International Conference on Membrane Computing.

1 Introduction

In the 17th Brainstorming Week on Membrane Computing, an apparently innocent problem was presented by the authors: the `ONLY-ONE-OBJECT` problem. The goal is to build a system able to distinguish whether in a given region, at a given moment, there is only one copy of an object, or if the multiplicity of the object is strictly greater than one. Besides, the notion of efficient solvability by means of a single recognizer polarizationless P system with active membranes, without dissolution rules and using division for elementary and non-elementary membranes, was proposed. Following a reasoning based on the dependency graph technique, a negative answer to the previous question was concluded (i.e. the problem is not solvable in the proposed framework).

In some sense, the previous question links up with others that were proposed by P. Sosík [17], which raise the possibility of being able to solve **P**-complete problems

or **NP**-complete problems by means of a single membrane system. Specifically, two "open problems" were "formulated" in [17], expressed in an informal way as follows:

- **Open Problem 1.** *Is there any known standard model of P system capable of solving a P-complete problem in polynomial time without the use of families, i.e., all instances are solved by the same P systems?*
- **Open Problem 2.** *How to design a natural (not much "extraordinary") model of P system capable of solving an NP-complete problem in polynomial time without the use of families?*

Of course, these questions should be expressed in a formal way and their answers will depend on the definitions given about what *solving a decision problem through a single membrane system* means.

For instance, two possible definitions could be considered according to the way of entering the input inside the membrane system: (a) by using *precomputed* resources (that is, waiting for a polynomial time *prior* to the initial step of the computation, to calculate which is the input multiset that has to be provided to the system); or (b) by directly introducing the input multiset without any preprocessing, that is, *free* of external resources.

For a comprehensive introduction to membrane systems, we refer the reader to [12, 15].

2 The complexity class $\mathbf{PMC}_{\mathcal{R}}^{1p}$

First, let us define a solution to a decision problem through a single membrane system allowing the possibility to use (external) *precomputed* resources for providing the input multiset to the system. In other words, we assume that there is an available device able to execute the function that computes the input multiset, and this process should be performed before the computation of the membrane system starts.

Definition 1. *Let $\mathcal{R}$ be a class of recognizer membrane system. Let $X = (I_X, \theta_X)$ be a decision problem. We say that problem X is solvable in polynomial time by a single membrane system Π from $\mathcal{R}$ with precomputed resources, denoted by $X \in \mathbf{PMC}_{\mathcal{R}}^{1p}$, if the following hold:*

- *There exists a* polynomial encoding *cod from X to Π providing a "reasonable encoding scheme" which maps problem instances into the multisets describing them [3]; that is, there exists a polynomial time computable function, cod, whose domain is I_X such that for every instance $u \in I_X$, $cod(u)$ is a multiset over the input alphabet of Π.*
- *The system Π is* polynomially bounded *with regard to (X, cod); that is, there exists a polynomial $p(r)$ such that for each instance $u \in I_X$, every computation of the system Π with input multiset $cod(u)$ performs at most $p(|u|)$ steps.*

- *The system Π is* sound *with regard to (X, cod); that is, for each instance $u \in I_X$, if there exists an accepting computation of the system Π with input multiset $cod(u)$ then $\theta_X(u) = 1$.*
- *The system Π is* complete *with regard to (X, cod); that is, for each instance $u \in I_X$ such that $\theta_X(u) = 1$, every computation of the system Π with input multiset $cod(u)$ is an accepting computation.*

In this definition, the input multiset that is allocated into the initial configuration of the system is precomputed by means of a polynomial-time computable function.

Proposition 1. *If $\mathcal{R}$ is a class of recognizer membrane systems, then*

$$\mathbf{P} \subseteq \mathbf{PMC}^{1p}_{\mathcal{R}} \subseteq \mathbf{PMC}_{\mathcal{R}}$$

Proof. In order to show that $\mathbf{P} \subseteq \mathbf{PMC}^{1p}_{\mathcal{R}}$, let $X = (I_X, \theta_X)$ be a decision problem in class $\mathbf{P}$. Let us consider the deterministic recognizer (cell-like) membrane system $\Pi = \{\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{R}, i_{in}\}$ of degree 1 defined as follows:

- $\Gamma = \Sigma = \{\mathtt{yes}, \mathtt{no}\}$.
- $\mu = [\ ]_1$.
- $\mathcal{M}_1 = \emptyset$
- $\mathcal{R} = \{[\,\mathtt{yes}\,]_1 \to \mathtt{yes}\ [\ ]_1;\ [\,\mathtt{no}\,]_1 \to \mathtt{no}\ [\ ]_1\}$
- $i_{in} = 1$.

Let us consider *cod* as the map whose domain is I_X defined as follows: for every $u \in I_X$, $cod(u) = \{\mathtt{yes}\}$ if $\theta_X(u) = 1$, and $cod(u) = \{\mathtt{no}\}$, otherwise. Since $X \in \mathbf{P}$, *cod* is a polynomial-time function. Then, we have:

- The system Π is *polynomially bounded* with regard to (X, cod): for every instance $u \in I_X$, the computation of Π with input multiset $cod(u)$ performs 1 transition step.
- For every instance $u \in I_X$, the computation of the system Π with input multiset $cod(u)$ is an accepting computation if and only if $\theta_X(u) = 1$.

This definition can be easily adjusted for any class of recognizer membrane systems $\mathcal{R}$, in such a way that we have $X \in \mathbf{PMC}^{1p}_{\mathcal{R}}$. Then, we conclude that $\mathbf{P} \subseteq \mathbf{PMC}^{1p}_{\mathcal{R}}$.

In order to show that $\mathbf{PMC}^{1p}_{\mathcal{R}} \subseteq \mathbf{PMC}_{\mathcal{R}}$, let $X = (I_X, \theta_X)$ be a decision problem such that $X \in \mathbf{PMC}^{1p}$. Let Π' a membrane system from $\mathcal{R}$ solving X according to Definition 1, being cod' a *polynomial encoding* from X to Π associated with that solution. Let us consider the family $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$ defined as follows $\Pi(t) = \Pi'$, for each $t \in \mathbb{N}$. Let us consider the polynomial encoding (cod, s) from the problem X to the family $\mathbf{\Pi}$ defined as follows: $cod = cod'$ and $s(u) = 0$, for each $u \in I_X$. Then it is easy to check that the family $\mathbf{\Pi}$ is polynomially uniform by Turing machines, polynomially bounded with regard to (X, cod, s), and sound and complete with regard to (X, cod, s). Thus, $X \in \mathbf{PMC}_{\mathcal{R}}$.

□

3 The complexity class $\mathbf{PMC}_{\mathcal{R}}^{\mathbf{1f}}$

The second definition refers to the case in which the input multiset is directly introduced inside the system as it is ("*free*" of external dependencies or resources), and thus the input alphabet should be chosen so that the system is able to "read" the instances of the problem to be solved.

Definition 2. *Let $\mathcal{R}$ be a class of recognizer membrane systems. Let $X = (I_X, \theta_X)$ be a decision problem such that I_X is a language over a finite alphabet Σ_X. We say that problem X is solvable in polynomial time by a single membrane system Π from $\mathcal{R}$ free of external resources, denoted by $X \in \mathbf{PMC}_{\mathcal{R}}^{1f}$, if the following hold:*

- *The input alphabet of Π is Σ_X.*
- *The system Π is* polynomially bounded *with regard to X; that is, there exists a polynomial $p(r)$ such that for each instance $u \in I_X$, every computation of the system Π with input multiset u performs at most $p(|u|)$ steps.*
- *The system Π is* sound *with regard to X; that is, for each instance $u \in I_X$, if there exists an accepting computation of the system Π with input multiset u then $\theta_X(u) = 1$.*
- *The system Π is* complete *with regard to X; that is, for each instance $u \in I_X$ such that $\theta_X(u) = 1$, every computation of the system Π with input multiset u is an accepting computation.*

Proposition 2. *Let $\mathcal{R}$ be a class of recognizer membrane systems. Then we have $\mathbf{PMC}_{\mathcal{R}}^{1f} \subseteq \mathbf{PMC}_{\mathcal{R}}^{1p}$.*

Proof. Let us assume that $X \in \mathbf{PMC}_{\mathcal{R}}^{1f}$. Let Π' a membrane system from $\mathcal{R}$ whose input alphabet is Σ_X (the working alphabet of the problem X) such that it is polynomially bounded, sound and complete with regard to X. Let us consider the polynomial encoding *cod* from X to Π' defined as follows: $cod(u) = u$, for every instance $u \in I_X$. Then, Π' is polynomially bounded, sound and complete with regard to (X, cod). Thus, $X \in \mathbf{PMC}_{\mathcal{R}}^{1p}$.

□

4 Decision problems with a finite number of instances

In this section, we work with decision problems whose set of instances is a finite set.

Proposition 3. *Let $\mathcal{T}(so)$ the class of all recognizer transition P systems which make use of send-out communication rules only. Then, if $X = (I_X, \theta_X)$ is a decision problem whose set of instances is a finite set, then $X \in \mathbf{PMC}_{\mathcal{T}(so)}^{1f}$.*

Proof. Let $X = (I_X, \theta_X)$ be a decision problem whose set of instances I_X is a finite language over the alphabet Σ_X. Let us consider the recognizer transition P system $\Pi = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{R}_1, i_{in})$, defined as follows:

- The working alphabet is $\Gamma = \Sigma_X \cup \{\texttt{yes}, \texttt{no}\}$ and the input alphabet Σ is Σ_X.
- The membrane structure is $\mu = [\ ]_1$ and the initial multiset is $\mathcal{M}_1 = \emptyset$.
- The set $\mathcal{R}_1$ of rules is

$$\{[\,u\,]_1 \to \texttt{yes}\ [\ ]_1 \mid \theta_X(u) = 1\} \cup \{[\,u\,]_1 \to \texttt{no}\ [\ ]_1 \mid \theta_X(u) = 0\}$$

- The input membrane is labelled by 1.

Obviously, membrane system Π belongs to the class $\mathcal{T}(so)$ and it solves problem X, according to Definition 2.

5 The NONE-OBJECT problem

In this section, we consider the NONE-OBJECT problem which informally corresponds to the task of determining whether there is any input object or not in the system. Formally, let $X = (I_X, \theta_X)$ be the decision problem defined as follows:

$$I_X = \{\emptyset\} \cup \{a^n \mid n \in \mathbb{N}, n \geq 1\}\ ,\ \theta_X(\emptyset) = 1, \text{and } \theta_X(a^n) = 0 \text{ for each } n \geq 1$$

That is, the problem X distinguishes two types of situations: absence of objects on one hand, and at least one copy of object a, on the other hand.

Theorem 1. *Let $\mathcal{T}(nc, ev, so, dis, pr)$ the class of all non-cooperative recognizer P systems which makes use of minimal production in object evolution rules (that is, only one object in the right-hand side of the rule), send-out communication rules, dissolution rules and priorities. Then,* NONE-OBJECT$\in \mathbf{PMC}^{1f}_{\mathcal{T}(nc,ev,so,dis,pr)}$.

Proof. Let us consider the system Π from $\mathcal{T}(nc, ev, so, dis, pr)$ defined as follows:

- The working alphabet is $\Gamma = \{a, b, c\}$ and the input alphabet is $\Sigma = \{a\}$.
- The membrane structure μ is $\mu = [\ [\ ]_2\ ]_1$ and the initial multisets are $\mathcal{M}_1 = \emptyset$ and $\mathcal{M}_2 = \{c\}$.
- The set $\mathcal{R}$ of rules of Π is the following:

$$\{[\ a \to b]_2;\ [\ b\ ]_2 \to \texttt{no};\ [\ c\ ]_2 \to \texttt{yes};\ [\ \texttt{yes}]_1 \to \texttt{yes}\,[\ \ ]_1;\ [\ \texttt{no}]_1 \to \texttt{no}\,[\ \ ]_1\}$$

- The set of priorities $\mathcal{P}$ among rules of Π is the following:

$$\big\{([\ a \to b]_2, [\ c\ ]_2 \to \texttt{yes});\ ([\ b\ ]_2 \to \texttt{no}, [\ c\ ]_2 \to \texttt{yes})\big\}$$

- The input membrane is labelled by 2.

Then, the following hold:

- For each natural number $n \geq 1$, the system Π with input multiset $\{a^n\}$ is deterministic, the computation of $\Pi + \{a^n\}$ performs three transition steps and it is a rejecting computation.
- The system Π with input multiset $\emptyset$ is deterministic, the computation of $\Pi + \emptyset$ performs two transition steps and it is an accepting computation.

Thus, NONE-OBJECT$\in \mathbf{PMC}^{1f}_{\mathcal{T}(nc,ev,so,dis,pr)}$. □

6 The ONLY-ONE-OBJECT problem

In this section, the problem of telling apart "one" from "more-than-one" object is considered. Formally, let $X = (I_X, \theta_X)$ be the decision problem defined as follows:

$$I_X = \{a^n \mid n \in \mathbb{N}, n \geq 1\} \text{ and } \theta_X(a^n) = 1 \text{ if and only if } n = 1$$

That is, the problem X distinguishes the case when there is only one copy of object a from the rest of possible cases with several copies of that object. We denote that problem as the ONLY-ONE-OBJECT problem. Obviously, the ONLY-ONE-OBJECT problem belongs to class $\mathbf{P}$ since it is easy to design a deterministic Turing machine solving that problem which takes two computation steps. Thus, ONLY-ONE-OBJECT$\in \mathbf{P}$. Bearing in mind that for every class $\mathcal{R}$ of recognizer membrane systems, we have we $\mathbf{P} \subseteq \mathbf{PMC}^{1p}_{\mathcal{R}}$, we deduce that ONLY-ONE-OBJECT$\in \mathbf{PMC}^{1p}_{\mathcal{R}}$.

It is easy to prove that ONLY-ONE-OBJECT$\in \mathbf{PMC}^{1f}_{\mathcal{T}(nc,ev,so,dis,pr)}$, but the following result shows that this problem cannot be solved by a membrane system from $\mathcal{AM}^0(-d,+ne)$ without using precomputed resources, being $\mathcal{AM}^0(-d,+ne)$ the class of polarizationless P systems without dissolution rules and with division rules for elementary and non-elementary membranes.

Theorem 2. *There does not exist a recognizer membrane system $\Pi' \in \mathcal{AM}^0(-d,+ne)$ solving the* ONLY-ONE-OBJECT *problem in a polynomial time by a single membrane system and free of resources. That is,* ONLY-ONE-OBJECT$\notin \mathbf{PMC}^{1f}_{\mathcal{AM}^0(-d,+ne)}$.

Proof. (Reasoning by *reductio ad absurdum*) Let us assume that there exists a recognizer membrane system Π' from $\mathcal{AM}^0(-d,+ne)$ verifying the following:

(a) The input alphabet of Π' is the singleton $\{a\}$.
(b) Every computation of Π' with input multiset $\{a\}$ is an accepting computation.
(c) Every computation of Π' with input multiset $\{a^n\}$, for each $n > 1$, is a rejecting computation.

Let us denote by $G_{\Pi'+\{a\}}$ (respectively, $G_{\Pi'+\{a^n\}}$, for each $n > 1$) the *dependency graph*[1] associated with the system $\Pi' + \{a\}$ (resp. $\Pi' + \{a^n\}$). Then, we have:

- For each $n > 1$, $G_{\Pi'+\{a\}} = G_{\Pi'+\{a^n\}}$. Indeed, in both graphs there is only one edge starting from s, specifically, the edge $\{s, (a, i_{in})\}$, and the rest of edges are given by the rules of Π', due to $\Pi' \in \mathcal{AM}^0(-d,+ne)$.
- A computation of $\Pi' + \{a\}$ is an accepting computation if and only if there exists a path in $G_{\Pi'+\{a\}}$ from s to $(\texttt{yes}, env)$.
- For each $n > 1$, a computation of $\Pi' + \{a^n\}$ is an accepting computation if and only if there exists a path in $G_{\Pi'+\{a^n\}}$ from s to $(\texttt{yes}, env)$.

[1] We will not recall the formal definition here (see [2, 18] for details). The dependency graph can be intuitively seen as a map of "reactants-product" relationship between objects: the nodes are pairs $(object, region)$ and for every rule of the system there will be an arc connecting each object on the left-hand-side to each object on the right-hand side.

Thus, bearing in mind that $G_{\Pi'+\{a\}} = G_{\Pi'+\{a^n\}}$ we deduce that every computation of $\Pi' + \{a\}$ is an accepting computation if and only if every computation of $\Pi' + \{a^n\}$, for each $n > 1$, is an accepting computation. Hence, conditions (b) and (c) are contradictory.

□

Corollary 1. $\mathbf{PMC}^{1f}_{\mathcal{AM}^0(-d,+ne)} \subsetneq \mathbf{P} \subseteq \mathbf{PMC}^{1p}_{\mathcal{AM}^0(-d,+ne)}$.

7 A version of the PARITY problem

In this section, a version of the PARITY problem is considered. Specifically, let $\texttt{PARITY} = (I_{\texttt{PARITY}}, \theta_{\texttt{PARITY}})$ be the decision problem defined as follows:

$$I_{\texttt{PARITY}} = \{a^n \mid n \in \mathbb{N}, n \geq 1\} \text{ and } \theta_{\texttt{PARITY}}(a^n) = 1 \text{ if and only if } n \text{ is even}$$

That is, the PARITY problem distinguishes an even number of copies of object a from an odd number of copies of that object. Obviously, this version of the PARITY problem belongs to class **P** since it is easy to design a deterministic Turing machine solving that problem.

Theorem 3. *Let $\mathcal{T}(mcmp, so, dis, pr)$ the class of all recognizer P systems which make use of minimal cooperation and minimal production in object evolution rules, send-out communication rules, dissolution rules and priorities. Then,* $\texttt{PARITY} \in \mathbf{PMC}^{1f}_{\mathcal{T}(mcmp,so,dis,pr)}$.

Proof. Let us consider the system Π from $\mathcal{T}(mcmp, so, dis, pr)$ defined as follows:

(a) The working alphabet is $\Gamma = \{a, b\}$ and the input alphabet is $\Sigma = \{a\}$.
(b) The membrane structure is $\mu = [\ [\ ]_2\ ]_1$, and the initial multisets are $\mathcal{M}_1 = \emptyset$ and $\mathcal{M}_2 = \emptyset$.
(d) The set $\mathcal{R}$ of rules of Π is the following:

$$\{[\ a^2 \to b]_2;\ [\ b^2 \to b]_2;\ [\ a]_2 \to \texttt{no};\ [\ b]_2 \to \texttt{yes}\} \cup$$
$$\{[\ \texttt{no}]_1 \to \texttt{no}\,[\ ]_1;\ [\ \texttt{yes}]_1 \to \texttt{yes}\,[\ ]_1\}$$

(e) The set of priorities $\mathcal{P}$ among rules of Π is the following:

$$\big\{([\ a^2 \to b]_2, [\ a]_2 \to \texttt{no});\ ([\ b^2 \to b]_2, [\ a]_2 \to \texttt{no});\ ([\ a^2 \to b]_2, [\ b]_2 \to \texttt{yes});$$
$$([\ b^2 \to b]_2, [\ b]_2 \to \texttt{yes});\ ([\ a]_2 \to \texttt{no}, [\ b]_2 \to \texttt{yes})\big\}$$

(f) The input membrane is labelled by 2.

Then, for each natural number $n \geq 1$, the following hold:

- The system Π with input multiset $\{a^n\}$ is deterministic.
- The computation of $\Pi + \{a^n\}$ performs $2 + \lfloor \log_2(n) \rfloor$ transition steps.
- The natural number n is odd if and only if the configuration $\mathcal{C}_{\lfloor \log_2(n) \rfloor}$ contains a copy of object a.
- The natural number n is even if and only if the computation of $\Pi + \{a^n\}$ is an accepting computation.

Thus, $\texttt{PARITY} \in \mathbf{PMC}^{1f}_{\mathcal{T}(mcmp,so,dis,pr)}$. □

8 Conclusions

In this work, the ability of solving problems by single "stand-alone" membrane systems instead of families of membrane systems is studied. While using precomputed resources, it is easy to see that problems from **P** can be solved by a single membrane system using only send-out rules. A question arises from here: What if we cannot access to a precomputed encoding and we have the raw instance as input? In this paper, the power of single membrane systems free of precomputed resources is also studied, giving, on the one hand, solutions to decision problems by means of a single membrane system solving them, and on the other hand demonstrating the inability of systems from $\mathcal{AM}^0(-d, +ne)$ to solve the `ONLY-ONE-OBJECT` problem by using the dependency graph technique in a novel way.

While talking about recognizer membrane systems, we suppose that they can, at least, send an object to the environment to return the answer. Even with this minimal definition, the lower bound for $\mathbf{PMC}^{1p}_{\mathcal{R}}$ has been demonstrated to be **P**. On the other hand, logic gates have been solved by a system using only non-cooperative send-out rules. This result gives a tool to tackle problems below **P** in the framework of Membrane Computing.

An interesting question is to obtain a lower bound of these systems using only unary alphabets; that is, not allowing cooperation implicit in the instance of the problem. It could be also worth investigating other "weaker" variants, for example obtained removing priorities.

Some open problems remain for future work, e.g. looking for upper bounds of the complexity classes $\mathbf{PMC}^{1p}_{\mathcal{R}}$ and $\mathbf{PMC}^{1f}_{\mathcal{R}}$.

Acknowledgements

This work was supported in part by the research project TIN2017-89842-P, co-financed by Ministerio de Economía, Industria y Competitividad (MINECO) of Spain, through the Agencia Estatal de Investigación (AEI), and by Fondo Europeo de Desarrollo Regional (FEDER) of the European Union.

References

1. A. Alhazov, T.-O. Ishdorj: Membrane operations in P systems with active membranes. In Proceedings of the Second Brainstorming Week on Membrane Computing, Sevilla, 2-7 February 2004, 37-44.
2. A. Cordón-Franco, M. A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez. Weak metrics on configurations of a P system. In Gh. Paun, A. Riscos, Á. Romero, F. Sancho (eds.) *Proceedings of the Second Brainstorming Week on Membrane Computing*, Report RGNC 01/2004, 2004, pp. 139-151.

3. M.R. Garey, D.S. Johnson D.S. *Computers and intractability*, W.H. Freeman and Company, New York, 1979.
4. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F.J. Romero-Campero: A linear solution of Subset Sum problem by using membrane creation. *Lecture Notes in Computer Science*, **3561** (2005), 258-267.
5. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero, A. Romero-Jiménez: Characterizing tractability by cell–like membrane systems. In K.G. Subramanian, K. Rangarajan, M. Mukund (eds.) *Formal models, languages and applications*, World Scientific, Singapore, 2006, pp. 137–154.
6. M.A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F.J. Romero-Campero: A linear time solution for QSAT with membrane creation. *Lecture Notes in Computer Science*, **3850** (2006), 241-252.
7. P.L. Luisi. The Chemical Implementation of Autopoiesis, *Self-Production of Supramolecular Structures* (G.R. Fleishaker et al., eds.), Kluwer, Dordrecht, 1994.
8. C. Martín-Vide, Gh. Păun, A. Rodríguez-Patón. On P Systems with Membrane Creation. *Computer Science Journal of Moldova*, **9**, 2(26) 2001, 134-145.
9. M. Mutyam, K. Krithivasan: P systems with membrane creation: Universality and efficiency. *Lecture Notes in Computer Science*, **2055** (2001), 276–287.
10. L. Pan, T.-O. Ishdorj: P systems with active membranes and separation rules. *Journal of Universal Computer Science*, **10**, 5 (2004), 630–649.
11. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998
12. Gh. Păun. Membrane Computing: An introduction. *Springer Natural Computing Series*, 2002.
13. Gh. Păun. P systems with active membranes: attacking **NP**–complete problems, *Journal of Automata, Languages and Combinatorics*, **6**, 1 (2001), 75-90.
14. Gh. Păun, M.J. Pérez-Jiménez, Gr. Rozenberg. Spike trains in spiking neural P systems. *International Journal of Foundations of Computer Science*, **17**, 4 (2006), 975-1002.
15. Gh. Păun, G. Rozenberg, A. Salomaa (eds). The Oxford Handbook of Membrane Computing. *Oxford University Press*, Oxford, U.K., 2009.
16. M.J. Pérez-Jiménez, A. Romero-Jiménez, F. Sancho-Caparrini: Complexity classes in cellular computing with membranes. *Natural Computing*, **2**, 3 (2003), 265–285.
17. P. Sosík. Active Membranes, Proteins on Membranes, Tissue P Systems: Complexity-Related Issues and Challenges. *Lecture Notes in Computer Science*, **8340** (2014), 40–55.
18. L. Valencia-Cabrera, D. Orellana-Martín, I. Pérez-Hurtado, M.J. Pérez-Jiménez. Dependency graph technique revisited. *in this volume*
19. G. Zhang, M.J. Pérez-Jiménez, M. Gheorghe. *Real-Life Modelling with Membrane Computing*. Series: Emergence, Complexity and Computation, Volume 25. Springer International Publishing, 2017, X + 367 pages.

An apparently innocent problem in Membrane Computing

David Orellana-Martín, Luis Valencia-Cabrera,
Agustín Riscos-Núñez, Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: {dorellana, lvalencia, ariscosn, marper}@us.es

Summary. The search for efficient solutions of computationally hard problems by means of families of membrane systems has lead to a wide and prosperous field of research. The study of computational complexity theory in *Membrane Computing* is mainly based on the look for frontiers of efficiency between different classes of membrane systems. Every frontier provides a powerful tool for tackling the **P** *versus* **NP** problem in the following way. Given two classes of recognizer membrane systems $\mathcal{R}_1$ and $\mathcal{R}_2$, being systems from $\mathcal{R}_1$ non-efficient (that is, capable of solving only problems from the class **P**) and systems from $\mathcal{R}_2$ presumably efficient (that is, capable of solving **NP**-complete problems), and $\mathcal{R}_2$ the same class that $\mathcal{R}_1$ with some ingredients added, passing from $\mathcal{R}_1$ to $\mathcal{R}_2$ is comparable to passing from the non efficiency to the presumed efficiency. In order to prove that $\mathbf{P} = \mathbf{NP}$, it would be enough to, given a solution of an **NP**-complete problem by means of a family of recognizer membrane systems from $\mathcal{R}_2$, try to remove the added ingredients to $\mathcal{R}_2$ from $\mathcal{R}_1$. In this paper, we study if it is possible to solve SAT by means of a family of recognizer P systems from $\mathcal{AM}^0(-d, +n)$, whose non-efficiency was demonstrated already.

Key words: Membrane Computing, polarizationless P systems with active membranes, cooperative rules, the **P** versus **NP** problem, SAT problem.

1 Introduction

Membrane Computing is a bio-inspired computing model based on the behavior and the structure of living cells. Introduced by Gh. Păun in 1998 [4], it has been used in a wide range of applications, and several variants have been developed depending on the field of study. From the beginning, the research of computational complexity issues from the perspective of membrane systems has been a prosperous field of study, with several papers written and interesting results found. The

first "solution" to an **NP**-complete problem, the SAT problem, in linear time is presented in [5]. We say "solution" since there was no definition of *solving a problem* by means of membrane systems. There was no definition until [8] where *recognizer* membrane systems (called *decision* membrane systems in the paper, and *accepting* membrane systems in a later paper), for a family of membrane systems of a certain type capable of *solving* a computational problem.

In the following years, for demonstrating the non-efficiency of membrane systems (that is, the capability of only solving problems from the class **P**), some tools were developed, as the *simulation technique* [7], the *algorithmic technique* [] and the *dependency graph technique* [1]. By using the former, in [1] it was demonstrated that P systems from $\mathbf{PMC}_{\mathcal{AM}^0(-d,+n)}$ were capable of solving only problems from the class **P**. As complexity classes $\mathbf{PMC}_{\mathcal{R}}$, being $\mathcal{R}$ a class of recognizer membrane systems, was demonstrated to be closed under polynomial reduction [], finding an efficient solution to *any* **NP**-complete problem by means of a family of P systems from $\mathcal{AM}^0(-d,+n)$ would lead to a negative answer to the $\mathbf{P} \neq \mathbf{NP}$ conjecture; that is, it solving an **NP**-complete problem in this framework leads to $\mathbf{P} = \mathbf{NP}$. It seems interesting then trying to find a solution based on the most common techniques while solving a computationally hard problem.

From here, the paper is organized as follows: in the next section, a brief view to the general structure of techniques to solve **NP**-complete problems by membrane systems is given. Section 3 is devoted to present a "solution" to the SAT problem, such that it depends of the existence of some *special machines*. These machines are detailed in the following two sections, explaining the structure in Section 4 and the behavior in Section 5. After that, in Section 6, the three kinds of special machines introduced are *reduced* to a single one, capable of solving each of the problems of the previous machines. Last, the main result is presented in Section 7. The paper ends with some conclusions and interesting open research lines.

2 Solutions to NP-complete problems

In the framework of Membrane Computing, several efficient solutions to computationally hard problems have been provided by means of a family of membrane systems; that is, they are solutions that run in polynomial time with respect of the size of the input. Usually, this is done by interchanging time and space, in the sense that we need to create an exponential workspace in terms of membranes or cells in the computation in order to obtain all the possible alternatives to solve the instance, and taking advantage of the inherent parallelism of membrane systems to check them at the same time. For this purpose, a family of membrane systems must be defined, each of its systems solving a subset of all the instances of the problem. Usually, the protocol to solve computationally hard problems is the following one:

1. *Generation stage*: In this stage, using division rules [6], separation rules [3] or membrane creation rules [2], among others, we can obtain an exponential workspace in terms of membranes or cells in polynomial (or even linear time).
2. *Checking stage*: In this stage, the presumed solutions in the previous stage are checked in order to know if any of them is a real solution of the instance.
3. *Output stage*: This stage consists in sending an object `yes` or an object `no` to the environment depending on the solvability or not of the instance.

In this sense, an interesting work for the reader is [9], where solutions are analyzed by decomposing the solutions in subroutines.

3 A "solution" to the SAT problem without using dissolution

Here we provide a solution to the `SAT` problem by means of a family of recognizer P systems with active membranes $\mathbf{\Pi} = \{\Pi(t) \mid t \in \mathbb{N}\}$ from $\mathcal{AM}^0(-d, +n)$ with a special mechanism whose behavior will be explained later. Given a Boolean formula φ in CNF and simplified with n variables and p clauses, the system $\Pi(s(\varphi)) + cod(\varphi)$ processes it, being $s(\varphi) = \langle n, p\rangle = \frac{(n+p)(n+p+1)}{2} + n$ and $cod(\varphi) = \{x_{i,j,0} \mid x_i \in C_j\} \cup \{\overline{x}_{i,j,0} \mid \neg x_i \in C_j\}$.

For each $n, p \in \mathbb{N}$, we consider the recognizer P system

$$\Pi(\langle n, p\rangle) = (\Gamma, \Sigma, \mu, \mathcal{M}_1, \mathcal{M}_2, \mathcal{M}_3, multisets(M_{i,j})(1 \leq i \leq n, 1 \leq j \leq p),$$
$$multisets(M_{i,j,l})(1 \leq i \leq \lceil \tfrac{n}{2^l} \rceil, 1 \leq j \leq p, 1 \leq l \leq \lceil \log_2 n \rceil),$$
$$multisets(M_j)(1 \leq j \leq p), multisets(M_{d,l})(1 \leq l \leq \lceil \log_2 n \rceil),$$
$$multisets(M_r), \mathcal{R}, i_{in}, i_{out}),$$

from $\mathcal{AM}^0(-d, +n)$ where:

1. Working alphabet Γ:
   $\{\mathtt{yes}, \mathtt{no}, a, a'\} \cup \{a_{i,j} \mid 1 \leq i \leq n, 1 \leq j \leq 2i-1\} \cup$
   $\{t_{i,j}, f_{i,j} \mid 1 \leq i \leq n, 2i \leq j \leq 2n-1\} \cup$
   $\{T_{i,j}, F_{i,j} \mid 1 \leq i \leq n, 0 \leq j \leq n-1\} \cup \{T_i, F_i, t_i, f_i \mid 1 \leq i \leq n\} \cup$
   $\{x_{i,j,k}, \overline{x}_{i,j,k} \mid 1 \leq i \leq n, 1 \leq j \leq p, 1 \leq k \leq 3n\} \cup$
   $\{c_{i,j,l} \mid 1 \leq i \leq \frac{2n}{2^l}, 1 \leq j \leq p, 1 \leq l \leq \lceil \log_2 n \rceil\} \cup$
   $\{c_{j,l} \mid 1 \leq j \leq p, 1 \leq l \leq (j-1)(k+2)+1\} \cup \{d_j \mid 1 \leq j \leq p\} \cup$
   $\{d_{p,l} \mid 1 \leq l \leq \lceil \log_2 n \rceil + 1\} \cup alphabet(M_{i,j})(1 \leq i \leq n, 1 \leq j \leq p) \cup$
   $alphabet(M_{i,j,l})(1 \leq i \leq \lceil \frac{n}{2^l} \rceil, 1 \leq j \leq p, 1 \leq l \leq \lceil \log_2 n \rceil) \cup$
   $alphabet(M_j)(1 \leq j \leq p) \cup alphabet(M_{d,l})(1 \leq l \leq \lceil \log_2 n \rceil) \cup alphabet(M_r)$
2. Input alphabet Σ:
   $\{x_{i,j,0}, \overline{x}_{i,j,0} \mid 1 \leq i \leq n, 1 \leq j \leq p\}$
3. Initial multisets:
   $\mathcal{M}_1 = \emptyset, \mathcal{M}_2 = \emptyset, \mathcal{M}_3 = \{a_{i,1} \mid 1 \leq i \leq n\}$
4. The rule set $\mathcal{R}$ consists on the following rules:

1.1 Rules to create p copies of every possible truth assignment in each of the 2^n membranes labelled by 2.

$$[\,a_{i,j} \to a_{i,j+1}\,]_3 \quad \text{for } 2 \le i \le n, 1 \le j \le 2i-2$$

$$[\,a_{i,2i-1}\,]_3 \to [\,t_{i,2i}\,]_3\,[\,f_{i,2i}\,]_3 \quad \text{for } 1 \le i \le n$$

$$[\,a_{n,2n-1}\,]_3 \to [\,T_{n,n+1}\,]_3\,[\,F_{n,n+1}\,]_3$$

$$[\,[\quad]_3\,[\quad]_3\,]_2 \to [\,[\quad]_3\,]_2\,[\,[\quad]_3\,]_2$$

$$\left.\begin{array}{l}[\,t_{i,j} \to t_{i,j+1}\,]_3\\ [\,f_{i,j} \to f_{i,j+1}\,]_3\end{array}\right\} \text{for } 1 \le i \le n-1, 2i \le j \le 2n-2$$

$$\left.\begin{array}{l}[\,t_{i,2n-1} \to T_{i,i+1}\,]_3\\ [\,f_{i,2n-1} \to F_{i,i+1}\,]_3\end{array}\right\} \text{for } 1 \le i \le n-1$$

$$\left.\begin{array}{l}[\,T_{i,j} \to T_{i,j-1}\,]_3\\ [\,F_{i,j} \to F_{i,j-1}\,]_3\end{array}\right\} \text{for } 1 \le i \le n, 1 \le j \le i+1$$

$$\left.\begin{array}{l}[\,T_{i,0}\,]_3 \to T_{i,i}\,[\quad]_3\\ [\,F_{i,0}\,]_3 \to F_{i,i}\,[\quad]_3\end{array}\right\} \text{for } 1 \le i \le n$$

$$[\,T_{n,0}\,]_3 \to T_n\,[\quad]_3$$

$$[\,F_{n,0}\,]_3 \to F_n\,[\quad]_3$$

$$\left.\begin{array}{l}[\,T_{i,j} \to T_{i,j+1}\,]_2\\ [\,F_{i,j} \to F_{i,j+1}\,]_2\end{array}\right\} \text{for } 1 \le i \le n-2, i \le j \le n-2$$

$$\left.\begin{array}{l}[\,T_{i,n-1} \to T_i\,]_2\\ [\,F_{i,n-1} \to F_i\,]_2\end{array}\right\} \text{for } 1 \le i \le n-1$$

$$\left.\begin{array}{l}[\,T_i \to t_i^p\,]_2\\ [\,F_i \to f_i^p\,]_2\end{array}\right\} \text{for } 1 \le i \le n-1$$

2.1 Rules to check which clauses are satisfied by the truth assignments.

$$\left.\begin{array}{l}[\,x_{i,j,k} \to x_{i,j,k+1}\,]_2\\ [\,\overline{x}_{i,j,k} \to \overline{x}_{i,j,k+1}\,]_2\end{array}\right\} \text{for } 1 \le i \le n, 1 \le j \le p, 0 \le k \le 3n-1$$

$$\left.\begin{array}{l}x_{i,j,3n}\,[\quad]_{M_{i,j}} \to [\,a'\,]_{M_{i,j}}\\ \overline{x}_{i,j,3n}\,[\quad]_{M_{i,j}} \to [\,a'\,]_{M_{i,j}}\end{array}\right\} \text{for } 1 \le i \le n, 1 \le j \le p$$

$$[\,a' \to a\,]_{M_{i,j}} \quad \text{for } 1 \le i \le n, 1 \le j \le p$$

$$\left.\begin{array}{l}t_i\,[\quad]_{M_{i,j}} \to [\,a\,]_{M_{i,j}}\\ f_i\,[\quad]_{M_{i,j}} \to [\,a\,]_{M_{i,j}}\end{array}\right\} \text{for } 1 \le i \le n, 1 \le j \le p$$

$$M_{i,j}(3n+2) : a^2 \rightsquigarrow c_{i,j,1} \quad \text{in } k \text{ steps for } 1 \le i \le n, 1 \le j \le p$$

2.2 Rules to obtain only one copy of each object c_j, if possible.

$$\left.\begin{array}{l} c_{2i-1,j,l}\,[\quad]_{M_{i,j,l}} \to [\,a'\,]_{M_{i,j,l}} \\ c_{2i,j,l}\,[\quad]_{M_{i,j,l}} \to [\,a\,]_{M_{i,j,l}} \end{array}\right\} \text{ for } \begin{array}{l} 1 \le i \le \lceil \frac{n}{2^l} \rceil, \\ 1 \le j \le p, 1 \le l \le \lceil \log_2 n \rceil \end{array}$$

$$[\,a' \to a\,]_{M_{i,j,l}} \quad \text{for } \begin{array}{l} 1 \le i \le \lceil \frac{n}{2^l} \rceil, \\ 1 \le j \le p, 1 \le l \le \lceil \log_2 n \rceil \end{array}$$

2^{l-1} membranes of each type $M_{i,j,l}$ are present in the step $3n+1$

$$M_{i,j,l}((3n+3)+2l+k(l-1)) : \left.\begin{array}{l} a^2 \rightsquigarrow c_{i,j,l+1} \\ a \rightsquigarrow c_{i,j,l+1} \end{array}\right\} \text{ in } k \text{ steps for } \begin{array}{l} 1 \le i \le \lceil \frac{n}{2^l} \rceil, \\ 1 \le j \le p, \\ 1 \le l \le \lceil \log_2 n \rceil - 1 \end{array}$$

$$M_{1,j,\lceil \log_2 n \rceil}((3n+3)+(2+k)\lceil \log_2 n \rceil - 1) : \left.\begin{array}{l} a^2 \rightsquigarrow c_{j,1} \\ a \rightsquigarrow c_{j,1} \end{array}\right\} \text{ in } k \text{ steps for } 1 \le j \le p$$

3.1 Rules to check if all the clauses are satisfied by a truth assignment.

$$\left.\begin{array}{l} d_{j-1}\,[\quad]_{M_j} \to [\,a'\,]_{M_j} \\ c_{j,(j-1)(k+2)+1}\,[\quad]_{M_j} \to [\,a\,]_{M_j} \end{array}\right\} \text{ for } 1 \le j \le p$$

$$[\,c_{j,l} \to c_{j,l+1}\,]_2 \quad \text{for } 1 \le j \le p, 1 \le j \le (j-1)(k+2)$$

$$M_j((3n+3)+(2+k)\lceil \log_2 n \rceil + 2j + k(j-1)) : a^2 \rightsquigarrow d_j \text{ in } k \text{ steps} \quad \text{for } 1 \le j \le p$$

4.1 Rules to obtain only one copy of the object d_p, if possible.

$$[\,d_p\,]_2 \to d_{p,1}[\quad]_2$$

$$d_{p,l}\,[\quad]_{M_{d,l}} \to [\,a\,]_{M_{d,l}} \quad \text{for } 1 \le l \le \lceil \log_2 n \rceil$$

2^{l-1} membranes of each type $M_{d,l}$ are present in the step $(3n+3)+(2+k)(\lceil \log_2 n \rceil + p)$

$$M_{d,l}((3n+3)+(2+k)(\lceil \log_2 n \rceil + p)+2l+k(l-1)) : \left.\begin{array}{l} a^2 \rightsquigarrow d_{p,l+1} \\ a \rightsquigarrow d_{p,l+1} \end{array}\right\} \text{ for } 1 \le l \le \lceil \log_2 n \rceil$$

4.2 Rules to return the correct answer.

$$d_{p,\lceil \log_2 n \rceil + 1}\,[\quad]_{M_r} \to [\,a\,]_{M_r}$$

$$M_r((3n+3)+(2+k)(2\lceil \log_2 n \rceil + p + 1)) : \left.\begin{array}{l} a \rightsquigarrow \texttt{yes} \\ a^0 \rightsquigarrow \texttt{no} \end{array}\right\} \text{ in } k \text{ steps}$$

$$[\,\texttt{yes}\,]_1 \to \texttt{yes}\,[\quad]_1$$

$$[\,\texttt{no}\,]_1 \to \texttt{no}\,[\quad]_1$$

5. The input membrane is the membrane labelled by 2 ($i_{in} = 2$) and the output zone is the environment ($i_{out} = env$).

The proposed solution follows a brute force algorithm in the framework of recognizer P systems with active membranes without dissolution rules and division rules for elementary and non-elementary membranes, and it consists on the following stages:

1. *Generation stage*: Using rules from **1.1**, 2^n membranes labelled by 2 and 3 will be produced. Each of the membranes labelled by 2 will contain a different truth assignment for the n variables. This stage takes $3n + 1$ time steps.
2. *First checking stage*: In this stage, an object $c_{j,1}$ will be produced in the membranes labelled by 2 whose truth assignment makes true the clause C_j. First of all, with rules from **2.1**, multiple copies of objects $c_{i,j,1}$ will be produced, and later with the rules from **2.2** will lead to a single copy of objects $c_{j,1}$ if there was at least one object $c_{i,j,1}$. This stage takes $(2 + k)\lceil \log_2 n \rceil$.
3. *Second checking stage*: In this stage, an object d_p will be produced in a membrane labelled by 2 if the truth assignment associated with it makes true the whole formula φ, by using rules from **3.1**. This stage takes $p(2+k)$ time steps.
4. *Output stage*: Finally, by using rules from **4.1**, a single object $\mathbf{d}_{p,\lceil \log_2 n \rceil}$ will be produced in the skin membrane if there exists at least one truth assignment that makes true the formula φ. If such an object exists, the system will send an object `yes` to the environment. Otherwise, it will return an object `no`. This is done by using the rules from **4.2**. This stage takes $\lceil \log_2 n \rceil + 4$ time steps.

4 Details of the special machines

In the previous solution, new syntax with respect to the classical of membrane systems appears. Let us define some kind of "machines" that follows a not-so-much special behavior. A machine M_i is no more than a P system from the corresponding family, in this case, from $\mathcal{AM}^0(-d, +n)$. Given a machine M_i, we say that $multisets(M_i)$ will be the multisets of objects placed initially in the membranes within the structure of the machine M_i, $alphabet(M_i)$ will be the working alphabet of the machine and if there is a rule of the kind $M_i(t) : rules$, the machine M_i will execute the corresponding rule associated to the number of objects placed in the system at configuration $\mathcal{C}_t$. In k steps, the machine M_i will return the corresponding answer to the parent membrane. Of course, not all the machines will spend the same amount of time steps, since in the end they are P systems and different machines can spend different times, but for the sake of simplicity, we say that they spend the same number of time steps. Later, we will look for a simple machine that must spend exactly k time steps, and that machine will be a "sub-routine" used for the ones used in the solution.

As opposed to oracles, that in this sense they can be though as machines that start working when they receive some input, these machines are "running" from the first configuration; that is, they are P systems that work as a P system of its family, so it cannot accomplish tasks that are impossible for P systems of its own

family.

We can think that the "machine" can wait until step t by using subscripts and evolution rules. Since division rules are allowed for both elementary and non-elementary rules, we can ensure that we can obtain enough number of machines of each kind. For this purpose, if a machine M_i needs to be replicated into 2^k copies, a single copy of this machine is present at the beginning of the computation. In the skin membrane of this machine, there is a *leaf* membrane labelled by d, and such that it contains an object: a_1. Let us define the following rules:

$[\, a_{2i} \to a_{2i+1} \,]_d$ for $1 \leq i \leq k-1$

$[\, a_{2i-1} \,]_d \to [\, a_{2i} \,]_d [\, a_{2i} \,]_d$ for $1 \leq i \leq k$

$[\, [\quad]_d [\quad]_d \,]_{skin_{M_i}} \to [\, [\quad]_d \,]_{skin_{M_i}} [\, [\quad]_d \,]_{skin_{M_i}}$

By using these rules, we can obtain 2^k exact copies of the machine M_i in $2k-1$ steps. The label d is a label such that it is not used in the whole machine M_i. Since the rest of objects are supposed to be evolving within their corresponding membranes, this process does not affect in this task.

5 Duties of the special machines

Five different machines are described in the previous solution. As discussed in the previous section, a machine M_i such that it has a rules defined as $M_i(t) : rule$, then when the configuration $\mathcal{C}_t$ is reached, the rule will be executed from the next transition. In k time steps, it will return to its parent membrane the corresponding answer.

- $M_{i,j}$: If there are 2 copies of the object a, then it returns an object $c_{i,j,1}$. Otherwise, it returns nothing.
- $M_{i,j,l}$: If there are 1 or 2 copies of the object a, then it returns an object $c_{i,j,l+1}$. Otherwise, it returns nothing.
- M_j: If there are 2 copies of the object a, then it returns an object d_j. Otherwise, it returns nothing.
- $M_{d,l}$: If there are 1 or 2 copies of the object a, then it returns an object $d_{p,l+1}$. Otherwise, it returns nothing.
- M_r: If there are 1 copy of the object a, then it returns an object `yes`. If there are no copies of the object a, it returns `no`. Otherwise, it returns nothing.

As we can observe, three different behaviors are required here. For instance, the answers of $M_{i,j}$ and M_j are similar. Since they do not return the same object, we can think that we have a single type of machine M_i, and it is within a membrane labelled by $M'_{i,j}$ (respectively, M'_j). The behavior of this machine is simple: If there are 2 copies of the object a, then it returns `yes`. Otherwise, it returns `no`. The corresponding answer would be sent at the $(k-1)$-th time step to the membrane

$M'_{i,j}$ (resp., M'_j). Then, in the k-th time step, the membrane $M'_{i,j}$ (resp., M'_j) would send to the parent membrane an object $c_{i,j,1}$ (resp., d_j) if an object `yes` appeared in the previous configuration, and it would not send anything if an object `no` had appeared.

1. $M_{i,j}, M_j$: Differentiate between 1 and 2 copies of object a.
2. $M_{i,j,l}, M_{d,l}$: Differentiate between 1 or 2 and none copies of object a.
3. M_r: Differentiate between 1 and 0 copies of object a.

In fact, taking into account that we are *deciding* if the number of objects a is equal to a number, then we can generalize these three cases into a single one, in order to have a single problem to be solved.

6 Covering all the cases

What we are trying to solve here is deciding if the number of objects of a certain type corresponds to a particular number.

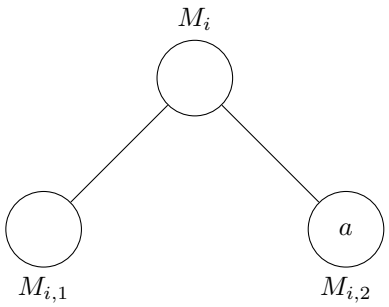
Then, the following question could generalize the previous problems:

> *Can we differentiate the existence of a single object from the non-existence or the multiple existence of the object?*

Or, more formally, if there is a single instance of an object, then return `yes`. Otherwise, return `no`.

We are going to give a proof on how these three cases can be reduced to this question.

1. The first case is reduced as follows: let $M_{i,1}$, $M_{i,2}$ and $M_{i,3}$ be three machines that solve the current problem. Then, we can think that the machine M_i is formed as follows:



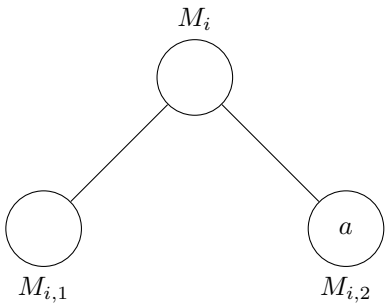
For sending objects a to each machine $M_{i,j}$, they can be replicated by using object evolution rules and sent to them by using send-in rules (for the order, subscripts can be used). $M_{i,1}$ will solve the problem normally; that is, if there is only one object a, then it will send an object $\texttt{yes}_1$ to M_i in k steps, otherwise, it will return an object $\texttt{no}_1$. At the same time, the problem will be solved in $M_{i,2}$, but as there is an object a present in the system, it will return an object $\texttt{yes}_2$ if there are no objects a in M_i, otherwise it will return an object $\texttt{no}_2$.

If $M_{i,1}$ returns an object $\texttt{yes}_1$, it means that there is only one copy of the object a. Therefore, M_i will not send anything to its parent membrane. If $M_{i,1}$ returns $\texttt{no}_1$, there are two possibilities: On the one hand, there can be no objects a, then $M_{i,2}$ will return an object $\texttt{yes}_2$. On the other hand, there can be two copies of object a, then $M_{i,2}$ will return $M_{i,2}$ will return an object $\texttt{no}_2$. The following table represent the desired output for each possibility:

Input	$M_{i,1}$	$M_{i,2}$	Output
$n = 0$	$\texttt{no}_1$	$\texttt{yes}_2$	no
$n = 1$	$\texttt{yes}_1$	$\texttt{no}_2$	no
$n = 2$	$\texttt{no}_1$	$\texttt{no}_2$	yes

Since these three cases are the only possible ones, it is easy to see that if only an object no is present, it must return no (that will not be sent to the parent membrane, since in these situations, there is only one or none copies of object a). The other possible situation is that two objects no appear. In that case, we should return an object yes (that later will be sent to the parent membrane as the corresponding object). It is easy to do it by changing objects $\texttt{no}_i$ to objects a. If there is a single copy, the machine M_i will return nothing. Otherwise, it will send to the parent membrane the corresponding object.

2. The second case is similar to the first one. In this case, the objective of M_i is to return an object when there are 1 or 2 copies of the object a. In the other case; that is, when there are no copies of the object a, it will return nothing. As in the previous case, we will have the following structure:



If $M_{i,1}$ and $M_{i,2}$ work as in the previous case, while there are 2 copies of the object a, it will return the correct answer. But in the case that there is a single copy of a, then it will return nothing, and this is not the behavior that we expect. But, with a simple trick, we can transform this answer to the correct one just by flipping the answer of $M_{i,2}$; that is, if there is a single object a, then return $\texttt{no}_2$. Otherwise, it returns $\texttt{yes}_2$. Remember that it is possible since it can have an external membrane that has object evolution rules similar to $\texttt{yes} \rightarrow \texttt{no}_2$ and $\texttt{no} \rightarrow \texttt{yes}_2$. Therefore, the following table represents the desired output taking into account the behavior of $M_{i,1}$ and $M_{i,2}$:

Input	$M_{i,1}$	$M_{i,2}$	Output
$n = 0$	$\texttt{no}_1$	$\texttt{yes}_2$	`no`
$n = 1$	$\texttt{yes}_1$	$\texttt{no}_2$	`yes`
$n = 2$	$\texttt{no}_1$	$\texttt{no}_2$	`yes`

By looking at the table, we can clearly see that if an object $\texttt{no}_2$ comes out from the machine $M_{i,2}$, then it will return `yes` (that will be sent to the parent membrane as the corresponding object). Then, we can transform object $\texttt{no}_2$ into an object a. Objects $\texttt{no}_1$, $\texttt{yes}_1$ and $\texttt{yes}_2$ will not evolve into an object a. Therefore, if objects $\texttt{no}_1$ and $\texttt{yes}_2$ are the output of machines $M_{i,1}$ and $M_{i,2}$ there will not be any object a in M_i, thus it will return a `no` (that will not be sent to the parent membrane). If object $\texttt{no}_2$ is the output of $M_{i,2}$, no matter the output of $M_{i,1}$, since it will not produce another object a will produce a `yes` as an output (and it will be sent to the parent membrane as the corresponding object).

3. The third case is trivial since only two scenarios can occur: On the one hand, if there is no object a it will return an object `no`. On the other hand, if there is an object a present in the machine, then it will return a `yes`.

The three previous problems have been reduced to the previously stated question. Therefore, having a P system from $\mathcal{AM}^0(-d, +n)$ capable of solving this

problem, would complete the solution. This machine must be totally independent from the input, and this machine will spend exactly k steps.

7 Reduction of the problem

In order to prove $\mathbf{NP} \cup \mathbf{co} - \mathbf{NP} \subseteq \mathbf{PMC}_{\mathcal{R}}$, an efficient solution to an **NP**-complete problem by means of a family of recognizer P systems from $\mathcal{R}$ must be provided. In this paper, a "solution" to the SAT problem has been provided by means of a family of recognizer membrane systems from $\mathcal{AM}^0(-d, +n)$. This solution depends on the ability of these P systems to solve the proposed problem. Thus, the existence of a single membrane system of this family capable of solving this question would lead to $\mathbf{NP} \cup \mathbf{co} - \mathbf{NP} \subseteq \mathbf{PMC}_{\mathcal{AM}^0(-d,+n)}$.

In [1], by using the dependency graph technique, it was proved that $\mathbf{P} = \mathbf{PMC}_{\mathcal{AM}^0(-d,+n)}$. Therefore, a single membrane system from this family solving the cited problem would not exist unless $\mathbf{P} = \mathbf{NP}$.

Thus, a new tool to tackle the **P** *versus* **NP** problem has been stablished: If there exists a single membrane from $\mathcal{AM}^0(-d, +n)$ solving the problem of differentiating a single appearance of a certain object from its non-existence or the multiple existence, then $\mathbf{P} = \mathbf{NP}$.

8 Conclusions and future work

In this paper, a "solution" to the SAT problem in the framework of recognizer P systems from $\mathcal{AM}^0(-d, +n)$ has been given. This is not a real solution since it needs of special "machines" that execute tasks whose possible execution in this framework has not been demonstrated. Since a positive solution of this problem yields $\mathbf{P} = \mathbf{NP}$, a very powerful tool to tackle this problem has been raised.

Two interesting research lines open up: On the one hand, solve this problem as it would lead to a very powerful result (in fact, if it ends up being a question with an affirmative answer, an efficient mechanism to solve **NP**-complete problems would raise from the solution). On the other hand, to explore the existence of more conjectures of this type, since they can be helpful to solve, in an affirmative or in a negative way, the **P** *versus* **NP** problem.

References

1. M.Á. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. On the power of dissolution in P systems with active membranes. In

R. Freund, Gh. Păun, Gr. Rozenberg, A. Salomaa (eds.). *Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers, Lecture Notes in Computer Science*, **3850** (2006), 224-240.

2. M.Á. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, F.J. Romero-Campero. A uniform solution to SAT using membrane creation. *Theoretical Computer Science*, **371**, 1–2 (2007), 54–61.
3. L. Pan, T-.O.Ishdorj. P systems with active membranes and separation rules. *Journal of Universal Computer Science*, **10**, 5 (2004), 630–649.
4. Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998.
5. Gh. Păun. P systems with active membranes: attacking **NP**–complete problems, *Journal of Automata, Languages and Combinatorics*, **6**, 1 (2001), 75-90.
6. M.J. Pérez-Jiménez, A. Riscos-Núñez, Á. Romero-Jiménez, D. Woods. Complexity: Membrane division, membrane creation. In: Păun et al. (eds.), The Oxford Handbook of Membrane Computing, Oxford University Press, 2010, chap. 12, 302–336.
7. M.J. Pérez-Jiménez, Á. Romero Jiménez. Simulating Turing Machines by P systems with External Output. *Fundamenta Informaticae*, Annales Societatis Mathematicae Polonae, Series IV, IOS Press, Amsterdam, **49**, 1-3 (2002), 273-287
8. M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini. Decision P systems and the $\mathbf{P} \neq \mathbf{NP}$ conjecture. In Gh. Păun, Gr. Rozenberg, A. Salomaa, C. Zandron (eds.) Membrane Computing 2002. *Lecture Notes in Computer Science*, **2597** (2003), 388-399. A preliminary version in Gh. Păun, C. Zandron (eds.) *Pre-proceedings of Workshop on Membrane Computing 2002*, MolCoNet project-IST-2001-32008, Publication No. 1, Curtea de Arges, Romanian, August 19-23, 2002, pp. 345-354.
9. Á. Romero-Jiménez, D. Orellana-Martín. Design Patterns for Efficient to NP-Complete Problems in Membrane Computing. In C. Graciani et al. (eds.), Enjoying Natural Computing: Essays Dedicated to Mario de Jesús Pérez-Jiménez on the Occasion of His 70th Birthday, Springer, 2018, chap. 19, 237–255.

A syntax for semantics in P-Lingua

Ignacio Pérez-Hurtado, David Orellana-Martín,
Agustín Riscos-Núñez, and Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla, Spain
{perezh,dorellana,ariscosn,marper}@us.es

Summary. P-Lingua is a software framework for Membrane Computing, it includes a programming language, also called P-Lingua, for writting P system definitions using a syntax close to standard scientific notation. The first line of a P-Lingua file is an unique identifier defining the variant or *model* of P system to be used, i.e, the semantics of the P system. Software tools based on P-Lingua use this identifier to select a simulation algorithm implementing the corresponding derivation mode. Derivation modes define how to obtain a configuration C_{t+1} from a configuration C_t. This information is usually hard-coded in the simulation algorithm.

The P system model also defines what types or rules can be used, the P-Lingua compiler uses the identifier to select an specific parser for the file. In this case, a set of parsers is codified within the compiler tool. One for each unique identifier.

P-Lingua has grown during the last 12 years, including more and more P system models. From a software engineering point of view, this approximation implies a continous development of the framework, leading to a monolithic software which is hard to debug and maintain.

In this paper, we propose a new software approximation for the framework, including a new syntax for defining rule patterns and derivation modes. The P-Lingua users can now define custom P system models instead of hard-coding them in the software. This approximation leads to a more flexible solution which is easier to maintain and debug. Moreover, users could define and *play* with new/experimental P system models.

1 Introduction

Membrane computing is an unconventional model of computation within natural computing that was introduced in 1998 by Gh. Păun [17]. The computational devices in membrane computing, also known as P systems, are non-deterministic theoretical machines inspired on the biochemical processes that take place inside the compartments of living cells.

Several kinds of P systems coexist, each of them having different syntactic ingredients, such as different alphabets and structures. The two most studied are

cell-like membrane systems, characterized by their rooted tree structure, where membranes act as filters that let certain elements to pass through them [17], and membrane systems structured as directed graphs, representing the communication between cells within a tissue of a living being, called tissue-like P systems [9] or between neurons in a brain, called spiking neural P systems [7]. The interchange of objects between the different *compartments* is defined by the *rules* of the system, that together with the corresponding semantics, mark the functioning of the system.

A *configuration* of a P system is defined by the structure of the compartments at a certain moment, and the elements (being usually objects, although other kinds of elements can be considered, as strings, catalysts [17] and anti-matter [14], among others) contained in each compartment, as well as other characteristics from specific types of P systems, providing a *snapshot* of the system at an instant t. By using the rules specified in a model, we can make its objects change, both evolving and moving between the different compartments (membranes in the case of cell-like P systems and cells in the case of tissue-like P systems).

On the one hand, in P systems with active membranes [19], both objects and membranes change through the application of evolution, communication, division, separation, creation and dissolution. In this framework, membranes can have a polarization associated to each membrane. On the other hand, in tissue P systems [9], symport/antiport rules are devoted to make objects move from a cell to another cell or to the *environment* (a special compartment where there exist an arbitrary number of objects of an alphabet defined *a priori*), while division and separation rules allow an exponential growth in linear time.

We say that a configuration C_t yields to a configuration C_{t+1} if, by applying the rules specified in the model according to its semantics, we can obtain C_{t+1} from C_t. Semantics rules the behavior of the system, determining which rules can be applied and how they affect the system according to a global clock. A *computation* of a P system is a (finite or infinite) sequence of instantaneous configurations.

We consider a *family* (or *model*) of P systems as the definition of a type of P system, that is, its syntax and semantics. According to the specification of a particular family of P systems, we consider a (specific) *model* as the definition of an individual P system, that is, its working alphabet, initial membrane structure with initial multisets of objects and the set of rewriting rules with another characteristics of the correspondent family. By the definition of the family, we can interpret the structure and behavior of a specific model within that family.

Membrane computing is a very flexible framework where different types of devices can be outlined. In fact, the intersection between Membrane Computing and other fields, such as engineering [20], biology [23] and ecology [2], as well as a long list of other scientific lines [5, 13, 24], has generated necessities that could only be filled by the creation of new kinds of P systems, expanding the scope of researchers in this area. For an exhaustive explanation of the different types of P systems, we refer the reader to [18] and [16].

In this work, we have *reinvented* the P-Lingua framework [6, 25] to include semantic features concerning to the models.

The paper is structured as follows: In the next section, some preliminaries concepts about P-Lingua are introduced. In Section 3, we propose an extension for the P-Lingua language to directly define model constraints in the own P-Lingua files, providing a more flexible and experimental framework. The next Section is devoted to the new GNU GPLv3 software tool to compile the input P-Lingua files. In Section 5 some examples of the new P-Lingua extension are introduced. Finally, some conclusions and future work are drawn.

2 Preliminaries

P-Lingua [6, 25] is a software framework that includes a definition language for P systems (also called P-Lingua) and a GNU GPLv3 Java library (pLinguaCore) that is able to parse P-Lingua files and simulate computations. The library contains three main components:

- A parser for reading input files in P-Lingua format and checking syntactic and semantic constraints related to predefined models. In order to achieve this, the first line of a P-Lingua file should include a P system model declaration by using an unique identifier. There are several P system models that can be used, each one with its own identifier, such as `transition`, `membrane_division`, `tissue_psystems`, and `probabilistic`. The analysis of semantic ingredients, such as rule patterns, is hard-coded for each model. Several versions of pLinguaCore [6, 8, 10, 21] have been launched to cover different types of models.
- For each type of model, the pLinguaCore library includes one or more built-in simulators, each one implementing a different simulation algorithm. For instance, Population Dynamic P systems [1] (`probabilistic` identifier in P-Lingua) can be simulated within the library by applying three different algorithms: `BBB`, `DNDP`, and `DCBA` [3, 11]. Remarkable software projects such as MeCoSim (Membrane Computing Simulator) [27, 22] use the simulators integrated in the library to perform P system computations and generate relevant information as result for custom applications.
- Alternatively, the pLinguaCore library is able to transform the input P-Lingua files to other formats such as XML or binary format in order to feed external simulators. The generated files for the given P systems are free of syntactic/semantic errors since the transformation is done after the parser analysis. Several external simulators use this feature, for example, the PMCGPU project (Parallel simulators for membrane computing on GPU) [12, 26] uses definitions generated by pLinguaCore in order to provide the input of CUDA GPU simulators.

The P-Lingua language is currently a standard widely used for the scientific community since the syntax is modular, parametric and close to the common

scientific notation. The description of the language can be found in the references [6, 8, 10, 21, 25]. As an example, the definition of a basic transition P system follows:

```
@model<transition>
def main()
{
  @mu = [[[]'3 []'4]'2]'1;
  @ms(3) = a,f;

  [a --> a,bp]'3;
  [a --> bp,@d]'3;
  [f --> f*2]'3;

  [bp --> b]'2;
  [b []'4 --> b [c]'4]'2;
  (1) [f*2 --> f ]'2;
  (2) [f --> a,@d]'2;
}
```

In the example, a module `main` is defined including an initial membrane structure $[\,[\quad]_3\,[\quad]_4\,]_2\,]_1$, an initial multiset for the membrane labelled 3, and a set of seven multiset rewriting rules. The special symbol `@d` is used to specify dissolution. The last two rules include priorities as integer numbers in parenthesis at the beginning of the left-hand side of the rules. More complex examples can be found in the P-Lingua web [25].

3 An extension of P-Lingua for semantic features

As explained above, the analysis of semantic ingredients belonging to P systems is hard-coded in the pLinguaCore library, i.e, the only way to define new types of models is by implementing code inside the library. In this section, we propose an extension for the P-Lingua language to directly define model constraints in the own P-Lingua files, providing a more flexible and experimental framework. Two types of semantic constraints can be defined with this extension: *rule patterns* and *derivation modes*.

3.1 Rule patterns

The P-Lingua parser is able to recognize rules with the next general syntax:

$$p$$
$$u[v_1[v_{1,1}]^{\alpha_{1,1}}_{h_{1,1}} \dots [v_{1,m_1}]^{\alpha_{1,m_1}}_{h_{1,m_1}}]^{\alpha_1}_{h_1} \dots [v_n[v_{n,1}]^{\alpha_{n,1}}_{h_{n,1}} \dots [v_{n,m_n}]^{\alpha_{n,m_n}}_{h_{n,m_n}}]^{\alpha_n}_{h_n}$$

$$\xrightarrow{q} \text{ or } \xleftrightarrow{q}$$

$$w_0[w_1[w_{1,1}]_{g_{1,1}}^{\beta_{1,1}} \dots [w_{1,r_1}]_{g_{1,r_1}}^{\beta_{1,r_1}}]_{g_1}^{\beta_1} \dots [w_s[w_{s,1}]_{g_{s,1}}^{\beta_{s,1}} \dots [w_{s,r_s}]_{g_{s,r_s}}^{\beta_{s,r_s}}]_{g_s}^{\beta_s}$$

where:

- p is a priority related to the rule given by a natural number, where a lower number means a higher rule priority.
- q is a probability related to the rule given by a real number in $[0, 1]$.
- $\alpha_i, \alpha_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $\beta_i, \beta_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ are electrical charges.
- $h_i, h_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $g_i, g_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ are membrane labels.
- $u, v_i, v_{i,j}, 1 \leq i \leq n, 1 \leq j \leq m_i$ and $w_i, w_{i,j}, 1 \leq i \leq s, 1 \leq j \leq r_i$ are multisets of objects.

Next, there is a list of P-Lingua rule examples matching the general rule syntax:

- `a,b [ d,e*2 ]'h --> [f,g]'h :: q;` where `q` is the probability of the rule.
- `(p) [a]'h --> [b]'h;` where `p` is the priority of the rule.
- `[a --> b]'h;`, the left-hand side and right-hand side of evolution rules can be collapsed.
- `+[a]'h --> +[b]'h -[c]'h;` a division rule using electrical charges.
- `[a]'h --> ;` a dissolution rule.
- `a[ ]'h --> [b]'h;` a send-in rule.
- `[a]'h --> b[ ]'h;` a send-out rule.
- `[a --> #]'h;` the symbol `#` can be optionally used as empty multiset.
- `[a]'1 <--> [b]'0;` a symport/antiport rule in the tissue-like framework.

The syntax of the general rule is very permissive, and so different parsers for different models have been developed in order to restrict the rules used in each one. In order to provide the researcher a more flexible framework, not having to depend on the implementation itself but acquiring the capacity of restricting the model by himself, we introduce the next syntax in P-Lingua for rule pattern matching:

```
!rule-type-id
{
pattern1
pattern2
...
patternN
}
```

where `rule-type-identifier` is an unique name for the type of rule that is going to be defined and `pattern1`, `pattern2`, ..., `patternN` are rule patterns following the same syntax than common rules in P-Lingua where anonymous variables beginning

with `?` can be optionally used instead of probabilities, charges and priorities. In the patterns, the symbols beginning with `a`, `b` or `c` always mean single objects and symbols beginning with `u`, `v` and `w` always mean multisets of objects. In Section 5, are given several examples of rule patterns in P-Lingua for different types of cell-like and tissue-like models.

3.2 Derivation modes

From an informal point of view, we can see a derivation mode as the way a step of a P system is performed. As a part of semantics, it rules the exact application of rules of the system, deciding when rules can be applied or not when they are applicable. An extensive study of derivation modes can be found in [4]. In order to make the work self-content, we give a minimal definition of a derivation mode.

A derivation mode ϑ is defined as a function that selects different multisets of rules "really applicable" to a configuration C_t of a P system depending on a specification. For this purpose, let Π be a P system with $\mathcal{R}$ as its set of rules, R a multiset of compatible rules applicable to a P system at configuration C_t, and let $\mathbf{R}$ be the set of all multisets applicable to a P system at configuration C_t.

In this extension of P-Lingua we provide two main derivation modes:

- **Maximally parallel derivation mode** (*max*): It is the default mode for P systems. In this mode, we only take multisets from R that are not extensible, that is:
$$\mathbf{R}' = \{R \mid R \in \mathbf{R} \wedge \nexists R' \in \mathbf{R} : R \subsetneqq R'\}.$$
The multiset of rules finally applied to C_t is selected non-deterministically from $\mathbf{R}'$.
- **Bounded-by-rule parallel derivation mode** ($bound_{B_1,\dots,B_r}$): Let $\{a, b, \dots\}$ be the set of different types of rules present in a P system. B_i can be of the following forms:
  - $B_i = j, j \in \{a, b, \dots\}$;
  - $B_i = \beta_n(B_{1_i}, \dots, B_{r_i})$, being $n \in \mathbb{N}$, and for each $B_j = \beta_{m_j}(B_{1_j}, \dots, B_{r_j})$, $j \in \{1_i, \dots r_i\}$, $m_j \leq n$;
  - As a restriction, a label for a type of rule cannot appear more than once in the whole definition of the derivation mode.

  We say that n is the *bound* of $B_i = \beta_n$. We say that a type of rule (j) is in the *context* of B_i if:
  - There exists $B_i = \beta_n(j)$ (we call B_i its *immediate context*); and
  - There exists $B_i = \beta_n(B_{1_i}, \dots, B_{r_i})$ such that B_j is a context of the type of rule (j).

  This mode is defined *recursively*, and we can understand the *applicability* of the rules in a defined bounded-by-rule parallel derivation mode in the following sense:

- In a *context* $\beta_n(B_1, \ldots, B_r)$, the number of rules that can be applied in parallel in a P system in a configuration C_t is n; and
- In a bounded-by-rule parallel derivation mode $bound_{B_1,\ldots,B_r}$, if $B_i = j (j \in \{a, b, \ldots\})$, being $1 \leq i \leq r$, then rules of type j can be applied in a maximal way.

With this mode, we can define the classical mode used in P systems with active membranes, that is, evolution rules (a) can be applied in a maximal parallel mode, while the other types of rules (send-in communication rules (b), send-out communication rules (c), dissolution rules (d), division rules for elementary (e) and non-elementary (f) membranes) can be applied at most once per membrane at each computation step. It would be defined as $bound_{a,\beta_1(b,c,d,e,f)}$. If $\mathcal{R}_j$ is the set of rules from $\mathcal{R}$ of the type j, we formally define the bounded-by-rule maximally parallel mode by

$$\begin{aligned}\mathbf{R}' = \{\ & R \mid R \in \mathbf{R} \\ & \wedge \mid \{r \mid r \in R, r \in \mathcal{R}_j\} \mid \leq n \text{ for all } j \text{ in the context of } B_i = \beta_n \\ & \wedge \nexists R' \in \mathbf{R} : R \subsetneqq R'\}\end{aligned}$$

Thus, a model type can be defined in P-Lingua by aggregating the allowed rule patterns and its corresponding derivation modes, the syntax is as follows:

```
@model(id) = rule-type-id1,..., rule-type-idN;
```

where `id` is an unique identifier for the model and `rule-type-id1` ,..., `rule-type-idN` are unique identifiers for the corresponding allowed rule patterns. By default all rules behave in maximally parallel derivation mode, but rules can be grouped in sets to behave in bounded parallel derivation mode as follows:

```
@model(id) = @bound{rule-type-id,..., rule-type-idN};
```

where `bound` is a natural number defining the maximum number of rules in the group that can be applied to a given configuration. In Section 5, several examples of model definitions in P-Lingua are given.

4 Command-line tools

A set of two GNU GPLv3 command-line tools called **plingua** and **psim** have been implemented in C++ language with Flex [28] and Bison [29]. The source code including examples and instructions can be downloaded from

`https://github.com/RGNC/plingua`.

The tool provides three main functionalities:

- **Parsing P-Lingua files** while printing the syntactic and semantic errors to the standard error output. In this sense, the tool acts as a conventional compiler, showing the name of the file, as well as the number of the line and column for each error with a short description. The analysis of semantic errors is done by using the rule patterns and derivation modes defined in the own P-Lingua files. Several files can be compiled together like conventional programs, furthermore standard *makefiles* can be also used. The developer can decide to write the rule patterns and derivation modes in a set of files and reuse them in several projects. More explanations can be found in the website.
- **Generating JSON/XML/Binary files**. The tool is able to translate the definitions contained in P-Lingua files to standard formats such as JSON, XML and Binary (compressed bit-level file format) for compatibility with third-party simulators. The conversion is done after parsing the input files, thus the output files are free of syntactic/semantic errors and the third-party applications do not have to check them. Several P-Lingua files can be combined together in one output file, including also the selected derivation modes.
- **Simulation of P system computations**.

Detailed information about how to use these tools, including examples, can be found in the website of the project `https://github.com/RGNC/plingua`.

5 Examples

Next, there are some examples of rule patterns and definiton of derivation modes for several common P system models. Please, see the website of the project for more information.

5.1 Transition P systems

```
!transition_evolution /* Limited to rules with 3 inner membranes */
{
       [a -> v]'h;
       [a -> v, @d]'h;
   (?) [a -> v]'h;
   (?) [a -> v, @d]'h;
       [a [ ]'h1 --> v [w]'h1]'h;
       [a [ ]'h1 --> v [w]'h1]'h;
   (?) [a [ ]'h1 --> v [w]'h1]'h;
   (?) [a [ ]'h1 --> v [w]'h1]'h;
       [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
       [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
   (?) [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
   (?) [a [ ]'h1 [ ]'h2 --> v [w1]'h1 [w2]'h2]'h;
       [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
       [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
   (?) [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
   (?) [a [ ]'h1 [ ]'h2 [ ]'h3 --> v [w1]'h1 [w2]'h2 [w3]'h3]'h;
}

@model(transition) = transition_evolution;
```

5.2 Active membranes with division rules

```
!dam_evolution
{
   ?[a -> v]'h;
   ?[a ->  ]'h;
}

!dam_send_in
{
   a ?[ ]'h -> ?[b]'h;
}

!dam_send_out
{
   ?[a]'h -> b ?[ ]'h;
}

!dam_dissolution
{
   ?[a]'h -> b;
   ?[a]'h ->  ;
}

!dam_division
{
   ?[a]'h -> ?[ ]'h ?[ ]'h;
   ?[a]'h -> ?[b]'h ?[ ]'h;
   ?[a]'h -> ?[ ]'h ?[b]'h;
   ?[a]'h -> ?[b]'h ?[c]'h;
}

@model(membrane_division) =
       dam_evolution, @1{dam_send_in, dam_send_out, dam_dissolution, dam_division};
```

5.3 Tissue-like P systems with communication and cell division

```
!tissue_communication
{
  [u]'h1 <--> [v]'h2;
}

!tissue_division
{
   [a]'h -> [ ]'h [ ]'h;
   [a]'h -> [b]'h [ ]'h;
   [a]'h -> [ ]'h [b]'h;
   [a]'h -> [b]'h [c]'h;
}

@model(tissue_division) =
       tissue_communication, @1{tissue_division};
```

5.4 Population Dynamics P Systems

```
!pdp_evolution
{
   u1 ?[v1]'h -> u2 ?[v2]'h :: ?;
}

!pdp_environment_communication
{
   [[a]'e1 [ ]'e2]'h -> [[ ]'e1 [b]'e2]'h :: ?;
```

```
}

@model(probabilistic) =
       pdp_evolution, pdp_environment_communication;
```

6 Conclusions and future work

This paper *reinvents* P-Lingua moving forward to a more flexible tool which is easier to maintain and debug. The goal is twofold: On the one hand, it pretends to be a good assistant for researchers while verifying their designs, even working with experimental models. On the other hand, more general simulators can be developed, covering a large set of P system variants by reading and simulating the custom derivation modes.

Several lines are open for future work. From the point of view of the language, the semantic ingredients that can be written in P-Lingua should be studied in order to cover more types of models. For instance, defining bounds for the multiplicities of objects in different compartments, such as the environment in tissue-like P systems, where the multiplicity of objects can be infinite. On the other hand, custom directives could be included in P-Lingua files and translated to C preprocessor directives for the simulator. For example, if the design is confluent, a directive could be written to optimize the simulation time, since it is not necessary to simulate the non-determinism by using random numbers.

From the point of view of the simulation tools, we are interested about the integration with CUDA, OpenMP, FPGA and POSIX threads, optimizing the use of parallel architectures.

Acknowledgements

The authors acknowledge the support of the research project TIN2017-89842-P, co-financed by *Ministerio de Economía, Industria y Competitividad (MINECO)* of Spain, through the *Agencia Estatal de Investigación (AEI)*, and by *Fondo Europeo de Desarrollo Regional (FEDER)* of the European Union.

References

1. M. Colomer, A. Margalida, and M.J. Pérez-Jiménez. Population Dynamics P System (PDP) Models: A Standardized Protocol for Describing and Applying Novel Bio-Inspired Computing Tools, *Plos One*, 2013 8 (14), 1–13.
2. M. Cardona, M.A. Colomer, M.J. Prez-Jimnez, D. Sanuy, A. Margalida. Modeling ecosystems using P systems: The bearded vulture, a case study. Membrane Computing, 9th International Workshop, WMC 2008, Edinburgh, UK, July 28-31, 2008, Revised Selected and Invited Papers. *Lecture Notes in Computer Science*, 5391 (2009), 137-156.

3. M. Colomer, I. Pérez-Hurtado, M.J. Pérez Jiménez, and A. Riscos-Núñez. Comparing simulation algorithms for multienvironment probabilistic Psystem over a standard virtual ecosystem, *Natural Computing*, 11 (2012), 369–379.
4. R. Freund, S. Verlan. A Formal Framework for Static (Tissue) P Systems. *Lecture Notes in Computer Science*, 4860 (2007), 271–284.
5. P. Frisco, M. Gheorghe, M. J. Prez-Jimnez. Applications of Membrane Computing in Systems and Synthetic Biology. *Emergence, Complexity and Computation* (Series ISSN 2194-7287), Volume 7. Springer International Publishing, eBook ISBN 978-3-319-03191-0, Hardcover ISBN 978-3-319-03190-3, 2014, XVII + 266 pages (doi: 10.1007/978-3-319-03191-0).
6. M. García-Quismondo, R. Gutiérrez-Escudero, I. Pérez-Hurtado, M.J. Pérez-Jiménez, and A. Riscos-Núñez. An overview of P-Lingua 2.0, *Lecture Notes in Computer Science*, 5957 (2010), 264–288.
7. M. Ionescu, Gh. Păun, T. Yokomori. Spiking Neural P systems. *Fundamenta Informaticae*, **71**, 2-3 (2006), 279-308.
8. L.F. Macías, I. Pérez-Hurtado, M. García-Quismondo, L. Valencia, M.J. Prez-Jimnez, A. Riscos-Nez. A P-Lingua based simulator for Spiking Neural P systems. *Lecture Notes in Computer Science*, 7184 (2012), 257–281.
9. C. Martín-Vide, Gh. Păun, J. Pazos, A. Rodríghez-Patón. Tissue P systems. *Theoretical Computer Science*, **296**, 2 (2003), 295-326.
10. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M.J. Pérez-Jiménez, A. Riscos-Núñez. A P-Lingua based simulator for Tissue P systems. *Journal of Logic and Algebraic Programming*, 79, 6 (2010), 374–382.
11. M.A. Martínez-del-Amor, I. Pérez-Hurtado, M. García-Quismondo, et al. DCBA: Simulating population dynamics P systems with proportional objects distribution, *Lecture Notes in Computer Science*, 7762 (2013), 257–276.
12. M.A. Martínez-del-Amor, M. García-Quismondo, L.F. Macías-Ramos, L. Valencia-Cabrera, A. Riscos-Núñez, M.J. Pérez-Jiménez. Simulating P systems on GPU devices: a survey. Fundamenta Informaticae, 136, 3 (2015), 269–284.
13. L. Pan, Gh. Paun, M. J. Prez-Jimnez, T. Song. Bio-inspired Computing: Theories and Applications. *Communications in Computer and Information Science* (Series ISSN 1865-0929), Volume 472, Springer-Verlag Berlin Heidelberg, Print ISBN 978-3-662-45048-2, Online ISBN 978-3-662-45049-9, 2014, XX + 672 pages (`doi: 10.1007/978-3-662-45049-9`).
14. L. Pan, Gh. Păun. Spiking Neural P Systems with Anti-Matter. *International Journal of Computers Communications & Control*, **4**, 3 (2009), 273–282.
15. L. Pan, T.-O. Ishdorj. P Systems with Active Membranes and Separation Rules. *Proceedings of the Second Brainstorming Week on Membrane Computing*, 2-7 February, 2004, Sevilla, Spain, pp. 325–341.
16. Gh. Păun, G. Rozenberg, A. Salomaa (eds.). *The Oxford Handbook of Membrane Computing*, Oxford University Press, Oxford, 2010.
17. Gh. Păun. Computing with membranes. *Journal of Computer and System Sciences*, **61**, 1 (2000), 108–143, and *Turku Center for CS-TUCS Report* No. 208, 1998.
18. Gh. Păun. *Membrane Computing. An introduction.* Springer-Verlag, Berlin, 2002.
19. Gh. Păun. P systems with active membranes: attacking **NP**–complete problems, *Journal of Automata, Languages and Combinatorics*, **6** (2001), 75–90.
20. H. Peng, J. Wang, J. Ming, P. Shi, M.J. Prez-Jimnez, W. Yu, Ch. Tao. Fault diagnosis of power systems using intuitionistic fuzzy spiking neural P systems. *IEEE Transactions on Smart Grid*, in press (2017) (`doi: 10.1109/TSG.2017.2670602`).

21. I. Pérez-Hurtado, L. Valencia-Cabrera, J.M. Chacón, A. Riscos-Núñez, M.J. Pérez-Jiménez. A P-Lingua based Simulator for Tissue P Systems with Cell Separation. *Romanian Journal of Information Science and Technology*, 17 , 1 (2014), 89–102.
22. I. Pérez-Hurtado, L. Valencia-Cabrera, M.J. Pérez-Jiménez, M. Colomer, and A. Riscos-Núñez. *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), 637–643.
23. F.J. Romero-Campero, M.J. Prez-Jimnez. A model of the Quorum Sensing System in Vibrio Fischeri using P systems. *Artificial Life*, 14, 1 (2008), 95-109 (`doi: 10.1162/artl.2008.14.1.95`).
24. G. Zhang, M. J. Prez-Jimnez, M. Gheorghe. Real-life applications with Membrane Computing. *Emergence, Complexity and Computation* (Series ISSN 2194-7287), Volume 25. Springer International Publishing, Online ISBN 978-3-319-55989-6, Print ISBN 978-3-319-55987-2, 2017, X + 367 pages (`doi: 10.1007/978-3-319-55989-6`).
25. The P-Lingua web page: `http://www.p-lingua.org`.
26. The PMCGPU web page: `https://sourceforge.net/projects/pmcgpu/`
27. The MeCoSim web page: `http://www.p-lingua.org/mecosim/`.
28. The Flex web page: `https://github.com/westes/flexl`
29. The Bison web page: `https://www.gnu.org/software/bison/`

Search Based Software Engineering in Membrane Computing

Ana Ţurlea[1], Marian Gheorghe[2], Florentin Ipate[1]

[1] Faculty of Mathematics and Computer Science and ICUB
University of Bucharest, Bucharest, Romania
ana.turlea@fmi.unibuc.com, florentin.ipate@ifsoft.ro
[2] School of Electrical Engineering and Computer Science,
University of Bradford, Bradford, UK
m.gheorghe@bradford.ac.uk

Summary. This paper presents a testing approach for kernel P Systems (*kP systems*), based on test data generation for a given scenario. This method uses Genetic Algorithms to generate the input sets needed to trigger the given computation steps.

Keywords: membrane computing; kernel P systems; testing; genetic algorithms: test data generation.

1 Introduction

Membrane Systems [17], now known as P Systems, were founded by Gheorghe Păun in 1998 [15, 16]. Initially inspired by the structure and functioning of the living cells, the field has been developed very fast and different types of *P systems* being investigated. *Kernel P systems* (*kP systems*, for short), have been introduced in [2]. These systems can be simulated using a software framework, called *kPWorkbench* [1] or some earlier variants (so called *simple kP systems*) using P-Lingua and the MeCoSim simulator [3]. Having many computational models with different software implementations, associated with various applications, it is very important to develop testing methods, to check that the implementation agrees with the system specification. This testing methodology is called conformance testing, which tries to find the differences betweem the behaviour of an implementation and its specification. The testing task is not trivial, given the fact that the models are parallel and non-deterministic. Previous works on P systems testing include testing cell-like P systems with methods like finite state-based inspired [6], stream X-machine based testing [7], mutation testing for evaluating the efficiency of the test sets [11], model-checking based testing [8] and testing identifiable kernel P systems using X-machines [4].

Automated test data generation is a topic of interest in software engineering community. There are many evolutionary testing approaches that generate test

date from code, finite state machines and other models, but there are no applications in membrane computing community.

In kernel P systems, we can simulate the evolution of the model for a given number of steps, starting with an initial multiset. We can change the evolution of the system by adding new multisets as inputs for each evolution step.

This paper presents a testing approach for kernel P systems, using genetic algorithms to generate test data that leads to a given set of computation steps.

In Section 2 we present some basic information about kP systems, evolutionary functional testing, genetic algorithms, search based testing for extended finite state machines. Section 3 describes the kP system type used for testing, the configuration of the algorithm and some experimetal results. In Section 4 we present conclusions and future work.

2 Preliminaries

In this section we will present some basic details about kernel P systems and Search Based Testing using Genetic Algorithms. We will also present some approaches that use evolutionary algorithms to test Extended Finite State Machines.

2.1 Kernel P systems

In the following we will give a formal definition of kernel P systems (or kP systems) [2]. We start by introducing the concept of a *compartment type* utilised later in defining the compartments of a kernel P system.

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.*

Definition 2. *A* kP system *of degree n is a tuple $k\Pi = (A, \mu, C_1, \ldots, C_n, i_0)$, where*

- *A is a finite set of elements called objects;*
- *μ defines the membrane structure, which is a graph, (V, E), where V is a set of vertices representing components (compartments), and E is a set of edges, i. e., links between components;*
- *$C_i = (t_i, w_{i,0})$, $1 \leq i \leq n$, is a compartment of the system consisting of a* compartment type*, t_i, from a set T and an initial multiset, $w_{i,0}$ over A; the type $t_i = (R_i, \sigma_i)$ consists of a set of evolution rules, R_i, and an execution strategy, σ_i;*
- *i_0 is the output compartment where the result is obtained.*

Within the general kP systems framework, the following types of evolution rules have been considered so far:

- *rewriting and communication* rule: $x \rightarrow y\{g\}$, where g represents a **guard**, $x \in A^+$ and $y \in A^*$, where y is a multiset with potential different compartment type targets (each symbol from the right side of the rule can be sent to a different compartment, specified by its type; if multiple compartments of the same type are linked to the current compartment, then one is randomly chosen to be the target). Unlike cell-like P systems, the targets in kP systems indicate only the types of compartments to which the objects will be sent, not particular instances (for example, $y = (a_1, t_1) \dots (a_h, t_h)$, where $h \geq 0$, and for each $1 \leq j \leq h$, $a_j \in A$ and t_j indicates a compartment type from T).
- *structure changing* rules: membrane division, membrane dissolution, link creation and link destruction rules, which all may also incorporate complex guards and that are covered in detail in [2]. However, this type of rules will not be considered in the following discussion.

For a multiset w over A and an element $a \in A$, we denote by $|w|_a$ the number of objects a occurring in w. Let us denote $Rel = \{<, \leq, =, \neq, \geq, >\}$, the set of relational operators, $\gamma \in Rel$, a relational operator, and a^n a multiset, consisting of n copies of a. We first introduce an *abstract relational expression.*

Definition 3. *If g is the* abstract relational expression *denoting γa^n and w a multiset, then the guard g applied to w denotes the* relational expression $|w|_a \gamma n$.

The abstract relational expression g is true for the multiset w, if $|w|_a \gamma n$ is true.

We consider now the following Boolean operators $\neg$ (negation), $\wedge$ (conjunction) and $\vee$ (disjunction). An *abstract Boolean expression* is defined by one of the following conditions:

- any abstract relational expression is an abstract Boolean expression;
- if g and h are abstract Boolean expressions then $\neg g$, $g \wedge h$ and $g \vee h$ are abstract Boolean expressions.

The concept of a guard, introduced for kP systems, is a generalisation of the promoter and inhibitor concepts utilised by some variants of P systems.

Definition 4. *If g is an* abstract Boolean expression *containing g_i, $1 \leq i \leq q$, abstract relational expressions and w a multiset, then g applied to w means the* Boolean expression *obtained from g by applying g_i to w for any $i, 1 \leq i \leq q$.*

As in the case of an abstract relational expression, the guard g is true with respect to the multiset w, if the abstract Boolean expression g applied to w is true.

For example, if g is the guard defined by the abstract Boolean expression $\geq a^4 \wedge < b^2 \vee \neg > c$ and w a multiset, then g applied to w is true if it has at least 4 a's and less than 2 b's or no more than one c.

2.2 Evolutionary Functional Testing

Software testing is the process of finding errors in a system, also measuring the quality of the system. The correctness of a system is the most essential purpose

of testing. Automated testing can be divided into white-box testing and black-box testing. White-box testing (structural testing) uses the source code of the system to generate test cases, while black-box testing (functional testing) uses the systems specifications for test generation. In white box testing the tester needs to have a look inside the source code and find out which unit of code is behaving inappropriately. In black box testing, a tester uses the system architecture or specification and does not have access to the source code [10].

One of the common approaches of automated testing is model based test cases generation. The generated test cases (based on the model) reveal faults and verify if the implementation conforms to its specification. Transforming this problem into an optimisation problem, we can use evolutionary approaches.

Search based software testing represents automated search in a potentially large input space, guided by a problem specific fitness function. The search space depends on the problem and on the configuration of the parameters of the system. The fitness function guides the search to the test goal and scores different inputs of the system according to the test goal.

Test cases generation has been intensively studied for EFSMs.

Test cases are set of input values and expected results developed to cover certain test conditions. A test suite is a collection of test cases.

2.3 Genetic Algorithms

Genetic Algorithms (GAs) are metaheuristic search techniques (mainly applied in optimization problems) that simulate the biological evolution and have the following elements: populations of chromosomes (individuals, candidate solutions), selection according to a fitness function, crossover to produce new offspring and random mutation of new offspring [14].

GAs start with initialization of a population with random candidate solutions, evolve the population several times, until a solution is found or a stop condition is met. Each element (chromosome) from the population represents a sequence of variables/parameters. Variable values can be represented in binary form, real-numbers, or even characters.

At each evolution, individuals are evaluated and selected for the next generation. The quality of each individual is determined by a fitness function that depends upon the problem considered. If the chromosome is fitter, it is likely to be selected to reproduce more times [14]. The optimization problem can be to minimize or to maximize the fitness function.

Crossover is applied to the randomly selected parent chromosomes, exchanging information between them and creating new children chromosomes. Some common types of crossovers are: single-point crossover, multi-point crossover and uniform crossover.

Mutation is applied, with some probability, to each chromosome, randomly changing some of the individual's genes. A new evolution starts with these new individuals. Mutation prevents genetic pool from premature convergence (getting

stuck in local maxima/minima). The main purpose of mutation is to bring diversity in population.

As described in [14], a simple GA works as follows:

1. Start with a randomly generated population (the initial population).
2. Compute fitness function for each chromosome in the population.
3. Repeat the following steps until a new generation is created:
   a) Select a pair of parent individuals from the current population (a chromosome can be selected only once to become a parent);
   b) Apply crossover on the current pair to form two offsprings.
   c) Mutate the two offspring chromosomes, with a given probability, and place the resulting individuals in the new population.
4. Selection is applied on the current population along with the new population.
5. Go to step 2.

A *generation* is represented by an iteration of this process. The entire set of generations is called a *run*. Two different runs will produce different behaviors. In order to evaluate the efficacy of a genetic algorithm, we should run it multiple times and analyse the results.

2.4 Search based Testing for EFSM Models

An *extended finite state machine* (EFSM) is a six-tuple (S, s_0, V, I, O, T) [9] where S is the finite set of states, s_0 is the initial state, V is the finite set of context variables, I is the finite set of inputs, O is the finite set of outputs and T is the finite set of transitions. A transition is represented by a start state, an input that may have associated input parameters, a guard (logical expression), a sequential operation (a method with assignments and output statements) and the end state.

A *path* of an EFSM is a sequence of adjacent transitions, $p = S_1 \xrightarrow{f_1[g_1]} S_2 \xrightarrow{f_2[g_2]} \dots S_m \xrightarrow{f_m[g_m]} S_{m+1}$, where S_i represents the state i from that path, f_i is the method executed on the transition i and g_i is the guard of the transition i. A path can be feasible, if there exist values for the input parameters to satisfy guards and to trigger all transitions for that path, and infeasible, otherwise.

There are many approaches that generate values for input parameters for each method f_i from a given path, that satisfy the guard conditions g_i and trigger all transitions. Lefticaru and Ipate investigated in [13] the use of search based techniques for functional testing using state machines. Its purpose is to generate input data for chosen path in a state machine, so that it triggers the transitions, using three search techniques: simulated annealing, genetic algorithms and particle swarm optimization. Kalaji et al. [9] proposed an integrated search based test data generation using EFSMs. The approach has two phases. In the first phase, feasible paths are generated using a GA with a feasibility metric based on dataflow dependence as fitness function, satisfying transition coverage criteria. In the second phase those paths are used as inputs to generate test data that trigger the paths,

using a GA with a fitness function based on the branch distance function and approach level.

The approach proposed by Lefticaru and Ipate [12] is based on the state diagram and uses a genetic algorithm to generate test data. The first step is to find feasible paths to achieve some coverage criteria. The second step is to find, for each path, the input values for parameters, to trigger the transitions. The test data generation problem is converted to an optimization problem, aiming to minimize the fitness function.

Paper [18] generates test data for EFSMs and uses a hybrid genetic algorithm, improving the algorithm presented in [12].

For a particular path in the EFSM, a chromosome (individual, possible solution) is a list of values, $x = (x_1, x_2, \ldots, x_n)$, corresponding to all parameters of the methods, as they appear on that path. A solution is a chromosome with fitness function 0 that triggers transitions between states according to the selected path and validates the guards of each transition.

The fitness function used in this approach is: $fitness = approach\_level + normalized\_branch\_level$ ($f = al + nbl$). $approach\_level$ is calculated by $m-1-p$, where m is the length of the path to be executed and p is the number of nodes executed until the first unsatisfied guard on the path. $normalized\_branch\_level$ is the mapping onto $[0, 1)$ for $branch\_level$. $branch\_level$ computes, for the predicate that is not satisfied, how close the predicate was to being true, using the transformations from Table 1. The normalization function is $norm : [0, 101] \rightarrow [0, 1], norm(d) = 1 - 1.05^{-d}$.

Element	Objective function value obj
Boolean	if TRUE then 0 else K
$a = b$	if $abs(a - b) = 0$ then 0 else $abs(a - b) + K$
$a \neq b$	if $abs(a - b) \neq 0$ then 0 else K
$a < b$	if $a - b < 0$ then 0 else $(a - b) + K$
$a \leq b$	if $a - b \leq 0$ then 0 else $(a - b) + K$
$a > b$	if $b - a < 0$ then 0 else $(b - a) + K$
$a \geq b$	if $b - a \leq 0$ then 0 else $(b - a) + K$
$a \wedge b$	$obj(a) + obj(b)$
$a \vee b$	$min(obj(a), obj(b))$
$a\ xor\ b$	$obj((a \wedge \neg b) \vee (\neg a \wedge b))$
$\neg a$	Negation is moved inwards and propagated over a

Table 1. Tracey's objective functions for relational predicates and logical connectives. The value K, $K > 0$, refers to a constant which is always added if the term is not true

The algorithm ends when the stop criteria is reached or when the maximum number of evolutions is exceeded. After the selection step, a new generation is created using recombination, crossover and mutation.

3 Search based Testing for kP systems

Modelling systems specification can be also done by using kP systems. In this paper we will introduce kP systems that behave similar to EFSMs and apply testing approaches based on EFSMs.

We will consider deterministic kP systems with two compartments.

The main compartment will only consist of rewriting rules of the form $Aa \rightarrow Bb\{g\}$, where A, B and a, b belong to two disjoint sets (A, B play the role of the input and output states of a transition of an EFSM, respectively, whereas a, b are input and output of the transition, with g being its guard).At any moment only one rule is applied, i.e. the system is always working in sequential manner.

The other compartment is meant to send symbols to the main compartment that are inputs a for its rules. This compartment is behaving similar to an environment of an EFSM that provides inputs to it. The rules have the form $C \rightarrow D(a, M)$, where C, D and a belong to disjoint sets, as in the case of the main compartment; M is the type of that compartment.

An execution of the system from a given configuration consists in applying a rule from the input compartment, sending the input multiset to the main compartment and applying a corresponding rule from the main compartment.

The testing strategy developed in this paper will refer to the main compartment; the other one will just provide inputs, as presented above.

When we aim to simulate the execution of the main compartment for N steps then the compartment generating inputs should be able to generate N inputs. Execution of each rule $Aa \rightarrow Bb\{g\}$ in the main compartment depends of the availability of Aa and also on guard g that must be true. In order to generate suitable inputs both these conditions have to be fulfilled and this is what happens for a test generation.

We can automatically generate test data for kPSystems using genetic algorithms and the kPWorkbench tool to simulate the evolution of the system.

3.1 kP System Definition

In this subsection we will present the kP system configuration.

The kP system will be denoted $k\pi = (A, \mu, C_1, C_2, i_0)$. The kP system will have the following elements:

$\mu = (V, E)$, where $V = \{cM, cInp\}$, cM = main compartment and cInp = the input compartment, $E = \{(cM, cInp)\}$.

We denote A_{ST} and A_{IO} two disjoint sets and $A = A_{ST} \cup A_{IO}$. A_{ST} denotes symbols that are either corresponding to states in cM or symbols used in $cInp$ for generating inputs. A_{IO} are input and output symbols as well as symbols that appear in guards.

The membrane structure contains two compartments, $cM = (t_M, w_{M0})$, where $t_M = (R_M, \sigma_M)$ is the compartment type and w_{M0} is the initial multiset over A and $cInp = (t_I, w_{I_0})$, where $t_I = (R_I, \sigma_I)$ is the compartment type and w_{I0} is the initial multiset over A.

The main compartment type $t_M = (R_M, \sigma_M)$ consists of a set of evolution rules R_M and an execution strategy σ_M working in sequential manner. R_M contains only rewriting rules:$Aa \to Bb\{g\}$, where $A, B \in A_{ST}$ and $a, b \in A_{IO}$.

In an evolution step and a given configuration, only one rule can be applied.

The input compartment type $t_I = (R_I, \sigma_I)$ consists of a set of evolution rules R_I and the execution strategy σ_I working in sequential manner. R_I contains only rewriting and communication rules $C \to D(a, t_M)$, where $C, D \in A_{ST}$ and $a \in A_{IO}$

The initial configuration contains the initial values from the memory and the output compartment i_0 is represented by the main compartment.

Example 1. Let us consider the kP system $k\Pi_1 = (A, \mu, cM, cInp, i_0)$, where $i_0 = cM$, $A_{ST} = \{A, B, C, D, E, F, A_1, ..., A_6\}$, $A_{IO} = \{a, b.f, d, o, t, x\}$

$$R_M = \begin{cases} r_1 : A, f \to A, a\{< 3a \& = f\} & r_2 : A, f \to E\{= 3a\} \\ r_3 : A, t \to B, a\{< 3a \& = t \& < f\} & r_4 : B, x \to C\{= x\} \\ r_5 : B, d \to D\{= d\} & r_6 : C, b, x \to C\{>= x\} \\ r_7 : D, d \to D, b\{>= d\} & r_8 : C, o \to F\{< x\} \\ r_9 : D, o \to F\{< d\} \end{cases}$$

$$R_I = \begin{cases} r_{10} : A_1 \to A_2, (f, t_M) & r_{11} : A2 \to A3, (f, t_M) \\ r_{12} : A3 \to A4, (t, t_M) & r_{13} : A4 \to A5, (x, t_M) \\ r_{14} : A5 \to A6, (3x, t_M) & r_{15} : A_6 \to A_7, (o, t_M) \end{cases}$$

The inital configuration of $k\pi_1$ is $M_0 = (100b\, A, A_1)$.

The only applicable rule is r_{10} for $cInp$ and $A_1 \Longrightarrow^{r_{10}} A_2, (f, M)$ and f goes to cM. Hence, the next configuration is $M_1 = (100b\, A\, f, A_2)$.

In this configuration we can only apply rule r_1 in cM and rule r_{11} in $cInp$, $A_2 \Longrightarrow^{r_{11}} A_3, (f, M)$ in $cInp$, f goes to cM and $A \Longrightarrow^{r_1} A, a$ in cM and the next configuration is $M_2 = (100b\, A\, a\, f, A_3)$.

The next computational step is $A_3 \Longrightarrow^{r_{12}} A_4, (t, M)$ in $cInp$, t goes to cM and $A, f \Longrightarrow^{r_1} A, a$ in cM and the next configuration is $M_2 = (100b\, A\, 2a\, t, A_4)$.

The next configuration is $M_3 = (100b\ B\ 3a\ x, A_5)$ obtained by applying $A_4 \Longrightarrow^{r_{13}} A_5, (x, M)$ in $cInp$, sending x to cM and applying $A, t \Longrightarrow^{r_3} B, a$ in cM.

After this step, the only applicable rules are $A_5 \Longrightarrow^{r_{14}} A_6, (3x, M)$ in $cInp$ and $B, x \Longrightarrow^{r_4} C$ in cM, sending $3x$ to cM and the next configuration is $M_4 = (100b\, C\, 3a\, 3x, A_6)$.

From this configuration we can apply $A_6 \Longrightarrow^{r_{15}} A_7, (o, M)$ in $cInp$ and $C, b, x \Longrightarrow^{r_6} C$ in cM, sending o to cM and reaching configuration $M_5 = (99b\, C\, 3a\, 2x\, o, A_7)$.

The next computational step contains the rule $C, b, x \Longrightarrow^{r_6} C$ applied in cM, obtaining the configuration $M_6 = (98b\, C\, 3a\, x\, o, A_7)$. In the next step, the same rule is applied identically in cM and $M_7 = (97b\, C\, 3a\, o, A_7)$.

In the last computational step, the applicable rule is $C, o \Longrightarrow^{r_8} F$ in cM and the final configuration is $M_8 = (97b\, F\, 3a, A_7)$.

The evolution steps obtained by this simulation are the following:

- Step 1: rule r_{10}
- Step 2: rules $r_{11}r_1$
- Step 3: rules $r_{12}r_1$
- Step 4: rules $r_{13}r_3$
- Step 5: rules $r_{14}r_4$
- Step 6: rules $r_{15}r_6$
- Step 7: rule r_6
- Step 8: rule r_6
- Step 9: rule r_8

To simulate the execution of a system we use kPWorkbench. kPWorkbench is an integrated software suite aimed to provide support for kP systems. Among other functionallities, kPWorkbench contains tools for modelling, simulating and verifying kP systems. A simulation trace represents the evolution of the system during some computations.

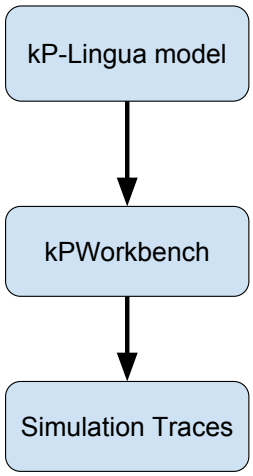


Fig. 1. kPWorkbench simulation steps

3.2 Genetic Algorithm Configuration

The Genetic Algorithm has the following steps:

- create random initial population - length N;

- evaluate the population using the fitness function;
- repeat the following steps until the stopping condition is reached:
  - offspring population ← reproduction(population);
  - evaluate offsprig popultion;
  - population ← reinsertion+selection(population, offspring population).

A chromosome $(x_1, x_2, \ldots, x_n)$ is represented as a list of input symbols corresponding to the input set. A gene represents the input for the corresponding step and consists of a list of strings (input symbols) $x_i = (r_1 s_1, r_2 s_2, \ldots, r_n s_n)$, where s_i is a symbol from the alphabet, and r_i is the number of times the symbol appears in the input set.

The *Crossover Operator* creates two new chromosomes from the two existing parent chromosomes, using one of the two operations, with equal probability.

- exchange only the value for a gene from a random selected point

$$\begin{cases}(x_1, x_2, \ldots, x_n)\\(y_1, y_2, \ldots, y_n)\end{cases} \rightarrow \begin{cases}(x_1, x_2, \ldots, y_i, \ldots, x_n)\\(y_1, y_2, \ldots, x_i, \ldots, y_n)\end{cases}$$

- exchange for a random point only a part of the gene

$$\begin{cases}(x_1, x_2, \ldots, x_i, \ldots, x_n)\\(y_1, y_2, \ldots, y_i, \ldots, y_n)\end{cases} \rightarrow \begin{cases}(x_1, x_2, \ldots, x'_i, \ldots, x_n)\\(y_1, y_2, \ldots, y'_i, \ldots, y_n)\end{cases}$$

  where
  $x_i = (r_1 s_1, \ldots, r_n s_n), y_i = (t_1 s_1, \ldots, t_n s_n)$,
  $x'_i = (r_1 s_1, \ldots, t_i s_i, \ldots, t_n s_n), y'_i = (t_1 s_1, \ldots, r_i s_i, \ldots, r_n s_n)$

A chromosome can be mutated in many different ways. To identify possible mutation operators, we considered the characteristics of a chromosome.We have defined the following different mutation operators, which are all applied with 0.5 probability:

- completly change a gene (an input value)
- remove gene part - for an input value choose randomly a symbol that will not be used ($r_i = 0$)
- for a random gene - replace random symbol number(r_i)
- exchange materials between two genes

We tried to apply the selection operator as it was used in many test data generation approaches. The reinsertion of the offspring population into the new population was made in different ways:

- the new population = the offspring population [12];
- the new population = the offspring population and the fittest individual is kept in the next generation [18];

- apply selection operator and select N chromosomes from the offspring population along with the old population, using different selectors: best chromosome selection, binary tournment selection.

None of these methods worked for our problem. The algorithm was stuck in a local optima. In order to overcome this problem we change the reinsertion method: the best 50% of the current generation and the best 50% of the new offspring are retained. In the next evolution, the crossover operator will use a parent that came from the old population and a parent that came from the offspring population and will create a new individual. This reinsertion method was inspired and adapted from paper [5].

$P = \{x_1, y_1, x_2, y_2, \ldots, x_{25}, y_{25}\}$, where $x_1, x_2, \ldots, x_{25} \in Old_{Population}$ and $y_1, y_2, \ldots, y_{25} \in Offspring_{population}$

To evaluate an individual we need to compute the objective function. To verify if a chromosome is the solution, we need to simulate the system with the corresponding input values and compare the simulation traces with the given steps. The fitness function is based on the *approach level* and the *branch distance*. It checkes if the input steps are exactly the needed ones (representing a solution) or how far is the chromosome from the solution. The approach level records how many steps were not executed by a particular input. The fewer steps executed, the *further away* the input is from executing the steps. The branch distance is computed using the conditions of the guards of the rule at which the evolution diverted away from the current target step.

To compute the fitness function we need to perform a simulation of the system. To simulate the system we use kPWorkbench.

3.3 Experiments

In our experiments we used the kP system presented in Example 1.

Experiment 1

consisted in generating test data for the following evolution steps:

$Steps = \{r_{10}, (r_1, r_{11}), (r_3, r_{12}), (r_4, r_{13}), (r_7, r14), r_7, r_7, r_7, r_7, t_8\}$

The size of the input set is $size = 5$. In this example we have 10 steps, but only the first 6 will receive inputs. The other steps will consume the inputs until the system reaches the final configuration.

The maximum number of evolution is set to 50.

This experiment was made to find the suitable configuration for the algorithm, including the reinsertion method. As described in Subsection 3.2, the first experiments failed. Running 100 times the algorithm for each configuration, we couldn't find a solution. Only for the new reinsertion method the algorithm was successful with a success rate of 75%. Also, the average number of evolutions needed to find a solution was 37.

There are many input sets that create the given scenario. Table 3.3 contains some examples of solutions obtained using during this experiment. The first column shows the number of generations needed to find the solution.

Evolutions	Input1	Input 2	Input 3	Input 4	Input 5
45	$1f\ 2x$	$1t$	$1d$	$1f\ 3x\ 3d$	$2x\ 2d\ 1o$
41	$1f\ 2x$	$1t\ 1d$		$2x\ 2d$	$1f\ 6x\ 3d\ 1o$
33	$1f$	$1f\ 1t\ 5x$	$1d\ 1o$	$5x$	$4d$
32	$1f\ 1t$	$1d$	$1x$	$1f\ 4x\ 5d\ 1o$	$1x$
28	$1f\ 1t\ 1d$	$1o$	$2x$	$3x\ 1d$	$1f\ 1x\ 4d$
27	$1f\ 1t$		$1d\ 1o$	$1t\ 2d$	$1f\ 3d$
43	$1f\ 1t\ 1x$	$3x$	$1d$	$2x\ 1d\ 1o$	$1f\ 4x\ 4d$

Table 2. Example of solutions for Experiment 1

Experiment 2

consisted in generating test data for other examples of evolution steps, using the configuration establish during *Experiment 1*. One of the following evolution steps set we used was:

$Steps = \{r_{10}, (r_1, r_{11}), (r_3, r_{12}), (r_4, r_{13}), (r_6, r14), r_6, r_6, r_6, r_6, t_8\}$

The size of the input set is $size = 5$. In this example we have 10 steps, but only the first 6 will receive inputs. The other steps will consume the inputs until the system reaches the final configuration. The maximum number of evolution is set to 50. Running 100 times the algorithm, we couldn't find a solution. The success rate for this example was 62% and the average number of evolutions needed to find a solution was 36.

4 Conclusions

In conclusion, we used genetic algorithms to generate test data for kP systems. The algorithm input is reperesented by a kP system model and a set of computation steps, the ouput being the set of input sets needed to create the given input scenario. The algorithm uses kP systems that behave similar to EFSMs. We tried to apply directly some algorithm defined for EFSMs, but it wasn't successful. We overcomed this problem by changing the reinsertion method of the population. With this configuration, the algorithm was successful.

As future work, we will extend this algorithm to other kP systems, starting from using different kinds of rules also.

Acknowledgements

This work is supported by a grant of the Romanian National Authority for Scientific Research, CNCS-UEFISCDI, project number PN-III-P4-ID-PCE-2016-0210.

References

1. Dragomir, C., Ipate, F., Konur, S., Lefticaru, R., Mierla, L.: Model checking kernel p systems. In: Alhazov, A., Cojocaru, S., Gheorghe, M., Rogozhin, Y., Rozenberg, G., Salomaa, A. (eds.) Membrane Computing. Lecture Notes in Computer Science, vol. 8340, pp. 151–172. Springer Berlin Heidelberg (2014)
2. Gheorghe, M., Ipate, F., Dragomir, C., Mierla, L., Valencia-Cabrera, L., García-Quismondo, M., Pérez-Jiménez, M.J.: Kernel P Systems - Version I. Eleventh Brainstorming Week on Membrane Computing (11BWMC) pp. 97–124 (2013)
3. Gheorghe, M., Ipate, F., Lefticaru, R., Pérez-Jiménez, M.J., Turcanu, A., Valencia-Cabrera, L., García-Quismondo, M., Mierla, L.: 3-col problem modelling using simple kernel P systems. International Journal of Computer Mathematics **90**(4), 816–830 (2013). https://doi.org/10.1080/00207160.2012.743712, https://doi.org/10.1080/00207160.2012.743712
4. Gheorghe, M., Ipate, F., Lefticaru, R., Turlea, A.: Testing identifiable kernel p systems using an x-machine approach. In: International Conference on Membrane Computing. pp. 142–159. Springer (2018)
5. Harman, M., McMinn, P.: A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of the 2007 international symposium on Software testing and analysis. pp. 73–83. ACM (2007)
6. Ipate, F., Gheorghe, M.: Finite state based testing of P systems. Natural Computing **8**(4), 833 (2009). https://doi.org/10.1007/s11047-008-9099-3, https://doi.org/10.1007/s11047-008-9099-3
7. Ipate, F., Gheorghe, M.: Testing non-deterministic stream X-machine models and P systems. Electronic Notes in Theoretical Computer Science **227**, 113–126 (2009). https://doi.org/10.1016/j.entcs.2008.12.107, https://doi.org/10.1016/j.entcs.2008.12.107
8. Ipate, F., Gheorghe, M., Lefticaru, R.: Test generation from P systems using model checking. Journal of Logic and Algebraic Programming **79**(6), 350–362 (2010). https://doi.org/10.1016/j.jlap.2010.03.007, https://doi.org/10.1016/j.jlap.2010.03.007
9. Kalaji, A.S., Hierons, R.M., Swift, S.: An integrated search-based approach for automatic testing from extended finite state machine (EFSM) models. Information & Software Technology **53**(12), 1297–1318 (2011)
10. Khan, M.E., Khan, F.: A comparative study of white box, black box and grey box testing techniques. International Journal of Advanced Computer Sciences and Applications **3**(6), 12–1 (2012)
11. Lefticaru, R., Gheorghe, M., Ipate, F.: An empirical evaluation of P system testing techniques. Natural Computing **10**(1), 151–165 (2011). https://doi.org/10.1007/s11047-010-9188-y, https://doi.org/10.1007/s11047-010-9188-y

12. Lefticaru, R., Ipate, F.: Automatic state-based test generation using genetic algorithms. In: Proc. SYNASC'07. pp. 188–195. IEEE Computer Society (2007)
13. Lefticaru, R., Ipate, F.: Functional search-based testing from state machines. In: First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008. pp. 525–528 (2008)
14. Mitchell, M.: An introduction to genetic algorithms. MIT Press (1998)
15. Păun, G.: Computing with membranes. Tech. rep., Turku Centre for Computer Science (1998)
16. Păun, G.: Computing with membranes. Journal of Computer and System Sciences **61**(1), 108–143 (2000). https://doi.org/10.1006/jcss.1999.1693, https://doi.org/10.1006/jcss.1999.1693
17. The P systems website. http://ppage.psystems.eu, [Online; accessed 12/05/2018]
18. Turlea, A., Ipate, F., Lefticaru, R.: A hybrid test generation approach based on extended finite state machines. In: 18th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2016, Timisoara, Romania, September 24-27, 2016. pp. 173–180 (2016). https://doi.org/10.1109/SYNASC.2016.037, https://doi.org/10.1109/SYNASC.2016.037

New applications for an old tool

Luis Valencia-Cabrera, David Orellana-Martín,
Ignacio Pérez-Hurtado, Mario J. Pérez-Jiménez

Research Group on Natural Computing
Department of Computer Science and Artificial Intelligence
Universidad de Sevilla
Avda. Reina Mercedes s/n, 41012 Sevilla, Spain
E-mail: {lvalencia, dorellana, perezh, marper}@us.es

Summary. First, the dependency graph technique, not so far from its current application, was developed trying to find the shortest computations for membrane systems solving instances of SAT. Certain families of membrane systems have been demonstrated to be non-efficient by means of the reduction of finding an accepting computation (respectively, rejecting computation) to the problem of reaching from a node of the dependency graph to another one.

In this paper, a novel application to this technique is explained. Supposing that a problem can be solved by means of a kind of membrane systems leads to a contradiction by means of using the dependency graph as a reasoning method. In this case, it is demonstrated that a single system without dissolution, polarizations and cooperation cannot distinguish a single object from more than one object.

An extended version of this work will be presented in the 20th International Conference on Membrane Computing.

1 Introduction

The *computational efficiency* of a model in a computing paradigm refers to its ability to provide polynomial time solutions for computationally hard problems, generally achieved by making use of an exponential workspace constructed in a natural way. Aspects related to the computational efficiency within membrane computing were first analyzed in 1999, with the introduction of a new computing model called *P system with active membranes* [5]. These systems are non-cooperative (the left hand side of any rule consists of only one object) and their membranes play a relevant role in computations to the extent that new membranes can be created by division rules. The membranes of these systems are supposed to have one of three possible electrical polarizations: positive, negative or neutral. In this context, it was given an *ad-hoc* solution to the Boolean satisfiability problem (SAT)

by means of such kind of P systems. More specifically, a P system with active membranes which makes use of *simple* object evolution rules (only one object is produced for this kind of rules), dissolution rules and division rules for elementary and non-elementary membranes, is associated with every instance φ of SAT. Thus, the syntactic structure of the formula is "captured" by the description of the system and, furthermore, in this context a P system can only process one instance of the problem. The solution provided runs in linear time with respect to the *size* of the input formula φ, that is, the maximum between the number of variables and the number of clauses in φ.

Usually, computational complexity theory deals with decision problems, that is, problems requiring a yes/no answer. Each decision problem has a language associated with it, in a natural way, so that solving such problems is defined through the recognition of the corresponding language. Thus, in order to describe in a formal way what solving a decision problem means, basic *recognizer transition P systems* (initially called *decision P systems*) were defined [7].

Let us recall that an abstract problem can be solved by using a single Turing machine, that is, for every instance of the problem, the Turing machine receiving the input corresponding to that instance returns the correct answer. This is due to the fact that these machines have an unlimited and unrestricted memory, given the infinite tape it includes (consisting of an infinite number of cells). Bearing in mind that the ingredients necessary to define a membrane system are finite, an abstract problem should be solved, in general, by an infinite numerable family of membrane systems, in such a way that each system in the family is in charge of processing all the instances having the same size.

It seems interesting to analyze what kind of membrane systems are capable of solving decision problems through only one unique system. In this context, it is essential to clarify how the instances of the problem are introduced into the system. Next, we consider the case in which the instances are directly introduced inside the system (*free* of resources) by means of a representation of the problem to be solved. It is important to remark that this means that the input alphabet of the P system is the same one that the alphabet of the problem, so there is no possibility of encoding, for instance, an instance of a problem from **P** to an object yes or an object no.

Definition 1. *Let $X = (I_X, \theta_X)$ be a decision problem where I_X is a language over a finite alphabet Σ_X. Let $\mathcal{R}$ be a class of recognizer membrane systems with input membrane. We say that problem X is solvable in polynomial time by a single membrane system Π from $\mathcal{R}$, free of resources, denoted by $X \in \mathbf{PMC}_{\mathcal{R}}^{1f}$, if the following holds:*

- *The input alphabet of Π is Σ_X.*
- *The system Π is* polynomially bounded *with regard to X; that is, there exists a polynomial $p(r)$ such that for each instance $u \in I_X$, every computation of the system Π with input multiset u performs at most $p(|u|)$ steps.*

- *The system Π is* sound *with regard to X; that is, for each instance $u \in I_X$, if there exists an accepting computation of the system Π with input multiset u then $\theta_X(u) = 1$.*
- *The system Π is* complete *with regard to X; that is, for each instance $u \in I_X$ such that $\theta_X(u) = 1$, every computation of the system Π with input multiset u is an accepting computation.*

From the previous definition it is easy to prove that $\mathbf{PMC}^{1f}_{\mathcal{R}} \subseteq \mathbf{PMC}_{\mathcal{R}}$, for every class $\mathcal{R}$ of recognizer membrane systems with input membrane.

2 Previous uses of the dependency graph

The dynamics of a membrane system provides, in a natural way, a tree of computation. More precisely, the *computation tree* of a membrane system Π, denoted $Comp(\Pi)$, is a rooted labelled maximal tree, whose maximal branches will be called *computations* of Π. A computation of Π is a halting computation if and only if it is a finite branch. The labels of the leaves of $Comp(\Pi)$ are called *halting configurations*.

Given a semi-uniform or uniform solution (in polynomial time) for a decision problem by means of a family of recognizer membrane systems, every instance of the problem is processed by a system of the family. This system must be confluent, so in order to know its answer for any instance it is enough to consider only one computation of such system. In this context, an exciting challenge would be looking for a computation with minimum length. For that, some *weak metrics* on the degree of closeness configurations of a membrane system with a fixed structure of membranes have been studied in [2]. In this context, in order to search for the shortest paths in a graph providing a *sound* computation of the system, the *dependency graph* associated with the set of rules of a recognizer membrane system was introduced. This concept is based on the dependence among elements of the alphabet with respect to the set of rules of the P system. Several weak metrics over the set of configurations of the system based on the concept of dependency graph were considered, starting from the notion of *distance* between two nodes of the graph (the length of the shortest path connecting v_1 and v_2, or *infinite* if there is no path from v_1 to v_2).

Also, in some kind of recognizer membrane systems, it is possible to consider a directed graph (also called *dependency graph*) verifying the following properties: (a) it can be constructed from the set or rules of the system in polynomial time, that is, in a time bounded by a polynomial function depending on the total number of rules and the maximum length of them; and (b) the accepting computations of such systems can be characterized by means of a "reachability" property in the dependency graph associated with it (the existence of a path in the graph between two specific nodes). Therefore, dependency graphs provide a technique to tackle the limits on efficient computations in membrane systems; that is, the non-efficiency of such systems.

3 Dependency graph as a technique to prove negative results in membrane systems

Let $\mathcal{R}$ be a class of recognizer membrane systems such that every system from $\mathcal{R}$ is associated with a dependency graph verifying the following property: a computation of a system from $\mathcal{R}$ is an accepting computation if and only if there exists a path between two distinguished nodes in the dependency graph associated with the system. In this situation, it is possible to show that some decision problem $X = (I_X, \theta_X)$ *cannot be solved* in polynomial time in a uniform way by means of a single membrane system, free of resources, from $\mathcal{R}$. This remark is illustrated by an example.

The `ONLY-ONE-OBJECT` problem is the decision problem $X = (I_X, \theta_X)$ defined as follows: $I_X = \{a^n \mid n \in \mathbb{N}, n \geq 1\}$ and $\theta_X(a^n) = 1$ if and only if $n = 1$. It is easy to design a deterministic Turing machine which takes two computation steps, solving the `ONLY-ONE-OBJECT` problem. Let us see that `ONLY-ONE-OBJECT` $\notin \mathbf{PMC}^{1f}_{\mathcal{AM}^0(-d,+ne)}$.

Theorem 1. *There is no recognizer membrane system from the class* $\mathcal{AM}^0(-d, +ne)$ *solving the* `ONLY-ONE-OBJECT` *problem in polynomial time by a single membrane system and free of resources.*

Proof. Let us assume that there exists a recognizer membrane system Π from $\mathcal{AM}^0(-d, +ne)$ verifying the following: (a) the input alphabet of Π is the singleton $\{a\}$; (b) every computation of Π with input multiset $\{a\}$ is an accepting computation; and (c) every computation of Π with input multiset $\{a^n\}$, for each $n > 1$, is a rejecting computation.

Let us denote by $G_{\Pi+\{a\}}$ (respectively, $G_{\Pi+\{a^n\}}$, for each $n > 1$) the dependency graph associated with the system $\Pi + \{a\}$ (resp. $\Pi + \{a^n\}$). Then, for each $n > 1$, we have $G_{\Pi+\{a\}} = G_{\Pi+\{a^n\}}$, since there would always be an edge $(s_\Pi, (a, i_{in}))$ in the dependency graph, and the rest of the graph would remain the same. Besides, every computation of $\Pi + \{a\}$ is an accepting computation if and only if every computation of $\Pi + \{a^n\}$, for each $n > 1$, is an accepting computation, which is a contradiction of the initial hypothesis, thus there cannot exist such membrane system.

4 Conclusions

Along this work, some of the main results concerning the use of dependency graphs within membrane computing to analyze the computational efficiency of computing models have been reviewed. It is worth pointing out that, albeit the **P** *versus* **NP** problem is the most important one in Computer Science, there are other interesting problems in the field of Computational Complexity Theory, also below **P**. When using polynomial precomputed resources, problems from **P** can be easily solved. But considering membrane systems *free of precomputed resources*, things change

in such a way that there cannot be trivial solutions that could be obtained at first. Then, it would be useful to study these kinds of systems to solve problems below this complexity class. In this case, with this technique it has been demonstrated that there is no solution to the `ONLY-ONE-OBJECT` problem by means of a single membrane system from $\mathcal{AM}^0(-d, +ne)$.

Adapting currently used methodologies to new applications is an interesting future research line to improve existing results and obtain new ones. Besides, the search for new techniques to demonstrate the non-efficiency or the inability for certain membrane systems to solve some decision problems is critical when addressing the **P** *versus* **NP** and other interesting problems in the field of Computational Complexity Theory.

Acknowledgements

This work was supported in part by the research project TIN2017-89842-P, cofinanced by Ministerio de Economía, Industria y Competitividad (MINECO) of Spain, through the Agencia Estatal de Investigación (AEI), and by Fondo Europeo de Desarrollo Regional (FEDER) of the European Union.

References

1. A. Alhazov, M.J. Pérez-Jiménez. Uniform solution to QSAT using polarizationless active membranes. In J. Durand-Lose, M. Margenstern (eds.) *Machines, Computations, and Universality 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007. Proceedings. Lecture Notes in Computer Science*, **4664** (2007), 122-133.
2. A. Cordón-Franco, M. A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez. Weak metrics on configurations of a P system. In Gh. Paun, A. Riscos, Á. Romero, F. Sancho (eds.) *Proceedings of the Second Brainstorming Week on Membrane Computing*, Report RGNC 01/2004, 2004, pp. 139-151.
3. M. A. Gutiérrez-Naranjo, M.J. Pérez-Jiménez, A. Riscos-Núñez, F.J. Romero-Campero. On the power of dissolution in P systems with active membranes. In R. Freund, Gh. Păun, G. Rozenberg, A. Salomaa (eds.) *Membrane Computing, 6th International Workshop, WMC 2005, Vienna, Austria, July 18-21, 2005, Revised Selected and Invited Papers. Lecture Notes in Computer Science*, **3850** (2006), 224-240.
4. C.H. Papadimitriou. *Computational Complexity*, Addison–Wesley, Massachusetts, 1995.
5. Gh. Păun. P systems with active membranes: attacking **NP**–complete problems, *Journal of Automata, Languages and Combinatorics*, **6**, 1 (2001), 75-90. A preliminary version in *Centre for Discrete Mathematics and Theoretical Computer Science, CDMTCS Research Report Series*-102, May 1999, 16 pages.
6. Gh. Păun. Further twenty six open problems in membrane computing. In M. A. Gutiérrez-Naranjo, A. Riscos-Núñez, F.J. Romero-Campero, D. Sburlan (eds.) *Proceedings of the Third Brainstorming Week on Membrane Computing*, Fénix Editora, Sevilla, 2005, 249-262.

7. M.J. Pérez-Jiménez, Á. Romero-Jiménez, F. Sancho-Caparrini. Decision P systems and the $\mathbf{P} \neq \mathbf{NP}$ conjecture. In Gh. Păun, Gr. Rozenberg, A. Salomaa, C. Zandron (eds.) Membrane Computing 2002. *Lecture Notes in Computer Science*, **2597** (2003), 388-399. A preliminary version in Gh. Păun, C. Zandron (eds.) *Pre-proceedings of Workshop on Membrane Computing 2002*, MolCoNet project-IST-2001-32008, Publication No. 1, Curtea de Arges, Romanian, August 19-23, 2002, pp. 345-354.

The DBSCAN Clustering Algorithm on P Systems

György Vaszil

Department of Computer Science, Faculty of Informatics
University of Debrecen
Kassai út 26, 4028 Debrecen, Hungary
vaszil.gyorgy@inf.unideb.hu

Summary. We show how to implement the DBSCAN clustering algorithm (Density Based Spatial Clustering of Applications with Noise) on membrane systems using evolution rules with promoters and priorities.

1 Introduction

Clustering is the process of partitioning elements of a dataset according to some similarity measure in such a way that elements in the same cluster are similar, elements in different clusters are dissimilar. Clustering analysis is widely used in several areas of data mining as a tool to discover implicit patterns and deduce knowledge based on the data, or it can also be used for preprocessing before the application of other algorithms. The reader is referred to [3] for more details about clustering, and the area of data mining in general.

The density based clustering of applications with noise (DBSCAN) clustering algorithm was introduced in [2]. It clusters data points based on density (a point is dense if it has many neighbors within a given radius. The algorithm can be summarized in the following steps:

Input: A set of points, the neighborhood radius ϵ, and the density threshold $MinPts$.

1. Mark all points "univisited".
2. Pick an unvisited point p,
   - change its mark to "visited", and
   - count the number of points in its ϵ neighborhood to see if it is a core point, that is, if the number of points in its ϵ neighborhood is at least $MinPts$.
   - If p is not a core point, mark it as "noise", otherwise create C as a new cluster and add p to C, together with those points in its ϵ neighborhood which do not belong to any cluster yet.

3. Pick an unvisited point p' in C
   - change the mark of p' to "visited",
   - count the number of points in its ϵ neighborhood to see if it is a core point.
   - If p' is a core point, add those points to C in its ϵ neighborhood which do not belong to any cluster yet.
   - If there are unvisited points in C, go back to 3.
4. If there are unvisited points in the data set, go back to 2.

Output: The clustering result.

In the following we intend to give an implementation of this algorithm in terms of P systems. The system will use evolution rules with promoters and priorities. Our goal is to exploit the parallelism of P systems in order to parallelize, and thus, to speed up the DBSCAN algorithm which in its original version works with $O(n^2)$ time complexity on a sequential machine (where n is the number of points to be clustered). On P systems, the running time can be reduced to $O(n)$.

Ours is not the only proposal for clustering with P systems. As examples, see [4], or see [9] for a so called k-nearest base clustering algorithm on P systems with active membranes, and [8] for clustering with splicing P systems. Even a DBSCAN algorithm implementation was already presented in [11], we believe however, that our present proposal is conceptually simpler.

2 Preliminaries and Definitions

Let O be a finite nonempty set (the set of object) and $\mathbb{N}$ be the set of non-negative integers. A *multiset* M (or an *multiset* M for short), over O is a pair (O, f), where $f : O \to \mathbb{N}$ is mapping which gives the *multiplicity* of each object $a \in O$. The set $supp(M) = \{a \in O \mid f(a) > 0\}$ is called the *support* of M. If $supp(M) = \emptyset$, then M is the empty multiset. If $a \in supp(M)$, then $a \in M$, and $a \in^n M$ if $f(a) = n$. In the following we represent a multiset M over $O = \{a_1, \ldots, a_k\}$ by the string $a_1^{M(a_1)} \ldots a_k^{M(a_k)}$ (or any of its permutations).

Membrane systems, or P systems, were introduced in [5] as computing models inspired by the functioning of the living cell. The main component of such a system is a membrane structure with membranes enclosing regions as multisets of objects. Each membrane has an associated set of operators working on the objects contained by the region. These operators can be of different types, they can change the objects present in the regions or they can provide the possibility of transferring the objects from one region to another one. The evolution of the objects inside the membrane structure from an initial configuration to a final configuration corresponds to a computation having a result which is derived from some properties of the final configuration.

Several variants of the basic notion have been introduced and studied proving the power of the framework, see the monograph [6] for a comprehensive introduc-

tion, the handbook [7] for a summary of notions and results of the area, and the volumes [1, 10] for various applications.

An $n+3$-tuple $\Pi = (O, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho)$ is a P *system* of degree n, where

- O is a finite set called the alphabet of objects Π;
- w_i, $1 \leq i \leq n$, is a finite multiset of objects containing the initial contents of compartment i of Π;
- R_i, $1 \leq i \leq n$, is a finite set of rules of the form $u \to v$ or $u \to v|_\alpha$ with $u, \alpha \in O^*$ and $v \in O \cup \{here, in, out\}$;
- $\rho \subset R \times R$ is a priority relation defined on the rules of $R = \bigcup_{1 \leq i \leq n} R_i$ which may also be empty.

For a P system $\Pi = (O, w_1, \ldots, w_n, R_1, \ldots, R_n, \rho)$ as above, an n-tuple $c = (u_1, \ldots, u_n)$ with $u_i \in O^*$ for each i, is called a *configuration* of Π and $c_0 = (w_1, \ldots, w_n)$ is called its *initial configuration*. The multisets $u_1, \ldots, u_n$ are also called the *contents* of compartments $1, \ldots, n$, in configuration c.

A P system changes its configurations by applying its rules in the so-called maximally parallel manner. A multiset of rules from R_i for some $1 \leq i \leq n$, as given above, is applicable to a configuration c, if and only if the union of the multisets on the lefthand sides of the rules is a submultiset of u_i, the contents of the ith region. As a result of applying the rules to c, each object of the multisets on the righthand sides of the rules replace the objects on the lefthand side. Moreover, if the objects are moved to the respective neighboring regions according to the target indicators $here, in, out$. Rules multisets are applied in all regions in parallel, producing a series of configuration changes.

We say that the configuration $c' = (v_1, \ldots, v_n)$ of Π is obtained directly from $c = (u_1, \ldots, u_n)$ by applying the rules in a maximally parallel manner, if the rule multisets applied in the regions are maximal, that is, by adding any rule to the multiset, they are not applicable simultaneously any more. A rule of the form $x \to y|\alpha$ is applicable only if $\alpha \in O^*$ (the promoter mutiset) is a submutiset of the respective region.

When the priority relation ρ is nonempty, we denote by $r_1 > r_2$ if $(r_1, r_2) \in \rho$, that is, if a rule r_1 has higher priority than r_2. In such a case, r_2 can only be applied to configurations where r_1 is not applicable.

A sequence of configurations $c_0, c_1, \ldots$ of Π, obtained directly from each other and starting from the initial configuration, is called a computation. The computation halts if no rule can be applied in the current configuration. The result of a halting computation are the multisets of objects in the compartments at halting.

3 Implementing the DBSCAN Algorithm

In order to perform the clustering algorithm, let us construct a P system $\Pi = (O, [\,], w, R, \rho)$ with

$$O = \{p_i, p'_i, p_{i,j?}, p_{i,j}, p'_{i,j}, p_i^{cr}, p_i^{ns}, p_{i,j?}^{ns}, p_{i,j}^{cr}, p_{i,j}^{ncr} \mid 1 \leq i,j \leq n\} \cup \{E_i, H_i \mid 1 \leq i \leq n\} \cup \{A, B, C, D, F, F', F''\}.$$

The objects of the form p_i represent the n points of the data set. We assume that we have a distance function $d : \{p_1, \ldots, p_n\}^2 \to \mathbb{N}$.

The initial contents of the system is the multiset

$$w = Ap_1 \ldots p_n,$$

corresponding to the n points, and a synchronizing symbol A.

Now we present the rules of R and at the same time, describe the functioning of the system. Let $R = R_{pick} \cup R_{cehck} \cup R_{mark} \cup R_{check2} \cup R_{mark2}$, and let us describe these rule sets as follows.

$$R_{pick} = \{Ap_i \to Bp'_i \mid 1 \leq i \leq n\}.$$

Using the single occurrence of A, the application of one of these rules picks a point p_i for some $1 \leq i \leq n$ by changing it to its primed version p'_i. Now, in the presence of p'_i, the system checks whether the ith point is dense, that is, whether it has more than $MinPts$ points in its ϵ neighborhood. This is achieved by the rules

$$R_{check} = \{p_k \to E_i p_{k,i?}|_{Bp'_i}, p_{k,j}^{ncr} \to E_i p_{k,j}^{ncr}|_{Bp'_i}, p_k^{ns} \to E_i p_{k,i?}^{ns}|_{Bp'_i} \mid \text{ for } 1 \leq i,j,k \leq n, \text{ such that } d(p_i,p_k) < \epsilon\} \cup \{B \to C\}$$

where $d(p_i, p_k)$ denotes the distance between the locations of the ith and the kth points. The application of these rules produce a number of E_i objects which is equal to the number of points that are in the ϵ neighborhood of the ith point.

Now we mark the point corresponding to p'_i core or non-core, depending on the number of its ϵ neighbors with the following rules. If $m = MinPts$, then we have

$$\begin{aligned} R_{mark} = {} & \{p'_i \to p_i^{cr}|_{C(E_i)^m} > p'_i \to p_i^{ns}|_C \mid 1 \leq i \leq n\} \cup \\ & \{p_{j,i?} \to p'_{j,i}|_{Dp_i^{cr}}, p_{j,i?}^{ns} \to p'_{j,i}|_{Dp_i^{cr}} \mid 1 \leq i,j \leq n\} \cup \\ & \{p_{j,i?} \to p_j|_{Dp_i^{ns}}, p_{j,i?}^{ns} \to p_j^{ns}|_{Dp_i^{ns}} \mid 1 \leq i,j \leq n\} \cup \\ & \{E_i \to \varepsilon|_C \mid 1 \leq i \leq n\} \cup \{C \to D, D \to F\} \end{aligned}$$

where the symbol $>$ shows the priority among the first group of rules in R_{mark}. The rules here are used for two consecutive steps: First, the chosen point is marked core or noise (based the number of points in its ϵ neighborhood) by changing p'_i to p_i^{cr} or to p_i^{ns} depending on the number of E_i symbols that are present (these were created in the previous step, their number corresponds to the number of neighbors). Second, if the point is marked core, its ϵ neighborhood is also added to this cluster (the cluster denoted by i).

The next group of rules serves to see if the recently created cluster (cluster i) can be expanded further.

$$R_{check2} = \{p^{cr}_{j,i} \to H_i p^{cr}_{j,i}|_{Fp'_{k,i}}, p^{ncr}_{j,i} \to H_i p^{ncr}_{j,i}|_{Fp'_{k,i}},\ p^{ns}_{j} \to H_i p^{ns}_{j,i?}|_{Fp'_{k,i}},$$
$$p_j \to H_i p_{j,i?}|_{Fp'_{k,i}} \mid \text{ for } 1 \le i,j,k \le n, \text{ such that } d(p_j,p_k) < \epsilon\} \cup$$
$$\{F \to F'\}.$$

These rules examine the neighborhood of the kth point which has recently been added to the cluster denoted by i. The number of H_i symbols is the same as the number of points in the ϵ neighborhood of point (k).

The next group of rules is the following.

$$\begin{aligned} R_{mark2} = {} & \{p'_{k,i} \to p^{cr}_{k,i}|_{F'(H_i)^m} > p'_{k,i} \to p^{ncr}_{k,i}|_{F'} \mid 1 \le i,k \le n\} \cup \\ & \{p_{j,i?} \to p'_{j,i}|_{F''p^{cr}_{k,i}}, p^{ns}_{j,i?} \to p'_{j,i}|_{F''p^{cr}_{k,i}} \mid 1 \le i,j,k \le n\} \cup \\ & \{p_{j,i?} \to p_j|_{F''p^{ncr}_{k,i}}, p^{ns}_{j,i?} \to p^{ns}_{j}|_{F''p^{ncr}_{i}} \mid 1 \le i,j \le n\} \cup \\ & \{H_i \to \varepsilon|_{F'} \mid 1 \le i \le n\} \cup \{F' \to F''\} \\ & \{F'' \to F|_{p'_{k,i}} > F'' \to A \mid 1 \le i,k \le n\}, \end{aligned}$$

where $m = MinPts$, as before. Similarly to R_{mark}, these rules decide whether the cluster denoted by i should be expanded with the elements of the ϵ neighborhood of point (k). If the number of neighbors is sufficient, they are added to the cluster, and also marked for further investigation. If there are points which are newly added to the cluster, that is, if further neighborhood checks are necessary, then the symbol F' is changed to F'', and then back to F, so the rules in R_{check2} become applicable again, and the checking process can repeated. If no new points are added to the cluster, F'' is changed to A, so the rules in R_{check} are activated, and the search for additional clusters can start with the identification of a yet unclassified point by R_{pick}.

To see how the system Π operates, consider the initial configuration

$$Ap_1 \dots p_n.$$

By applying a rule $Ap_i \to Bp'_i \in R_{pick}$ for some $1 \le i \le n$, a not yet classified point (i) is chosen from the point set $(1), \dots, (n)$, and we obtain

$$Bp_1 \dots p'_i \dots p_n.$$

To show how the system works, we start with a more general case

$$Bx_1 \dots p'_i \dots x_n,$$

where $x_i \in \{p_i, p^{ns}_i, p^{cr}_i, p^{cr}_{i,j}, p^{ncr}_{i,j}\}$.

Now, because B is present, the rules of R_{check} are applicable, so we get

$$C \dots p'_i \dots y_1 E_i \dots y_l E_i \dots$$

where $y_j \in \{p_{k,i?}, p^{ns}_{k,i?} \mid \text{for some } 1 \le k \le n\}$, $1 \le j \le l$. All symbols which correspond to unclassified points or noise points (k) in the ϵ neighborhood of

point (i) are marked as candidates for inclusion in the same cluster as (i) (denoted by the index $i?$).

If the number of neighbors (equal to the number of E_i symbols) is not sufficient (less than $MinPts$), we get

$$D \dots p_i^{ns} \dots y_1 \dots y_l \dots,$$

and then

$$F x_1 \dots p_i^{ns} \dots x_n$$

by the rules of R_{mark}. Now F is changed to F', F'', and then back to A, when the rules of R_{pick} become applicable again, and the system continues with choosing an other point for examination.

Otherwise, if the number of neighbors of point (i) are sufficient, we get

$$D \dots p_i^{cr} \dots y_1 \dots y_l \dots,$$

and then

$$F \dots p_i^{cr} \dots p'_{X_1,i} \dots p'_{X_l,i} \dots,$$

where $1 \leq X_j \leq n$, $1 \leq j \leq l$, marking the point (i) as core point, and marking its neighbors as members of the cluster denoted by i.

Now, the newly added points $(X_1), \dots, (X_l)$, corresponding to the symbols $p'_{X_1,i}, \dots, p'_{X_l,i}$ have to be checked, which is done by the rules of R_{check2} If the ϵ neighborhood of a point (k) contains a sufficient number of points, then similarly to R_{check}, the rules of R_{check2} introduce a sufficient number of H_i symbols for the rule $p'_{k,i} \to p^{cr}_{k,i}$ to be applicable, and point (k) is marked as a core point, denoted by a corresponding symbol $p^{cr}_{k,i}$. Otherwise, if the number of neighbors is not sufficient, then point (k) is marked as non-core, denoted by the symbol $p^{ncr}_{k,i}$ introduced by the rule $p'_{k,i} \to p^{ncr}_{k,i}$.

These checks are executed in parallel for all $p'_{X_k,i}$, resulting in marking some of them core, some of them non-core, and priming all neighbors of core points in two computational steps, using the rules in R_{check2} and R_{mark2}. If the result contains primed points, that is, points that have to be checked by counting the number of their neghbors, then F'' is rewritten to F, so the process can start all over again, otherwise it is rewritten to A, meaning that the cluster denoted by i cannot be expanded any more, the search for new clusters can begin by picking a new point using the rules of R_{pick}.

If all points are classified, then no rule of R_{pick} can be used, the system halts in a configuration containing the symbol A, and with all points (i), $1 \leq i \leq n$, having a corresponding symbol, which is either

- p_i^{cr} : point (i) is a core point, belonging to the cluster denoted by i,
- $p_{i,j}^{cr}$: point (i) is a core point, belonging to the cluster denoted by j,
- $p_{i,j}^{ncr}$: point (i) is a non-core point, belonging to the cluster denoted by j,
- p_i^{ns} : point (i) is a noise point, it does not belong to any cluster.

4 Conclusion

We have shown how to implement the DBSCAN clustering algorithm on P systems. The model we used worked with evolution rules, promoters and priorities, Due to the parallel nature of P systems, our implementation has a time complexity of $O(n)$ which is clear improvement compared to the time complexity of $O(n^2)$ of a sequential DBSCAN implementation. Our P system implementation presented in in this paper is not the first one, but we believe that it is conceptually more simple than the implementation presented in [11].

Considering the parallelity of our approach, the points of the dataset are examined by the algorithm one-by-one, but the number of neighbors of the examined points is calculated in parallel. Moreover, all points examined in step 3 of the algorithm (see the introduction for the numbering of the steps of the algorithm) are examined in parallel which further reduces the time complexity of the P system implementation. An interesting challenge would be to find a way in which the parallelity of the algorithm can further be increased, and thus, the time complexity further reduced.

Acknowledgments

This research is supported in part by project no. K 120558 of the National Research, Development and Innovation Fund of Hungary, financed under the K 16 funding scheme, and by the EFOP-3.6.1-16-2016-00022 project, co-financed by the European Union and the European Social Fund.

References

1. Ciobanu, G., Pérez-Jiménez, M.J., Păun, G. (eds.): Applications of Membrane Computing. Natural Computing Series, Springer (2006)
2. Ester, M., Kriegel, H.P., Sander, J., Xu, X.: A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In: Proceedings of the Second International Conference on Knowledge Discovery and Data Mining. pp. 226–231. KDD'96, AAAI Press (1996)
3. Han, J., Kamber, M., Pei, J.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edn. (2011)
4. Jiang, Y., Peng, H., Huang, X., Zhang, J., Shi, P.: A novel clustering algorithm based on p systems. International journal of innovative computing, information & control: IJICIC 10, 753–765 (01 2014)
5. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108 – 143 (2000)
6. Păun, G.: Membrane Computing: An Introduction. Springer-Verlag, Berlin, Heidelberg (2002)
7. Păun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)

8. Xu, J., Liu, X., Xue, J.: Cluster analysis by a class of splicing p systems. In: Park, J.J.J.H., Pan, Y., Kim, C.S., Yang, Y. (eds.) Future Information Technology. pp. 575–581. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
9. Xue, J., Liu, X.: A k-nearest based clustering algorithm by P systems with active membranes. JSW 9(3), 716–725 (2014), https://doi.org/10.4304/jsw.9.3.716-725
10. Zhang, G., Pérez-Jiménez, M.J., Gheorghe, M.: Real-life Applications with Membrane Computing. Springer Publishing Company, Incorporated (2017)
11. Zhao, Y., Liu, X., Li, X.: An improved DBSCAN algorithm based on cell-like P systems with promoters and inhibitors. PLoS ONE 13, e0200751 (Dec 2018)

Author Index